KALRAY

Kalray kv3 V2 VLIW Core Architecture Manual

Kalray S.A.

Instruction Set Architecture, Micro-Architecture, and Application Binary Interface, of the Kalray kv3 core.

1676 pages

This document and the information therein are the exclusive property of Kalray SA.

Kalray – all rights reserved

Kalray kv3 V2 VLIW Core Architecture Manual

Kalray S.A.[1]

[1] info@kalray.eu, Kalray S.A.

Abstract: Instruction Set Architecture, Micro-Architecture, and Application Binary Interface, of the Kalray kv3 core.

Keywords: VLIW, DSP, ABI

Contents

1 Core Overview **33**
1.1 Datapath Organization . . . 33
1.1.1 Pipelined Implementation . . . 34
1.1.2 Prefetch Buffer (PFB) . . . 34
1.1.3 Branch and Control Unit (BCU) . . . 34
1.1.4 Arithmetic and Logic Units (ALUs) . . . 35
1.1.5 Multiply-Accumulate Unit (MAU) . . . 35
1.1.6 Floating-Point Unit (FPU) . . . 35
1.1.7 Load/Store Unit (LSU) . . . 36
1.2 Data Types and Representations . . . 36
1.2.1 Signed Integers . . . 37
1.2.2 Unsigned Integers . . . 37
1.2.3 Floating-Point Numbers . . . 37
1.2.4 Data Transfers . . . 38
1.2.5 Materialization of Constants . . . 38
1.3 VLIW Instruction Bundles . . . 38
1.3.1 Instruction Bundle Memory Structure . . . 38
1.3.2 Instruction Bundles in Assembly Language . . . 39
1.3.3 Instruction Bundle Resource Constraints . . . 39
1.4 Memory Subsystem . . . 39
1.4.1 Processor Cache Arbiter (PCA) . . . 40
1.4.2 Instruction Cache (IC) . . . 40
1.4.3 Data Cache (DC) . . . 40

2 Architectural State **42**
2.1 General Purpose Registers . . . 42
2.1.1 GPR Loads and Stores . . . 42
2.1.2 GPRs as Address Registers . . . 42
2.2 Execution Privilege Levels . . . 43
2.3 System Function Registers . . . 43
2.3.1 Program Counter (PC) . . . 45

Kalray – all rights reserved

2.3.2 Processing Status (PS) . . . 45
2.3.3 Processing Configuration Register (PCR) . . . 48
2.3.4 Return Address (RA) . . . 50
2.3.5 Compute Status (CS) . . . 50
2.3.6 Compute Status Interrupt (CSIT) . . . 51
2.3.7 Arithmetic Exception Saved PC (AESPC) . . . 53
2.3.8 Hardware Loop Registers (LS, LE, LC) . . . 53
2.3.9 Inter Process Event (IPE) . . . 53
2.3.10 Miscellaneous External Notifications (MEN) . . . 54
2.3.11 Performance Monitor Registers (PM*) . . . 54
2.3.12 Real Time Clock and Watchdog Registers (T*) (W*) . . . 54
2.3.13 Memory Translation Registers (MMC, TEL and TEH) . . . 54
2.3.14 Syscall Owners (SYO) . . . 55
2.3.15 Hardware Trap Owners (HTO) . . . 55
2.3.16 Interrupts Owners (ITO) . . . 57
2.3.17 Debug Owner (DO) . . . 57
2.3.18 Miscellaneous Owners (MO) . . . 58
2.3.19 Processor State Owner (PSO) . . . 60
2.3.20 Debug Control (DC0) . . . 62
2.3.21 Debug Control (DC1) . . . 63
2.3.22 Debug Control (DC2) . . . 63
2.3.23 Debug Control (DC3) . . . 64
2.3.24 Debug Breakpoint Address (DBA*) . . . 64
2.3.25 Debug Watchpoint Address (DWA*) . . . 64
2.3.26 Memory Error Status (MES) . . . 65
2.3.27 Wake-up Status (WS) . . . 65
2.3.28 Shadow Program Counter (SPC_PL*) . . . 65
2.3.29 Shadow Processing Status (SPS_PL*) . . . 66
2.3.30 Excepting Address (EA_PL*) . . . 66
2.3.31 Exception Vector (EV_PL*) . . . 66
2.3.32 System-Reserved Registers (SR_PL*) . . . 66
2.3.33 Debug Support Registers . . . 66
2.3.34 Exception Syndrome Register (ES_PL*) . . . 67
2.3.35 Write aliases to read only SFRs owner registers . . . 74
2.3.36 Aliases to banked SFR registers . . . 74

3 System Control **75**
3.1 Privilege Level Transitions . . . 75
3.2 Privilege Level Ownership . . . 76
3.3 Privilege Level Manipulations . . . 77
3.4 Idle Modes and Transitions . . . 79

4 Memory Management Unit **81**
4.1 MMU Overview . . . 81

Kalray – all rights reserved

4.1.1 MMU Main Features . . . 81
4.1.2 Memory Management Control (MMC) Register . . . 81
4.1.3 TLB Entry Low (TEL) and High (TEH) Registers . . . 83
4.1.4 MMU Trap Causes . . . 84
4.2 Translation Lookaside Buffers (TLBs) . . . 85
4.2.1 Architectural and Micro TLBs . . . 85
4.2.2 TLB Format (TLBF) . . . 87
4.2.3 Configuration of supported Page Sizes . . . 89
4.3 TLB Maintenance Instructions . . . 89
4.3.1 Writing to Architectural TLB Entries . . . 89
4.3.2 Reading from Architectural TLB Entries . . . 90
4.3.3 Probing Architectural TLB Entries . . . 90
4.4 Special Memory Accesses . . . 90
4.4.1 Speculative Memory Instructions . . . 91
4.4.2 Unaligned Accesses . . . 91
4.4.3 Global Cache maintenance instructions . . . 91
4.4.4 Line Cache maintenance instructions . . . 91
4.4.5 Atomic instructions . . . 92
4.5 32-bits and 64-bits modes . . . 92
4.6 TLB Shootdown Procedure . . . 92
4.7 Default memory space mapping . . . 92
4.8 Local Cluster address space . . . 93
4.9 Virtual Spaces . . . 93

5 Integrated Peripherals 95
5.1 Interrupt Controller . . . 95
5.1.1 Interrupt Controller Overview . . . 95
5.1.2 Interrupt Triggering . . . 95
5.1.3 ITC Control SFRs . . . 96
5.2 Timers and Watchdog . . . 96
5.2.1 Timers Value Registers (T0V and T1V) . . . 96
5.2.2 Timers Reload Registers (T0R and T1R) . . . 97
5.2.3 Timers Control Register (TCR) . . . 97
5.2.4 Timers Programming . . . 98
5.2.5 Watchdog Programming . . . 98
5.3 Performance Monitoring . . . 99
5.3.1 Performance Monitor Counters . . . 99
5.3.2 Performance Monitor Count Registers (PM0 to PM7) . . . 99
5.3.3 Performance Monitor Control Registers (PMC and PMC2) . . . 99
5.4 On Chip Emulation (OCE) . . . 109
5.5 Cycle counter . . . 109
5.6 Sequence Rupture helper PC . . . 109

6 Instruction Set Description 110

Kalray – all rights reserved

6.1 Instruction Syntax . . . 110
6.2 Instruction Bundle Encoding . . . 110
6.2.1 Structure of Instruction Bundles . . . 110
6.2.2 Immediate Extensions in Instruction Bundles . . . 111
6.3 Instruction Execution . . . 111
6.4 BCU Instructions . . . 114
6.4.1 AWAIT: Wait for any event or interrupt. . . . 114
6.4.2 BARRIER: Instruction Pipeline Barrier . . . 115
6.4.3 BREAK: Debugger Breakpoint . . . 116
6.4.4 CALL: Call Subroutine PC-Relative . . . 117
6.4.5 CB: Conditional Branch PC-Relative . . . 118
6.4.6 ERROP: Error Opcode . . . 119
6.4.7 GET: Get System Register . . . 120
6.4.8 GOTO: Branch Unconditional PC-Relative . . . 121
6.4.9 ICALL: Indirect Call from Register . . . 122
6.4.10 IGET: Indirect Get System Register . . . 123
6.4.11 IGOTO: Indirect Branch from Register . . . 124
6.4.12 LOOPDO: Hardware Loop setup and Do execute . . . 125
6.4.13 RET: Return from Call . . . 126
6.4.14 RFE: Return from Exception . . . 127
6.4.15 RSWAP: General and System Register Swap . . . 128
6.4.16 SCALL: System Call . . . 130
6.4.17 SET: Set System Register . . . 132
6.4.18 SLEEP: Wait for an eligible interrupt. . . . 135
6.4.19 STOP: Wait for power controller wake-up. . . . 136
6.4.20 SYNCGROUP: Synchronize Group of Cores . . . 137
6.4.21 TLBDINVAL: Invalidate Data Micro-TLB. . . . 138
6.4.22 TLBIINVAL: Invalidate Instruction Micro-TLB. . . . 139
6.4.23 TLBPROBE: Probe TLB . . . 140
6.4.24 TLBREAD: Read TLB Entry . . . 141
6.4.25 TLBWRITE: TLB Entry Write . . . 142
6.4.26 WAITIT: Wait for Interrupt . . . 143
6.4.27 WFXL: Effects on Least Significant Word of System Register . . . 144
6.4.28 WFXM: Word Effects on Most Significant Word of System Register . . . 146
6.4.29 XACCESSO: Access from Extension into Octuple Word . . . 148
6.4.30 XALIGNO: Align Extension into Octuple Word . . . 154
6.5 LSU Instructions . . . 160
6.5.1 ACSWAPD: Atomic Compare and Swap Double Word . . . 160
6.5.2 ACSWAPQ: Atomic Compare and Swap Quadruple Word . . . 163
6.5.3 ACSWAPW: Atomic Compare and Swap Word . . . 166
6.5.4 ALADDD: Atomic Load and Add Double Word . . . 169
6.5.5 ALADDW: Atomic Load and Add Word . . . 172
6.5.6 ALCLRD: Atomic Load and Clear Double Word . . . 175
6.5.7 ALCLRW: Atomic Load and Clear Word . . . 178

Kalray – all rights reserved

6.5.8 ALD: Atomic Load Double Word . . . 181
6.5.9 ALW: Atomic Load Word . . . 184
6.5.10 ASD: Atomic Store Double Word . . . 187
6.5.11 ASW: Atomic Store Word . . . 190
6.5.12 COPYO: Copy Octuple Word . . . 193
6.5.13 D1INVAL: Data Cache 1 Invalidate . . . 194
6.5.14 DFLUSHL: Data Cache Flush Line . . . 195
6.5.15 DFLUSHSW: Data Cache Flush Line by Set and Way . . . 199
6.5.16 DINVALL: Data Cache Invalidate Line . . . 200
6.5.17 DINVALSW: Data Cache Invalidate Line by Set and Way . . . 204
6.5.18 DPURGEL: Data Cache Purge Line . . . 205
6.5.19 DPURGESW: Data Cache Purge Line by Set and Way . . . 209
6.5.20 DTOUCHL: Data Cache Touch Line . . . 210
6.5.21 FENCE: Memory Fence . . . 214
6.5.22 I1INVAL: Instruction Cache 1 Invalidate . . . 215
6.5.23 I1INVALS: Instruction Cache 1 Invalidate Set . . . 216
6.5.24 LBS: Load Byte Sign Extended . . . 220
6.5.25 LBZ: Load Byte Zero Extended . . . 227
6.5.26 LD: Load Double Word . . . 234
6.5.27 LHS: Load Half Word Sign Extended . . . 241
6.5.28 LHZ: Load Half Word Zero Extended . . . 248
6.5.29 LO: Load Octuple Word . . . 255
6.5.30 LQ: Load Quadruple Word . . . 265
6.5.31 LWS: Load Word Sign Extended . . . 272
6.5.32 LWZ: Load Word Zero Extended . . . 279
6.5.33 SB: Store Byte . . . 286
6.5.34 SD: Store Double Word . . . 293
6.5.35 SH: Store Half Word . . . 300
6.5.36 SO: Store Octuple Word . . . 307
6.5.37 SQ: Store Quadruple Word . . . 317
6.5.38 SW: Store Word . . . 324
6.5.39 XLO: Extension Load Octuple Word . . . 331
6.5.40 XPLB: Extension Preload Byte . . . 348
6.5.41 XPLD: Extension Preload Double Word . . . 366
6.5.42 XPLH: Extension Preload Half . . . 384
6.5.43 XPLO: Extension Preload Octuple Word . . . 402
6.5.44 XPLQ: Extension Preload Quadruple Word . . . 420
6.5.45 XPLW: Extension Preload Word . . . 438
6.5.46 XSO: Extension Store Octuple Word . . . 456
6.6 ALU Instructions . . . 463
6.6.1 ABDBO: Absolute Difference Byte Octuple . . . 463
6.6.2 ABDD: Absolute Difference of Double Words . . . 465
6.6.3 ABDHQ: Absolute Difference Half Words Quadruple . . . 470
6.6.4 ABDSBO: Absolute Difference Saturated of Byte Octuple . . . 472

Kalray – all rights reserved

6.6.5 ABDSD: Absolute Difference Saturated of Double Words 474
6.6.6 ABDSHQ: Absolute Difference Saturated of Half Words Quadruple 476
6.6.7 ABDSW: Absolute Difference Saturated of Words 478
6.6.8 ABDSWP: Absolute Difference Saturated of Words Packed 480
6.6.9 ABDUBO: Absolute Difference Byte Octuple 482
6.6.10 ABDUD: Absolute Difference Unsigned of Double Words 484
6.6.11 ABDUHQ: Absolute Difference Unsigned Half Words Quadruple 486
6.6.12 ABDUW: Absolute Difference Unsigned of Words 488
6.6.13 ABDUWP: Absolute Difference Unsigned Words Packed 490
6.6.14 ABDW: Absolute Difference of Words 492
6.6.15 ABDWP: Absolute Difference Words Packed 495
6.6.16 ADDBO: Add Byte Octuple 497
6.6.17 ADDCD: Add Double Word to Double Word with Carry 499
6.6.18 ADDCD.I: Add Double Word to Double Word with Carry Initial 501
6.6.19 ADDD: Add Double Word to Double Word 503
6.6.20 ADDHQ: Add Half Words Quadruple 508
6.6.21 ADDRBOD: Add Reduce Byte Octuple to Double Word 510
6.6.22 ADDRHQD: Add Reduce Half Word Quadruple to Double Word 511
6.6.23 ADDRWPD: Add Reduce Word Pair to Double Word 512
6.6.24 ADDSBO: Add Saturated Byte Octuple 513
6.6.25 ADDSD: Add Saturated Double Words 515
6.6.26 ADDSHQ: Add Saturated Half Words Quadruple 517
6.6.27 ADDSW: Add Saturated Words 519
6.6.28 ADDSWP: Add Saturated Words Packed 521
6.6.29 ADDURBOD: Add Unsigned Reduce Byte Octuple to Double Word 523
6.6.30 ADDURHQD: Add Unsigned Reduce Half Word Quadruple to Double Word 524
6.6.31 ADDURWPD: Add Unsigned Reduce Word Pair to Double Word 525
6.6.32 ADDUSBO: Add Unsigned Saturated Byte Octuple 526
6.6.33 ADDUSD: Add Unsigned Saturated Double Words 528
6.6.34 ADDUSHQ: Add Unsigned Saturated Half Words Quadruple 530
6.6.35 ADDUSW: Add Unsigned Saturated Words 532
6.6.36 ADDUSWP: Add Unsigned Saturated Words Packed 534
6.6.37 ADDUWD: Add Unsigned Word to Double Word 536
6.6.38 ADDW: Add Word to Word 538
6.6.39 ADDWD: Add Word to Double Word 541
6.6.40 ADDWP: Add Words Packed 543
6.6.41 ADDX16BO: Add Integer to Integer Times 16 Byte Octuple 545
6.6.42 ADDX16D: Add Double Word to Double Word Times 16 547
6.6.43 ADDX16HQ: Add Integer to Integer Times 16 Half Words Quadruple 549
6.6.44 ADDX16UWD: Add Unsigned Word Times 16 to Double Word 551
6.6.45 ADDX16W: Add Word to Word Times 16 553
6.6.46 ADDX16WD: Add Word Times 16 to Double Word 555
6.6.47 ADDX16WP: Add Integer to Integer Times 16 Words Packed 557
6.6.48 ADDX2BO: Add Integer to Integer Times 2 Byte Octuple 559

Kalray – all rights reserved

6.6.49 ADDX2D: Add Double Word to Double Word Times 2 561
6.6.50 ADDX2HQ: Add Integer to Integer Times 2 Half Words Quadruple 563
6.6.51 ADDX2UWD: Add Unsigned Word Times 2 to Double Word 565
6.6.52 ADDX2W: Add Word to Word Times 2 567
6.6.53 ADDX2WD: Add Word Times 2 to Double Word 569
6.6.54 ADDX2WP: Add Integer to Integer Times 2 Words Packed 571
6.6.55 ADDX32D: Add Double Word to Double Word Times 32 573
6.6.56 ADDX32UWD: Add Unsigned Word Times 32 to Double Word 575
6.6.57 ADDX32W: Add Word to Word Times 32 577
6.6.58 ADDX32WD: Add Word Times 32 to Double Word 579
6.6.59 ADDX4BO: Add Integer to Integer Times 4 Byte Octuple 581
6.6.60 ADDX4D: Add Double Word to Double Word Times 4 583
6.6.61 ADDX4HQ: Add Integer to Integer Times 4 Half Words Quadruple 585
6.6.62 ADDX4UWD: Add Unsigned Word Times 4 to Double Word 587
6.6.63 ADDX4W: Add Word to Word Times 4 589
6.6.64 ADDX4WD: Add Word Times 4 to Double Word 591
6.6.65 ADDX4WP: Add Integer to Integer Times 4 Words Packed 593
6.6.66 ADDX64D: Add Double Word to Double Word Times 64 595
6.6.67 ADDX64UWD: Add Unsigned Word Times 64 to Double Word 597
6.6.68 ADDX64W: Add Word to Word Times 64 599
6.6.69 ADDX64WD: Add Word Times 64 to Double Word 601
6.6.70 ADDX8BO: Add Integer to Integer Times 8 Byte Octuple 603
6.6.71 ADDX8D: Add Double Word to Double Word Times 8 605
6.6.72 ADDX8HQ: Add Integer to Integer Times 8 Half Words Quadruple 607
6.6.73 ADDX8UWD: Add Unsigned Word Times 8 to Double Word 609
6.6.74 ADDX8W: Add Word to Word Times 8 611
6.6.75 ADDX8WD: Add Word Times 8 to Double Word 613
6.6.76 ADDX8WP: Add Integer to Integer Times 8 Words Packed 615
6.6.77 ANDD: Bitwise And Between Double Words 617
6.6.78 ANDND: Bitwise And with Not Operand Between Double Words 622
6.6.79 ANDNW: Bitwise And with Not Operand Between Words 627
6.6.80 ANDRBOD: Bitwise AND Reduce Byte Octuple to Double Word 630
6.6.81 ANDRHQD: Bitwise AND Reduce Half Word Quadruple to Double Word . . 631
6.6.82 ANDRWPD: Bitwise AND Reduce Word Pair to Double Word 632
6.6.83 ANDW: Bitwise And Between Words 633
6.6.84 AVGBO: Average of Byte Octuple 636
6.6.85 AVGHQ: Average of Half Words Quadruple 638
6.6.86 AVGRBO: Average Rounded of Byte Octuple 640
6.6.87 AVGRHQ: Average Rounded of Half Words Quadruple 642
6.6.88 AVGRUBO: Average Rounded Unsigned of Byte Octuple 644
6.6.89 AVGRUHQ: Average Rounded Unsigned of Half Words Quadruple 646
6.6.90 AVGRUW: Average Rounded of Unsigned Words 648
6.6.91 AVGRUWP: Average Rounded Unsigned of Words Packed 650
6.6.92 AVGRW: Average Rounded of Words 652

Kalray – all rights reserved

6.6.93 AVGRWP: Average Rounded of Words Packed . . . 654
6.6.94 AVGUBO: Average Unsigned of Byte Octuple . . . 656
6.6.95 AVGUHQ: Average Unsigned of Half Words Quadruple . . . 658
6.6.96 AVGUW: Average of Unsigned Words . . . 660
6.6.97 AVGUWP: Average Unsigned of Words Packed . . . 662
6.6.98 AVGW: Average of Words . . . 664
6.6.99 AVGWP: Average of Words Packed . . . 666
6.6.100 CBSD: Count Bit Set of Double Word . . . 668
6.6.101 CBSW: Count Bit Set of Word . . . 669
6.6.102 CBSWP: Count Bit Sets of Word Pair . . . 670
6.6.103 CLRF: Clear Field . . . 671
6.6.104 CLSD: Count Leading Sign of Double Word . . . 672
6.6.105 CLSW: Count Leading Sign of Word . . . 673
6.6.106 CLSWP: Count Leading Signs of Word Pair . . . 674
6.6.107 CLZD: Count Leading Zero of Double Word . . . 675
6.6.108 CLZW: Count Leading Zero of Word . . . 676
6.6.109 CLZWP: Count Leading Zeroes of Word Pair . . . 677
6.6.110 CMOVEBO: Conditional Move Byte Octuple . . . 678
6.6.111 CMOVED: Conditional Move Double Word . . . 681
6.6.112 CMOVEHQ: Conditional Move Half Words Quadruple . . . 685
6.6.113 CMOVEWP: Conditional Move Words Packed . . . 687
6.6.114 COMPD: Compare Double Words . . . 689
6.6.115 COMPNBO: Compare Negate Byte Octuple . . . 693
6.6.116 COMPND: Compare Negate Double Words . . . 695
6.6.117 COMPNHQ: Compare Negate Half Words Quadruple . . . 697
6.6.118 COMPNW: Compare Negate Words . . . 699
6.6.119 COMPNWP: Compare Negate Word Pair . . . 701
6.6.120 COMPUWD: Compare Unsigned Word to Double Word . . . 703
6.6.121 COMPW: Compare Words . . . 705
6.6.122 COMPWD: Compare Word to Double Word . . . 707
6.6.123 CTZD: Count Trailing Zero of Double Word . . . 709
6.6.124 CTZW: Count Trailing Zero of Word . . . 710
6.6.125 CTZWP: Count Trailing Zeroes of Word Pair . . . 711
6.6.126 EORD: Bitwise Exclusive Or Between Double Words . . . 712
6.6.127 EORRBOD: Bitwise Exclusive OR Reduce Byte Octuple to Double Word . . . 717
6.6.128 EORRHQD: Bitwise Exclusive OR Reduce Half Word Quadruple to Double Word . . . 718
6.6.129 EORRWPD: Bitwise Exclusive OR Reduce Word Pair to Double Word . . . 719
6.6.130 EORW: Bitwise Exclusive Or Between Words . . . 720
6.6.131 EXTFS: Extract Bit Field and Sign Extend to Double Word . . . 723
6.6.132 EXTFZ: Extract Bit Field and Zero Extend to Double Word . . . 724
6.6.133 INSF: Insert Bit Field into Word . . . 725
6.6.134 IORD: Bitwise Inclusive Or Between Double Words . . . 726
6.6.135 IORND: Bitwise Inclusive Or with Not Operand Between Double Words . . . 731

Kalray – all rights reserved

6.6.136 IORNW: Bitwise Inclusive Or with Not Operand Between Double Words . . 736
6.6.137 IORRBOD: Bitwise OR Reduce Byte Octuple to Double Word 739
6.6.138 IORRHQD: Bitwise OR Reduce Half Word Quadruple to Double Word . . . 740
6.6.139 IORRWPD: Bitwise OR Reduce Word Pair to Double Word 741
6.6.140 IORW: Bitwise Inclusive Or Between Words 742
6.6.141 LANDD: Logical And of Double Words 745
6.6.142 LANDW: Logical And of Words 746
6.6.143 LIORD: Logical Or of Double Words 748
6.6.144 LIORW: Logical Or of Words 749
6.6.145 LNANDD: Logical Not And of Double Words 751
6.6.146 LNANDW: Logical Not And of Words 752
6.6.147 LNIORD: Logical Not Or of Double Words 754
6.6.148 LNIORW: Logical Not Or of Words 755
6.6.149 MAKE: Make Word from Immediate 757
6.6.150 MAXBO: Maximum of Byte Octuple 760
6.6.151 MAXD: Maximum of Double Words 762
6.6.152 MAXHQ: Maximum of Half Words Quadruple 767
6.6.153 MAXRBOD: Maximum Reduce Byte Octuple to Double Word 769
6.6.154 MAXRHQD: Maximum Reduce Half Word Quadruple to Double Word . . . 770
6.6.155 MAXRWPD: Maximum Reduce Word Pair to Double Word 771
6.6.156 MAXUBO: Maximum of Unsigned Byte Octuple 772
6.6.157 MAXUD: Maximum of Unsigned Double Words 774
6.6.158 MAXUHQ: Maximum of Unsigned Half Words Quadruple 779
6.6.159 MAXURBOD: Maximum Unsigned Reduce Byte Octuple to Double Word . 781
6.6.160 MAXURHQD: Maximum Unsigned Reduce Half Word Quadruple to Double Word 782
6.6.161 MAXURWPD: Maximum Unsigned Reduce Word Pair to Double Word . . . 783
6.6.162 MAXUW: Maximum of Unsigned Words 784
6.6.163 MAXUWP: Maximum of Unsigned Words Packed 787
6.6.164 MAXW: Maximum of Words 789
6.6.165 MAXWP: Maximum of Words Packed 792
6.6.166 MINBO: Minimum of Byte Octuple 794
6.6.167 MIND: Minimum of Double Words 796
6.6.168 MINHQ: Minimum of Half Words Quadruple 801
6.6.169 MINRBOD: Minimum Reduce Byte Octuple to Double Word 803
6.6.170 MINRHQD: Minimum Reduce Half Word Quadruple to Double Word 804
6.6.171 MINRWPD: Minimum Reduce Word Pair to Double Word 805
6.6.172 MINUBO: Minimum of Unsigned Byte Octuple 806
6.6.173 MINUD: Minimum of Unsigned Double Words 808
6.6.174 MINUHQ: Minimum of Unsigned Half Words Quadruple 813
6.6.175 MINURBOD: Minimum Unsigned Reduce Byte Octuple to Double Word . . 815
6.6.176 MINURHQD: Minimum Unsigned Reduce Half Word Quadruple to Double Word 816
6.6.177 MINURWPD: Minimum Unsigned Reduce Word Pair to Double Word 817

Kalray – all rights reserved

6.6.178 MINUW: Minimum of Unsigned Words . . . 818
6.6.179 MINUWP: Minimum of Unsigned Words Packed . . . 821
6.6.180 MINW: Minimum of Words . . . 823
6.6.181 MINWP: Minimum of Words Packed . . . 826
6.6.182 NANDD: Bitwise Not And Between Double Words . . . 828
6.6.183 NANDW: Bitwise Not And Between Words . . . 833
6.6.184 NEORD: Bitwise Not Exclusive Or Between Double Words . . . 836
6.6.185 NEORW: Bitwise Not Exclusive Or Between Words . . . 841
6.6.186 NIORD: Bitwise Not Inclusive Or Between Double Words . . . 844
6.6.187 NIORW: Bitwise Not Inclusive Or Between Words . . . 849
6.6.188 NOP: No Operation . . . 852
6.6.189 PCREL: PC-Relative . . . 853
6.6.190 ROLW: Rotate Left Word . . . 856
6.6.191 ROLWPS: Rotate Left Word Pair by Scalar . . . 858
6.6.192 RORW: Rotate Right Word . . . 860
6.6.193 RORWPS: Rotate Right Word Pair by Scalar . . . 862
6.6.194 SBFBO: Subtract Byte Octuple . . . 864
6.6.195 SBFCD: Subtract Double Word from Double Word with Carry . . . 866
6.6.196 SBFCD.I: Subtract Double Word from Double Word with Carry Initial . . . 868
6.6.197 SBFD: Subtract Double Word From Double Word . . . 870
6.6.198 SBFHQ: Subtract Half Words Quadruple . . . 875
6.6.199 SBFSBO: Subtract Saturated Byte Octuple . . . 877
6.6.200 SBFSD: Subtract Saturated Double Words . . . 879
6.6.201 SBFSHQ: Subtract Saturated Half Words Quadruple . . . 881
6.6.202 SBFSW: Subtract Saturated Words . . . 883
6.6.203 SBFSWP: Subtract Saturated Words Packed . . . 885
6.6.204 SBFUSBO: Subtract Unsigned Saturated Byte Octuple . . . 887
6.6.205 SBFUSD: Subtract Unsigned Saturated Double Words . . . 889
6.6.206 SBFUSHQ: Subtract Unsigned Saturated Half Words Quadruple . . . 891
6.6.207 SBFUSW: Subtract Unsigned Saturated Words . . . 893
6.6.208 SBFUSWP: Subtract Unsigned Saturated Words Packed . . . 895
6.6.209 SBFUWD: Subtract Unsigned Word from Double Word . . . 897
6.6.210 SBFW: Subtract Word from Word . . . 899
6.6.211 SBFWD: Subtract Word from Double Word . . . 902
6.6.212 SBFWP: Subtract Words Packed . . . 904
6.6.213 SBFX16BO: Subtract Integer to Integer Times 16 Byte Octuple . . . 906
6.6.214 SBFX16D: Subtract Double Word Times 16 from Double Word . . . 908
6.6.215 SBFX16HQ: Subtract Integer to Integer Times 16 Half Words Quadruple . . . 910
6.6.216 SBFX16UWD: Subtract Unsigned Word Times 16 from Double Word . . . 912
6.6.217 SBFX16W: Subtract Word Times 16 from Word . . . 914
6.6.218 SBFX16WD: Subtract Word Times 16 from Double Word . . . 916
6.6.219 SBFX16WP: Subtract Integer to Integer Times 16 Words Packed . . . 918
6.6.220 SBFX2BO: Subtract Integer to Integer Times 2 Byte Octuple . . . 920
6.6.221 SBFX2D: Subtract Double Word Times 2 from Double Word . . . 922

Kalray – all rights reserved

6.6.222 SBFX2HQ: Subtract Integer to Integer Times 2 Half Words Quadruple 924
6.6.223 SBFX2UWD: Subtract Unsigned Word Times 2 from Double Word 926
6.6.224 SBFX2W: Subtract Word Times 2 from Word 928
6.6.225 SBFX2WD: Subtract Word Times 2 from Double Word 930
6.6.226 SBFX2WP: Subtract Integer to Integer Times 2 Words Packed 932
6.6.227 SBFX32D: Subtract Double Word Times 32 from Double Word 934
6.6.228 SBFX32UWD: Subtract Unsigned Word Times 32 from Double Word 936
6.6.229 SBFX32W: Subtract Word Times 32 from Word 938
6.6.230 SBFX32WD: Subtract Word Times 32 from Double Word 940
6.6.231 SBFX4BO: Subtract Integer to Integer Times 4 Byte Octuple 942
6.6.232 SBFX4D: Subtract Double Word Times 4 from Double Word 944
6.6.233 SBFX4HQ: Subtract Integer to Integer Times 4 Half Words Quadruple 946
6.6.234 SBFX4UWD: Subtract Unsigned Word Times 4 from Double Word 948
6.6.235 SBFX4W: Subtract Word Times 4 from Word 950
6.6.236 SBFX4WD: Subtract Word Times 4 from Double Word 952
6.6.237 SBFX4WP: Subtract Integer to Integer Times 4 Words Packed 954
6.6.238 SBFX64D: Subtract Double Word Times 64 from Double Word 956
6.6.239 SBFX64UWD: Subtract Unsigned Word Times 64 from Double Word 958
6.6.240 SBFX64W: Subtract Word Times 64 from Word 960
6.6.241 SBFX64WD: Subtract Word Times 64 from Double Word 962
6.6.242 SBFX8BO: Subtract Integer to Integer Times 8 Byte Octuple 964
6.6.243 SBFX8D: Subtract Double Word Times 8 from Double Word 966
6.6.244 SBFX8HQ: Subtract Integer to Integer Times 8 Half Words Quadruple 968
6.6.245 SBFX8UWD: Subtract Unsigned Word Times 8 from Double Word 970
6.6.246 SBFX8W: Subtract Word Times 8 from Word 972
6.6.247 SBFX8WD: Subtract Word Times 8 from Double Word 974
6.6.248 SBFX8WP: Subtract Integer to Integer Times 8 Words Packed 976
6.6.249 SBMM8D: Swapped Bit Matrix Multiplication 8×8 978
6.6.250 SBMMT8D: Swapped Bit Matrix Multiplication Transposed 8×8 983
6.6.251 SLLBOS: Shift Left Logical Byte Octuple by Scalar 988
6.6.252 SLLD: Shift Left Logical Double Word 990
6.6.253 SLLHQS: Shift Left Logical Half Word Quadruple by Scalar 992
6.6.254 SLLW: Shift Left Logical Word 994
6.6.255 SLLWPS: Shift Left Logical Word Pair by Scalar 996
6.6.256 SLSBOS: Shift Left Saturated Byte Octuple by Scalar 998
6.6.257 SLSD: Shift Left Saturated Double Word 1000
6.6.258 SLSHQS: Shift Left Saturated Half Word Quadruple by Scalar 1002
6.6.259 SLSW: Shift Left Saturated Word 1004
6.6.260 SLSWPS: Shift Left Saturated Word Pair by Scalar 1006
6.6.261 SLUSBOS: Shift Left Unsigned Saturated Byte Octuple by Scalar 1008
6.6.262 SLUSD: Shift Left Unsigned Saturated Double Word 1010
6.6.263 SLUSHQS: Shift Left Unsigned Saturated Half Word Quadruple by Scalar . . 1012
6.6.264 SLUSW: Shift Left Unsigned Saturated Word 1014
6.6.265 SLUSWPS: Shift Left Unsigned Saturated Word Pair by Scalar 1016

Kalray – all rights reserved

6.6.266 SRABOS: Shift Right Arithmetic Byte Octuple by Scalar 1018
6.6.267 SRAD: Shift Right Arithmetic Double Word 1020
6.6.268 SRAHQS: Shift Right Arithmetic Half Word Quadruple by Scalar 1022
6.6.269 SRAW: Shift Right Arithmetic Word 1024
6.6.270 SRAWPS: Shift Right Arithmetic Word Pair by Scalar 1026
6.6.271 SRLBOS: Shift Right Logical Byte Octuple by Scalar 1028
6.6.272 SRLD: Shift Right Logical Double Word 1030
6.6.273 SRLHQS: Shift Right Logical Half Word Quadruple by Scalar 1032
6.6.274 SRLW: Shift Right Logical Word 1034
6.6.275 SRLWPS: Shift Right Logical Word Pair by Scalar 1036
6.6.276 SRSBOS: Shift Right Symmetric Byte Octuple by Scalar 1038
6.6.277 SRSD: Shift Right Symmetric Double Word 1040
6.6.278 SRSHQS: Shift Right Symmetric Half Word Quadruple by Scalar 1042
6.6.279 SRSW: Shift Right Symmetric Word 1044
6.6.280 SRSWPS: Shift Right Symmetric Word Pair by Scalar 1046
6.6.281 STSUD: Subtract, Test and Shift Unsigned Double Words 1048
6.6.282 STSUHQ: Subtract, Test and Shift Unsigned Half Word Quadruple 1050
6.6.283 STSUW: Subtract, Test and Shift Unsigned Word 1052
6.6.284 STSUWP: Subtract, Test and Shift Unsigned Word Pair 1054
6.6.285 SXLBHQ: Sign Extend Least Significant Bytes to Half Word Quadruple . . . 1056
6.6.286 SXLHWP: Sign Extend Least Significant Half Words to Word Pair 1057
6.6.287 SXMBHQ: Sign Extend Most Significant Bytes to Half Word Quadruple . . . 1058
6.6.288 SXMHWP: Sign Extend Most Significant Half Words to Word Pair 1059
6.6.289 XMOVETD: Move To Extension Double Word 1060
6.6.290 XMOVETQ: Move To Extension Quadruple Word 1064
6.6.291 XRECVO: Receive Extension Octuple Word from Channel 1066
6.6.292 XSENDO: Send Extension Octuple Word to Channel 1067
6.6.293 XSENDRECVO: Send and Receive Octuple Word to/from Channels 1068
6.6.294 XSPLATDO: Move To Extension Double Word 1069
6.6.295 ZXLBHQ: Zero Extend Least Significant Bytes to Half Word Quadruple . . . 1072
6.6.296 ZXLHWP: Zero Extend Least Significant Half Words to Word Pair 1073
6.6.297 ZXMBHQ: Zero Extend Most Significant Bytes to Half Word Quadruple . . . 1074
6.6.298 ZXMHWP: Zero Extend Most Significant Half Words to Word Pair 1075
6.7 MAU Instructions 1076
6.7.1 CMULDT: Carryless Multiply Double Words to Tetra Word 1076
6.7.2 CMULGHXDT: Carryless Multiply Galois High and Xor Double Words to Tetra Word 1077
6.7.3 CMULGLXDT: Carryless Multiply Galois Low and Xor Double Words to Tetra Word 1078
6.7.4 CMULGMXDT: Carryless Multiply Galois Middle and Xor Double Words to Tetra Word 1079
6.7.5 CMULXDT: Carryless Multiply and Xor Double Words to Tetra Word 1080
6.7.6 COPYQ: Copy Quadruple Word 1081

Kalray – all rights reserved

6.7.7 CRCBELLW: Cyclic Redundancy Check Big-Endian Least Significant with Least Significant Words . . . 1082
6.7.8 CRCBELMW: Cyclic Redundancy Check Big-Endian Least Significant with Most Significant Words . . . 1084
6.7.9 CRCLELLW: Cyclic Redundancy Check Little-Endian Least Significant with Least Significant Words . . . 1086
6.7.10 CRCLELMW: Cyclic Redundancy Check Little-Endian Least Significant with Most Significant Words . . . 1088
6.7.11 DOT2SUWD: Dot product of 2x Signed by Unsigned Word Vectors into Double Word . . . 1090
6.7.12 DOT2SUWDP: Dot products of 2x Signed by Unsigned Word Vectors into Double Word Pair . . . 1091
6.7.13 DOT2UWD: Dot product of 2x Unsigned Word Vectors into Double Word . . . 1092
6.7.14 DOT2UWDP: Dot products of 2x Unsigned Word Vectors into Double Word Pair . . . 1093
6.7.15 DOT2W: Dot product of 2x Word Vectors into Word . . . 1094
6.7.16 DOT2WD: Dot product of 2x Word Vectors into Double Word . . . 1095
6.7.17 DOT2WDP: Dot products of 2x Word Vectors into Double Word Pair . . . 1096
6.7.18 DOT2WZP: Dot products of 2x Word Vectors into Word Zero Extended Pair . 1097
6.7.19 MADDD: Multiply-Add Double Word . . . 1098
6.7.20 MADDDT: Multiply-Add Double to Tetra Word . . . 1100
6.7.21 MADDHQ: Multiply-Add Half Word Quadruple . . . 1101
6.7.22 MADDHWQ: Multiply-Add Half Word to Word Quadruple . . . 1103
6.7.23 MADDMWQ: Multiply-Add Most Significant Word Quadruple . . . 1104
6.7.24 MADDSUDT: Multiply-Add Signed by Unsigned Double to Tetra Word . . . 1105
6.7.25 MADDSUHWQ: Multiply-Add Signed by Unsigned Half Word to Word Quadruple . . . 1106
6.7.26 MADDSUMWQ: Multiply-Add Signed by Unsigned Most Significant Word Quadruple . . . 1107
6.7.27 MADDSUWD: Multiply-Add Signed by Unsigned Words to Double Word . . 1108
6.7.28 MADDSUWDP: Multiply-Add Signed by Unsigned Words to Double Word Pair . . . 1110
6.7.29 MADDUDT: Multiply-Add Unsigned Double to Tetra Word . . . 1112
6.7.30 MADDUHWQ: Multiply-Add Unsigned Half Word to Word Quadruple . . . 1113
6.7.31 MADDUMWQ: Multiply-Add Unsigned Most Significant Word Quadruple . 1114
6.7.32 MADDUWD: Multiply-Add Unsigned Words to Double Word . . . 1115
6.7.33 MADDUWDP: Multiply-Add Unsigned Words to Double Word Pair . . . 1117
6.7.34 MADDUZDT: Multiply-Add Unsigned Double to Tetra Word . . . 1119
6.7.35 MADDW: Multiply-Add Words . . . 1120
6.7.36 MADDWD: Multiply-Add Words to Double Word . . . 1122
6.7.37 MADDWDP: Multiply-Add Words to Double Word Pair . . . 1124
6.7.38 MADDWP: Multiply-Add Word Pair . . . 1126
6.7.39 MADDWQ: Multiply-Add Word Quadruple . . . 1128
6.7.40 MM212W: Matrix Multiply 2x1x2 Word Quadruple . . . 1129

Kalray – all rights reserved

6.7.41 MMA212W: Matrix Multiply-Add 2x1x2 Word 1131
6.7.42 MMS212W: Matrix Multiply-Subtract 2x1x2 Word 1133
6.7.43 MSBFD: Multiply-Subtract Double Word 1135
6.7.44 MSBFDT: Multiply-Subtract Double to Tetra Word 1137
6.7.45 MSBFHQ: Multiply-Subtract Half Word Quadruple 1138
6.7.46 MSBFHWQ: Multiply-Subtract Half Word from Word Quadruple 1140
6.7.47 MSBFMWQ: Multiply-Subtract Most Significant Word Quadruple 1141
6.7.48 MSBFSUDT: Multiply-Subtract Signed by Unsigned Double to Tetra Word . 1142
6.7.49 MSBFSUHWQ: Multiply-Subtract Signed by Unsigned Half Word from Word Quadruple 1143
6.7.50 MSBFSUMWQ: Multiply-Subtract Signed by Unsigned Most Significant Word Quadruple 1144
6.7.51 MSBFSUWD: Multiply-Subtract Signed by Unsigned Words to Double Word 1145
6.7.52 MSBFSUWDP: Multiply-Subtract Signed by Unsigned Words to Double Word Pair 1147
6.7.53 MSBFUDT: Multiply-Subtract Unsigned Double to Tetra Word 1149
6.7.54 MSBFUHWQ: Multiply-Subtract Unsigned Half Word from Word Quadruple 1150
6.7.55 MSBFUMWQ: Multiply-Subtract Unsigned Most Significant Word Quadruple 1151
6.7.56 MSBFUWD: Multiply-Subtract Unsigned Words to Double Word 1152
6.7.57 MSBFUWDP: Multiply-Subtract Unsigned Words to Double Word Pair . . . 1154
6.7.58 MSBFUZDT: Multiply-Subtract Unsigned Double to Quadruple Word 1156
6.7.59 MSBFW: Multiply-Subtract Words 1157
6.7.60 MSBFWD: Multiply-Subtract Words to Double Word 1159
6.7.61 MSBFWDP: Multiply-Subtract Words to Double Word Pair 1161
6.7.62 MSBFWP: Multiply-Subtract Word Pair 1163
6.7.63 MSBFWQ: Multiply-Subtract Word Quadruple 1165
6.7.64 MULD: Multiply Double Word 1166
6.7.65 MULDT: Multiply Double Word to Tetra Word 1168
6.7.66 MULHQ: Multiply Half Word Quadruple 1169
6.7.67 MULHWQ: Multiply Half Word to Word Quadruple 1171
6.7.68 MULMWQ: Multiply Most Significant Word Quadruple 1172
6.7.69 MULSUDT: Multiply Signed by Unsigned Double Words to Tetra Word . . . 1173
6.7.70 MULSUHWQ: Multiply Signed by Unsigned Half Word to Word Quadruple . 1174
6.7.71 MULSUMWQ: Multiply Signed by Unsigned Most Significant Word Quadruple 1175
6.7.72 MULSUWD: Multiply Signed by Unsigned Words to Double Word 1176
6.7.73 MULSUWDP: Multiply Signed by Unsigned Words to Double Word Pair . . 1178
6.7.74 MULUDT: Multiply Unsigned Double Words to Tetra Word 1180
6.7.75 MULUHWQ: Multiply Unsigned Half Word to Word Quadruple 1181
6.7.76 MULUMWQ: Multiply Unsigned Most Significant Word Quadruple 1182
6.7.77 MULUWD: Multiply Unsigned Words to Double Word 1183
6.7.78 MULUWDP: Multiply Unsigned Words to Double Word Pair 1185
6.7.79 MULW: Multiply Words 1187
6.7.80 MULWD: Multiply Words to Double Word 1189

Kalray – all rights reserved

6.7.81 MULWDP: Multiply Words to Double Word Pair 1191
6.7.82 MULWP: Multiply Word Pair 1193
6.7.83 MULWQ: Multiply Word Quadruple 1195
6.8 FPU Instructions 1196
6.8.1 FABSD: Floating-Point Absolute Value Double Word 1196
6.8.2 FABSHQ: Floating-Point Absolute Value Half Word Quadruple 1197
6.8.3 FABSW: Floating-Point Absolute Value Word 1198
6.8.4 FABSWP: Floating-Point Absolute Value Word Pair 1199
6.8.5 FADDD: Floating-Point Add Double Word 1200
6.8.6 FADDDC.C: Floating-Point Add Double Word Complex Conjugate 1201
6.8.7 FADDDP: Floating-Point Add Double Word Pair 1202
6.8.8 FADDHO: Floating-Point Add Half Word Octuple 1203
6.8.9 FADDHQ: Floating-Point Add Half Word Quadruple 1204
6.8.10 FADDW: Floating-Point Add Words 1205
6.8.11 FADDWC.C: Floating-Point Add Word Complex Conjugate 1206
6.8.12 FADDWCP.C: Floating-Point Add Word Complex Pair Conjugate 1207
6.8.13 FADDWP: Floating-Point Add Word Pair 1208
6.8.14 FADDWQ: Floating-Point Add Word Quadruple 1209
6.8.15 FCDIVD: Floating-Point Cases for Division Double Word 1210
6.8.16 FCDIVW: Floating-Point Cases for Word Division 1211
6.8.17 FCDIVWP: Floating-Point Cases for Division Word Pair 1212
6.8.18 FCOMPD: Floating-Point Compare Double 1213
6.8.19 FCOMPND: Floating-Point Compare Negate Double 1215
6.8.20 FCOMPNHQ: Floating-Point Compare Negate Half Word Quadruple 1217
6.8.21 FCOMPNW: Floating-Point Compare Negate Single 1219
6.8.22 FCOMPNWP: Floating-Point Compare Negate Word Pair 1221
6.8.23 FCOMPW: Floating-Point Compare Single 1223
6.8.24 FDOT2W: Floating-Point Dot Product x2 Word 1225
6.8.25 FDOT2WD: Floating-Point Dot Product x2 Word to Double Word 1226
6.8.26 FDOT2WDP: Floating-Point Dot Product x2 Word to Double Word Pair . . . 1227
6.8.27 FDOT2WZP: Floating-Point Dot Product x2 Word Zero Extended Pair 1228
6.8.28 FFDMASW: Floating-Point Fused Dual Multiply Add Subtract Word 1229
6.8.29 FFDMASWP: Floating-Point Fused Dual Multiply Add Subtract Word pair . 1230
6.8.30 FFDMASWQ: Floating-Point Fused Dual Multiply Add Subtract Word Quadruple 1231
6.8.31 FFDMAW: Floating-Point Fused Dual Multiply Add Word 1232
6.8.32 FFDMAWP: Floating-Point Fused Dual Multiply Add Word Pair 1233
6.8.33 FFDMAWQ: Floating-Point Fused Dual Multiply Add Word Quadruple . . . 1234
6.8.34 FFDMDAW: Floating-Point Fused Dual Multiply Dual Add Word 1235
6.8.35 FFDMDAWP: Floating-Point Fused Dual Multiply Dual Add Word Pair . . . 1236
6.8.36 FFDMDAWQ: Floating-Point Fused Dual Multiply Dual Add Word Quadruple 1237
6.8.37 FFDMDSW: Floating-Point Fused Dual Multiply Dual Subtract Word 1238
6.8.38 FFDMDSWP: Floating-Point Fused Dual Multiply Dual Subtract Word Pair . 1239

Kalray – all rights reserved

6.8.39 FFDMDSWQ: Floating-Point Fused Dual Multiply Dual Subtract Word Quadruple . . . 1240
6.8.40 FFDMSAW: Floating-Point Fused Dual Multiply Subtract Add Word 1241
6.8.41 FFDMSAWP: Floating-Point Fused Dual Multiply Subtract Add Word Pair . 1242
6.8.42 FFDMSAWQ: Floating-Point Fused Dual Multiply Subtract Add Word Quadruple . . . 1243
6.8.43 FFDMSW: Floating-Point Fused Dual Multiply Subtract Word 1244
6.8.44 FFDMSWP: Floating-Point Fused Dual Multiply Subtract Word Pair 1245
6.8.45 FFDMSWQ: Floating-Point Fused Dual Multiply Subtract Word Quadruple . 1246
6.8.46 FFMAD: Floating-Point Fused Multiply-Add Double 1247
6.8.47 FFMAHO: Floating-Point Fused Multiply-Add Half Word Octuple 1248
6.8.48 FFMAHQ: Floating-Point Fused Multiply-Add Half Word Quadruple 1249
6.8.49 FFMAHW: Floating-Point Fused Multiply-Add Half Words to Word 1250
6.8.50 FFMAHWQ: Floating-Point Fused Multiply-Add Half Words to Word Quadruple . . . 1251
6.8.51 FFMAW: Floating-Point Fused Multiply-Add Words 1252
6.8.52 FFMAWC: Floating-Point Multiply Add Word Complex 1253
6.8.53 FFMAWCP: Floating-Point Multiply Add Word Complex Pair 1255
6.8.54 FFMAWD: Floating-Point Fused Multiply-Add Words to Double Word . . . 1257
6.8.55 FFMAWDP: Floating-Point Fused Multiply-Add Words to Double Word Pair 1258
6.8.56 FFMAWP: Floating-Point Fused Multiply-Add Word Pair 1259
6.8.57 FFMAWQ: Floating-Point Fused Multiply-Add Word Quadruple 1260
6.8.58 FFMSD: Floating-Point Fused Multiply-Subtract Double 1261
6.8.59 FFMSHO: Floating-Point Fused Multiply-Subtract Half Word Octuple 1262
6.8.60 FFMSHQ: Floating-Point Fused Multiply-Subtract Half Word Quadruple . . 1263
6.8.61 FFMSHW: Floating-Point Fused Multiply-Subtract Half Words to Word . . . 1264
6.8.62 FFMSHWQ: Floating-Point Fused Multiply-Subtract Half Words to Word Quadruple . . . 1265
6.8.63 FFMSW: Floating-Point Fused Multiply-Subtract Words 1266
6.8.64 FFMSWC: Floating-Point Multiply Subtract Word Complex 1267
6.8.65 FFMSWCP: Floating-Point Multiply Subtract Word Complex Pair 1269
6.8.66 FFMSWD: Floating-Point Fused Multiply-Subtract Words to Double Word . 1271
6.8.67 FFMSWDP: Floating-Point Fused Multiply-Subtract Words to Double Word Pair . . . 1272
6.8.68 FFMSWP: Floating-Point Fused Multiply-Subtract Word Pair 1273
6.8.69 FFMSWQ: Floating-Point Fused Multiply-Subtract Word Quadruple 1274
6.8.70 FIXEDD: Floating Point Conversion to Fixed-Point Double 1275
6.8.71 FIXEDUD: Floating Point Conversion to Unsigned Fixed-Point Double . . . 1276
6.8.72 FIXEDUW: Floating Point Conversion to Unsigned Fixed-Point 1277
6.8.73 FIXEDUWP: Floating Point Conversion to Unsigned Fixed-Point Word Pair . 1278
6.8.74 FIXEDW: Floating Point Conversion to Fixed-Point 1279
6.8.75 FIXEDWP: Floating Point Conversion to Fixed-Point Word Pair 1280
6.8.76 FLOATD: Floating Point Conversion from Fixed-Point Double 1281
6.8.77 FLOATUD: Floating Point Conversion from Unsigned Fixed-Point Double . . 1282

Kalray – all rights reserved

6.8.78 FLOATUW: Floating Point Conversion from Unsigned Fixed-Point 1283
6.8.79 FLOATUWP: Floating Point Conversion from Unsigned Fixed-Point Word Pair 1284
6.8.80 FLOATW: Floating Point Conversion from Fixed-Point 1285
6.8.81 FLOATWP: Floating Point Conversion from Fixed-Point Word Pair 1286
6.8.82 FMAXD: Floating-Point Maximum Double Word 1287
6.8.83 FMAXHQ: Floating-Point Maximum Half Word Quadruple 1288
6.8.84 FMAXW: Floating-Point Maximum Word 1289
6.8.85 FMAXWP: Floating-Point Maximum Word Pair 1290
6.8.86 FMIND: Floating-Point Minimum Double Word 1291
6.8.87 FMINHQ: Floating-Point Minimum Half Word Quadruple 1292
6.8.88 FMINW: Floating-Point Minimum Word 1293
6.8.89 FMINWP: Floating-Point Minimum Word Pair 1294
6.8.90 FMM212W: Floating-Point Matrix Multiply 2x1x2 Word 1295
6.8.91 FMM222W: Floating-Point Matrix Multiply 2x2x2 Word 1296
6.8.92 FMMA212W: Floating-Point Matrix Multiply Add 2x1x2 Word 1297
6.8.93 FMMA222W: Floating-Point Matrix Multiply Add 2x2x2 Word 1298
6.8.94 FMMS212W: Floating-Point Matrix Multiply Subtract 2x1x2 Word 1300
6.8.95 FMMS222W: Floating-Point Matrix Multiply Subtract 2x2x2 Word 1301
6.8.96 FMULD: Floating-Point Multiply Double Word 1303
6.8.97 FMULHO: Floating-Point Multiply Half Word Octuple 1304
6.8.98 FMULHQ: Floating-Point Multiply Half Word Quadruple 1305
6.8.99 FMULHW: Floating-Point Multiply Half Words to Word 1306
6.8.100 FMULHWQ: Floating-Point Multiply Half Word to Word Quadruple 1307
6.8.101 FMULW: Floating-Point Multiply Words 1308
6.8.102 FMULWC: Floating-Point Multiply Word Complex 1309
6.8.103 FMULWCP: Floating-Point Multiply Word Complex Pair 1310
6.8.104 FMULWD: Floating-Point Multiply Words to Double Word 1312
6.8.105 FMULWDP: Floating-Point Multiply Word to Double Word Pair 1313
6.8.106 FMULWP: Floating-Point Multiply Word Pair 1314
6.8.107 FMULWQ: Floating-Point Multiply Word Quadruple 1315
6.8.108 FNARROWDW: Floating Point Narrow Double Word to Word 1316
6.8.109 FNARROWDWP: Floating Point Narrow Double to Word Pair 1317
6.8.110 FNARROWWH: Floating Point Narrow Word to Half Word 1318
6.8.111 FNARROWWHQ: Floating Point Narrow Word to Half Word Quadruple . . . 1319
6.8.112 FNEGD: Floating-Point Negate Double Word 1320
6.8.113 FNEGHQ: Floating-Point Negate Half Word Quadruple 1321
6.8.114 FNEGW: Floating-Point Negate Word 1322
6.8.115 FNEGWP: Floating-Point Negate Word Pair 1323
6.8.116 FRECW: Floating-Point Reciprocal of Word 1324
6.8.117 FRSRW: Floating-Point Reciprocal of Square Root Word 1325
6.8.118 FSBFD: Floating-Point Subtract From Double Word 1326
6.8.119 FSBFDC.C: Floating-Point Subtract From Double Word Complex Conjugate . 1327
6.8.120 FSBFDP: Floating-Point Subtract Word to Double Word Pair 1328
6.8.121 FSBFHO: Floating-Point Subtract From Half Word Octuple 1329

Kalray – all rights reserved

6.8.122 FSBFHQ: Floating-Point Subtract From Half Word Quadruple 1330
6.8.123 FSBFW: Floating-Point Subtract From Words 1331
6.8.124 FSBFWC.C: Floating-Point Subtract From Word Complex Conjugate 1332
6.8.125 FSBFWCP.C: Floating-Point Subtract From Word Complex Pair Conjugate . 1333
6.8.126 FSBFWP: Floating-Point Subtract From Word Pair 1334
6.8.127 FSBFWQ: Floating-Point Subtract From Word Quadruple 1335
6.8.128 FSDIVD: Floating-Point Seed for Division Double Word 1336
6.8.129 FSDIVW: Floating-Point Seed for Word Division 1337
6.8.130 FSDIVWP: Floating-Point Seed for Division Word Pair 1338
6.8.131 FSRECD: Floating-Point Seed for Reciprocal Double Word 1339
6.8.132 FSRECW: Floating-Point Seed for Reciprocal Word 1340
6.8.133 FSRECWP: Floating-Point Seed for Reciprocal on Word Pair 1341
6.8.134 FSRSRD: Floating-Point Seed for Reciprocal Square Root Double Word . . . 1342
6.8.135 FSRSRW: Floating-Point Seed for Reciprocal Square Root 1343
6.8.136 FSRSRWP: Floating-Point Seed for Reciprocal Square Root on Word Pair . . 1344
6.8.137 FWIDENLHW: Floating Point Widen Least Significant Half Word to Word . 1345
6.8.138 FWIDENLHWP: Floating Point Widen Least Significant Half Words to Word Pair 1346
6.8.139 FWIDENLWD: Floating Point Widen Least Significant Word to Double . . . 1347
6.8.140 FWIDENMHW: Floating Point Widen Most Significant Half Word to Word . 1348
6.8.141 FWIDENMHWP: Floating Point Widen Most Significant Half Words to Word Pair 1349
6.8.142 FWIDENMWD: Floating Point Widen Most Significant Word to Double . . . 1350
6.9 EXT Instructions 1351
6.9.1 XANDNO: Bitwise And with Not Operand Between Octuple Words 1351
6.9.2 XANDO: Bitwise And Between Octuple Words 1352
6.9.3 XCLAMPWO: Clamp Word Octuple 1353
6.9.4 XCOPYO: Copy Extension Octuple Word 1354
6.9.5 XCOPYV: Copy Word Vector 1355
6.9.6 XCOPYX: Copy Word Hexadecuple 1356
6.9.7 XEORO: Bitwise Exclusive Or Between Octuple Words 1358
6.9.8 XFFMA44HW: Floating-Point Fused Multiply-Add 4x4 Half Words to Words 1359
6.9.9 XFMAXHX: Floating-Point Maximum Half Word Hexadecuple 1361
6.9.10 XFMINHX: Floating-Point Minimum Half Word Hexadecuple 1362
6.9.11 XFMMA484HW: Floating-Point Matrix Multiply-Add 4x8x4 Half Words to Words 1363
6.9.12 XFNARROW44WH: Floating Point Narrow 4x4 Matrix Word to Half Word . 1366
6.9.13 XFSCALEWO: Floating Point Scale Word Vector 1368
6.9.14 XIORNO: Bitwise Or with Not Operand Between Octuple Words 1369
6.9.15 XIORO: Bitwise Or Between Octuple Words 1370
6.9.16 XMADD44BW0: Multiply Add 4x4 Matrix Byte to Word 0 1371
6.9.17 XMADD44BW1: Multiply Add 4x4 Matrix Byte to Word 1 1372
6.9.18 XMADDIFWO: Multiply Add Integer Floating Point Word Octuple 1373

Kalray – all rights reserved

6.9.19 XMADDSU44BW0: Multiply-Add Signed by Unsigned 4x4 Matrix Byte to Word 0 . . . 1374
6.9.20 XMADDSU44BW1: Multiply-Add Signed by Unsigned 4x4 Matrix Byte to Word 1 . . . 1375
6.9.21 XMADDU44BW0: Multiply-Add Unsigned 4x4 Matrix Byte to Word 0 . . . 1376
6.9.22 XMADDU44BW1: Multiply-Add Unsigned 4x4 Matrix Byte to Word 1 . . . 1377
6.9.23 XMMA4164BW: Matrix Multiply-Add 4x16x4 Bytes to Words . . . 1378
6.9.24 XMMA484BW: Extension Matrix Multiply-Add 4x8x4 Bytes to Words . . . 1380
6.9.25 XMMASU4164BW: Matrix Multiply-Add 4x16x4 Signed by Unsigned Bytes to Words . . . 1382
6.9.26 XMMASU484BW: Matrix Multiply-Add 4x8x4 Signed by Unsigned Bytes to Words . . . 1384
6.9.27 XMMAU4164BW: Matrix Multiply-Add 4x16x4 Unsigned by Unsigned Bytes to Words . . . 1386
6.9.28 XMMAU484BW: Matrix Multiply-Add 4x8x4 Unsigned by Unsigned Bytes to Words . . . 1388
6.9.29 XMMAUS4164BW: Matrix Multiply-Add 4x16x4 Unsigned by Signed Bytes to Words . . . 1390
6.9.30 XMMAUS484BW: Matrix Multiply-Add 4x8x4 Unsigned by Signed Bytes to Words . . . 1392
6.9.31 XMOVEFD: Move From Extension Double Word . . . 1394
6.9.32 XMOVEFO: Move From Extension Octuple Word . . . 1395
6.9.33 XMOVEFQ: Move From Extension Quadruple Word . . . 1396
6.9.34 XMSBFIFWO: Multiply Subtract Integer Floating Point Word Octuple . . . 1397
6.9.35 XNANDO: Bitwise Not And Between Octuple Words . . . 1398
6.9.36 XNEORO: Bitwise Not Exclusive Or Between Octuple Words . . . 1399
6.9.37 XNIORO: Bitwise Not Or Between Octuple Words . . . 1400
6.9.38 XSBMM8DQ: Swapped Bit Matrix Multiplication 8×8 Double Quadruple . . . 1401
6.9.39 XSBMMT8DQ: Swapped Bit Matrix Multiplication 8×8 Double Quadruple . . . 1402
6.9.40 XSPLATOV: Splat Word Octuple to Vector . . . 1403
6.9.41 XSPLATOX: Splat Word Octuple to Hexadecuple . . . 1404
6.9.42 XSX48BW: Sign Extend 4x8 Matrix Byte to Word . . . 1406
6.9.43 XTRUNC48WB: Truncate 4x8 Matrix Word to Byte . . . 1407
6.9.44 XZX48BW: Zero Extend 4x8 Matrix Byte to Word . . . 1408
6.10 Instruction operands . . . 1409
6.10.1 Immediate classes . . . 1409
6.10.2 Operand Modifiers . . . 1411
6.10.3 Register Classes . . . 1433
6.11 Synthetic Instructions . . . 1460
6.11.1 ABSBO: Absolute Value of Byte Octuple . . . 1461
6.11.2 ABSD: Absolute Value of Double Word . . . 1462
6.11.3 ABSHQ: Absolute Value of Half Word Quadruple . . . 1463
6.11.4 ABSSBO: Absolute Value Saturated of Byte Octuple . . . 1464
6.11.5 ABSSD: Absolute Value Saturated of Double Word . . . 1465

Kalray – all rights reserved

6.11.6 ABSSHQ: Absolute Value Saturated of Half Quadruple 1466
6.11.7 ABSSW: Absolute Value Saturated of Word 1467
6.11.8 ABSSWP: Absolute Value Saturated of Word Pair 1468
6.11.9 ABSW: Absolute Value of Word 1469
6.11.10 ABSWP: Absolute Value of Word Pair 1470
6.11.11 COPYD: Copy Double Word 1471
6.11.12 COPYW: Copy Word 1472
6.11.13 LNORD: Alias of LNIORD 1473
6.11.14 LNORW: Alias of LNIORW 1474
6.11.15 LORD: Alias of LIORD 1475
6.11.16 LORW: Alias of LIORW 1476
6.11.17 NEGBO: Negate Word Byte Octuple 1477
6.11.18 NEGD: Negate Double Word 1478
6.11.19 NEGHQ: Negate Word Half Word Quadruple 1479
6.11.20 NEGSBO: Negate Word Saturated Byte Octuple 1480
6.11.21 NEGSD: Negate Saturated Double Word 1481
6.11.22 NEGSHQ: Negate Word Half Word Quadruple 1482
6.11.23 NEGSW: Negate Saturated Word 1483
6.11.24 NEGSWP: Negate Saturated Word Pair 1484
6.11.25 NEGW: Negate Word 1485
6.11.26 NEGWP: Negate Word Pair 1486
6.11.27 NORD: Legacy form of NIORD 1487
6.11.28 NORW: Legacy form of NIORW 1489
6.11.29 NOTD: Complement Double Word 1490
6.11.30 NOTW: Complement Word 1491
6.11.31 NXORD: Alias of NEORD 1492
6.11.32 NXORW: Alias of NEORW 1494
6.11.33 ORD: Legacy form of IORD 1495
6.11.34 ORND: Legacy form of IORND 1497
6.11.35 ORNW: Legacy form of IORNW 1499
6.11.36 ORRBOD: Legacy form of IORRBOD 1500
6.11.37 ORRHQD: Legacy form of IORRHQD 1501
6.11.38 ORRWPD: Legacy form of IORRWPD 1502
6.11.39 ORW: Legacy form of IORW 1503
6.11.40 SBMM8: Swapped Bit Matrix Multiplication 8×8 1504
6.11.41 SBMMT8: Swapped Bit Matrix Multiplication Transposed 8×8 1506
6.11.42 SXBD: Sign Extend Byte to Double Word 1508
6.11.43 SXHD: Sign Extend Half Word to Double Word 1509
6.11.44 SXWD: Sign Extend Word to Double Word 1510
6.11.45 XORD: Alias of EORD 1511
6.11.46 XORRBOD: Alias of EORRBOD 1513
6.11.47 XORRHQD: Alias of EORRHQD 1514
6.11.48 XORRWPD: Alias of EORRWPD 1515
6.11.49 XORW: Alias of EORW 1516

Kalray – all rights reserved

6.11.50 ZXBD: Zero Extend Byte to Double Word . . . 1517
6.11.51 ZXHD: Zero Extend Half Word to Double Word . . . 1518
6.11.52 ZXWD: Zero Extend Word to Double Word . . . 1519
6.11.53 XMT44D: Extension Matrix Transpose 4x4 Double Words . . . 1520
6.11.54 XNORO: Alias of XNIORO . . . 1521
6.11.55 XNXORO: Alias of XNEORO . . . 1522
6.11.56 XORNO: Alias of XIORNO . . . 1523
6.11.57 XORO: Legacy form of XIORO . . . 1524
6.11.58 XXORO: Alias of XEORO . . . 1525
6.11.59 FADDDC: Floating-Point Add From Double Word Complex . . . 1526
6.11.60 FADDWC: Floating-Point Add From Word Complex . . . 1527
6.11.61 FADDWCP: Floating-Point Add Word Complex Pair . . . 1528
6.11.62 FSBFDC: Floating-Point Subtract Double Word Complex . . . 1529
6.11.63 FSBFWC: Floating-Point Subtract From Word Complex . . . 1530
6.11.64 FSBFWCP: Floating-Point Subtract From Word Complex Pair . . . 1531
6.12 Format Descriptions . . . 1532
6.12.1 BCU Opcode Error . . . 1532
6.12.2 BCU PCREL11 Dummy . . . 1532
6.12.3 BCU PCREL Dummy . . . 1532
6.12.4 BCU Break . . . 1532
6.12.5 BCU Move From Coprocessor Double Word . . . 1532
6.12.6 BCU Move From Coprocessor Quadruple Word . . . 1532
6.12.7 BCU Move From Coprocessor Octuple Word . . . 1532
6.12.8 BCU XACCESSO from 2-Register Buffer . . . 1533
6.12.9 BCU XACCESSO from 4-Register Buffer . . . 1533
6.12.10 BCU XACCESSO from 8-Register Buffer . . . 1533
6.12.11 BCU XACCESSO from 16-Register Buffer . . . 1533
6.12.12 BCU XACCESSO from 32-Register Buffer . . . 1533
6.12.13 BCU XACCESSO from 64-Register Buffer . . . 1533
6.12.14 BCU XCOPYO . . . 1533
6.12.15 BCU XALIGNO from 2-Register Buffer . . . 1534
6.12.16 BCU XALIGNO from 4-Register Buffer . . . 1534
6.12.17 BCU XALIGNO from 8-Register Buffer . . . 1534
6.12.18 BCU XALIGNO from 16-Register Buffer . . . 1534
6.12.19 BCU XALIGNO from 32-Register Buffer . . . 1534
6.12.20 BCU XALIGNO from 64-Register Buffer . . . 1534
6.12.21 BCU Reshape Vector to Vector . . . 1534
6.12.22 BCU Floating Point Scale Word Octuple . . . 1535
6.12.23 Extension Matrix Multiply Add Bytes to Words 0 . . . 1535
6.12.24 Extension Matrix Multiply Add Bytes to Words 1 . . . 1535
6.12.25 Extension Matrix Multiply Add Bytes to Words 2 . . . 1535
6.12.26 Extension Matrix Multiply Add Bytes to Words 3 . . . 1535
6.12.27 Extension Matrix Multiply Add Bytes to Words 0 . . . 1535
6.12.28 Extension Matrix Multiply Add Bytes to Words 1 . . . 1535

Kalray – all rights reserved

6.12.29 Extension Matrix Multiply Add Bytes to Words 2 . . . 1536
6.12.30 Extension Matrix Multiply Add Bytes to Words 3 . . . 1536
6.12.31 Extension Multiply Add 4x4 Matrix Byte to Word 0 . . . 1536
6.12.32 Extension Multiply Add 4x4 Matrix Byte to Word 1 . . . 1536
6.12.33 Extension Multiply Add Integer Floating Point Word Octuple . . . 1536
6.12.34 Extension Floating Point Fused Multiply Add Half to Word . . . 1536
6.12.35 Extension Floating-Point Matrix Multiply Add Half Words to Words . . . 1536
6.12.36 Extension Floating Point Narrow Word to Half Word . . . 1537
6.12.37 Extension Floating Point Minimum and Maximum . . . 1537
6.12.38 Extension Clamp Word Octuple . . . 1537
6.12.39 Extension Truncate Word to Byte Vector . . . 1537
6.12.40 Extension Extend Bytes to Words . . . 1537
6.12.41 Extension Coprocessor Splat Word Octuple to Hexadecuple . . . 1537
6.12.42 Extension Coprocessor Copy Word Hexadecuple . . . 1537
6.12.43 Extension Coprocessor Splat Word Octuple to Vector . . . 1538
6.12.44 Extension Coprocessor Copy Word Vector . . . 1538
6.12.45 Extension Bit Matrix Multiply . . . 1538
6.12.46 Extension Bitwise Octuple WOrd . . . 1538
6.12.47 BCU Conditional Branches . . . 1538
6.12.48 BCU Hardware Loop Setup . . . 1538
6.12.49 BCU TLB Management . . . 1538
6.12.50 BCU Instruction Pipeline Control . . . 1539
6.12.51 BCU Processing Group Instructions . . . 1539
6.12.52 BCU Half-Word Effects . . . 1539
6.12.53 BCU Half-Word Effects Alone . . . 1539
6.12.54 BCU Set System Registers . . . 1539
6.12.55 BCU Set System Register Alone . . . 1539
6.12.56 BCU Set System Register RA . . . 1539
6.12.57 BCU Get System Registers . . . 1540
6.12.58 BCU General and System Register Swap . . . 1540
6.12.59 BCU General and System Register Swap Alone . . . 1540
6.12.60 BCU Indirect Get System Registers . . . 1540
6.12.61 BCU Return from Call and System Calls . . . 1540
6.12.62 BCU Indirect Branch and Call from Register . . . 1540
6.12.63 BCU Supervisor Call Immediate . . . 1540
6.12.64 BCU Supervisor Call Register . . . 1541
6.12.65 BCU Unconditional Branches . . . 1541
6.12.66 LSU Load Scalar Base Offset . . . 1541
6.12.67 LSU Load Scalar Base Offset Extended . . . 1541
6.12.68 LSU Load Scalar Base Offset Extended Extended . . . 1541
6.12.69 LSU Load Quadruple Base Offset . . . 1541
6.12.70 LSU Load Quadruple Base Offset Extended . . . 1541
6.12.71 LSU Load Quadruple Base Offset Extended Extended . . . 1542
6.12.72 LSU Load Octuple Base Offset . . . 1542

Kalray – all rights reserved

6.12.73 LSU Load Octuple Base Offset Extended 1542
6.12.74 LSU Load Octuple Base Offset Extended Extended 1542
6.12.75 LSU Memory Effects Base Offset 1542
6.12.76 LSU Memory Effects Base Offset Extended 1542
6.12.77 LSU Memory Effects Base Offset Extended Extended 1543
6.12.78 LSU Memory and Cache Control 1543
6.12.79 LSU Memory Fence 1543
6.12.80 LSU Extension Load Octuple Base Offset 1543
6.12.81 LSU Extension Load Octuple Base Offset Extended 1543
6.12.82 LSU Extension Load Octuple Base Offset Extended Extended 1543
6.12.83 LSU Extension Load Octuple Cross 2 Base Offset 1544
6.12.84 LSU Extension Load Octuple Cross 2 Base Offset Extended 1544
6.12.85 LSU Extension Load Octuple Cross 2 Base Offset Extended Extended 1544
6.12.86 LSU Extension Load Octuple Cross 4 Base Offset 1544
6.12.87 LSU Extension Load Octuple Cross 4 Base Offset Extended 1544
6.12.88 LSU Extension Load Octuple Cross 4 Base Offset Extended Extended 1544
6.12.89 LSU Store Scalar Base Offset 1545
6.12.90 LSU Store Scalar Base Offset Extended 1545
6.12.91 LSU Store Scalar Base Offset Extended Extended 1545
6.12.92 LSU Store Quadruple Base Offset 1545
6.12.93 LSU Store Quadruple Base Offset Extended 1545
6.12.94 LSU Store Quadruple Base Offset Extended Extended 1545
6.12.95 LSU Store Octuple Base Offset 1546
6.12.96 LSU Store Octuple Base Offset Extended 1546
6.12.97 LSU Store Octuple Base Offset Extended Extended 1546
6.12.98 LSU Copy Octuple Word 1546
6.12.99 LSU Extension Store Octuple Base Offset 1546
6.12.100 LSU Extension Store Octuple Base Offset Extended 1546
6.12.101 LSU Extension Store Octuple Base Offset Extended Extended 1547
6.12.102 LSU Extension Store Octuple Cross 2 Base Offset 1547
6.12.103 LSU Extension Store Octuple Cross 2 Base Offset Extended 1547
6.12.104 LSU Extension Store Octuple Cross 2 Base Offset Extended Extended 1547
6.12.105 LSU Extension Store Octuple Cross 4 Base Offset 1547
6.12.106 LSU Extension Store Cross 4 Octuple Base Offset Extended 1547
6.12.107 LSU Extension Store Octuple Cross 4 Base Offset Extended Extended 1548
6.12.108 LSU Load Scalar Predicated Base 1548
6.12.109 LSU Load Scalar Predicated Base Offset 1548
6.12.110 LSU Load Scalar Predicated Base Double Offset 1548
6.12.111 LSU Load Quadruple Predicated Base 1548
6.12.112 LSU Load Quadruple Predicated Base Offset 1548
6.12.113 LSU Load Quadruple Predicated Base Double Offset 1549
6.12.114 LSU Load Octuple Predicated Base 1549
6.12.115 LSU Load Octuple Predicated Base Offset 1549
6.12.116 LSU Load Octuple Predicated Base Double Offset 1549

Kalray – all rights reserved

6.12.117 LSU Load Octuple Masked Base 1549
6.12.118 LSU Load Octuple Masked Base Offset 1549
6.12.119 LSU Load Octuple Masked Base Double Offset 1550
6.12.120 LSU Load Scalar Base + (Index*Scaling) 1550
6.12.121 LSU Load Quadruple Base + (Index*Scaling) 1550
6.12.122 LSU Load Octuple Base + (Index*Scaling) 1550
6.12.123 LSU Memory Effects Base + Index 1550
6.12.124 LSU Memory Effects by Set / Way 1550
6.12.125 LSU Extension Load Octuple Masked Base 1550
6.12.126 LSU Extension Load Octuple Masked Base Offset 1551
6.12.127 LSU Extension Load Octuple Masked Base Double Offset 1551
6.12.128 LSU Extension Load Octuple Cross 2 Masked Base 1551
6.12.129 LSU Extension Load Octuple Cross 2 Masked Base Offset 1551
6.12.130 LSU Extension Load Octuple Cross 2 Masked Base Double Offset 1551
6.12.131 LSU Extension Load Octuple Cross 4 Masked Base 1551
6.12.132 LSU Extension Load Octuple Cross 4 Masked Base Offset 1552
6.12.133 LSU Extension Load Octuple Cross 4 Masked Base Double Offset 1552
6.12.134 LSU Extension Preload 2-Register Buffer Packed Base 1552
6.12.135 LSU Extension Preload 2-Register Buffer Packed Base Offset 1552
6.12.136 LSU Extension Preload 2-Register Buffer Packed Base Double Offset 1552
6.12.137 LSU Extension Preload 4-Register Buffer Packed Base 1552
6.12.138 LSU Extension Preload 4-Register Buffer Packed Base Offset 1553
6.12.139 LSU Extension Preload 4-Register Buffer Packed Base Double Offset 1553
6.12.140 LSU Extension Preload 8-Register Buffer Packed Base 1553
6.12.141 LSU Extension Preload 8-Register Buffer Packed Base Offset 1553
6.12.142 LSU Extension Preload 8-Register Buffer Packed Base Double Offset 1553
6.12.143 LSU Extension Preload 16-Register Buffer Packed Base 1553
6.12.144 LSU Extension Preload 16-Register Buffer Packed Base Offset 1554
6.12.145 LSU Extension Preload 16-Register Buffer Packed Base Double Offset 1554
6.12.146 LSU Extension Preload 32-Register Buffer Packed Base 1554
6.12.147 LSU Extension Preload 32-Register Buffer Packed Base Offset 1554
6.12.148 LSU Extension Preload 32-Register Buffer Packed Base Double Offset 1554
6.12.149 LSU Extension Preload 64-Register Buffer Packed Base 1554
6.12.150 LSU Extension Preload 64-Register Buffer Packed Base Offset 1555
6.12.151 LSU Extension Preload 64-Register Buffer Packed Base Double Offset 1555
6.12.152 LSU Store Scalar Predicated Base 1555
6.12.153 LSU Store Scalar Predicated Base Offset 1555
6.12.154 LSU Store Scalar Predicated Base Double Offset 1555
6.12.155 LSU Store Quadruple Predicated Base 1555
6.12.156 LSU Store Quadruple Predicated Base Offset 1556
6.12.157 LSU Store Quadruple Predicated Base Double Offset 1556
6.12.158 LSU Store Octuple Predicated Base 1556
6.12.159 LSU Store Octuple Predicated Base Offset 1556
6.12.160 LSU Store Octuple Predicated Base Double Offset 1556

Kalray – all rights reserved

6.12.161 LSU Store Octuple Masked Base 1556
6.12.162 LSU Store Octuple Masked Base Offset 1557
6.12.163 LSU Store Octuple Masked Base Double Offset 1557
6.12.164 LSU Extension Store Octuple Masked Base 1557
6.12.165 LSU Extension Store Octuple Masked Base Offset 1557
6.12.166 LSU Extension Store Octuple Masked Base Double Offset 1557
6.12.167 LSU Extension Store Octuple Cross 2 Masked Base 1557
6.12.168 LSU Extension Store Octuple Cross 2 Masked Base Offset 1558
6.12.169 LSU Extension Store Octuple Cross 2 Masked Base Double Offset 1558
6.12.170 LSU Extension Store Octuple Cross 4 Masked Base 1558
6.12.171 LSU Extension Store Octuple Cross 4 Masked Base Offset 1558
6.12.172 LSU Extension Store Octuple Cross 4 Masked Base Double Offset 1558
6.12.173 LSU Atomic Load Scalar Base 1558
6.12.174 LSU Atomic Load Scalar Base Offset 1559
6.12.175 LSU Atomic Load Scalar Base Double Offset 1559
6.12.176 LSU Atomic Store Scalar Base 1559
6.12.177 LSU Atomic Store Scalar Base Offset 1559
6.12.178 LSU Atomic Store Scalar Base Double Offset 1559
6.12.179 LSU Atomic Load Add Scalar Base 1559
6.12.180 LSU Atomic Load Add Scalar Base Offset 1560
6.12.181 LSU Atomic Load Add Scalar Base Double Offset 1560
6.12.182 LSU Atomic Compare and Swap Scalar Base 1560
6.12.183 LSU Atomic Compare and Swap Scalar Base Offset 1560
6.12.184 LSU Atomic Compare and Swap Scala Base Double Offset 1560
6.12.185 LSU Atomic Compare and Swap Quadruple Base 1560
6.12.186 LSU Atomic Compare and Swap Quadruple Base Offset 1561
6.12.187 LSU Atomic Compare and Swap Quadruple Base Double Offset 1561
6.12.188 LSU Extension Load Base + (Index*Scaling) 1561
6.12.189 LSU Extension Load Octuple Cross 2 Base + (Index*Scaling) 1561
6.12.190 LSU Extension Load Octuple Cross 4 Base + (Index*Scaling) 1561
6.12.191 LSU Store Scalar Base + (Index*Scaling) 1561
6.12.192 LSU Store Quadruple Base + (Index*Scaling) 1561
6.12.193 LSU Store Octuple Base + (Index*Scaling) 1562
6.12.194 LSU Extension Store Octuple Base + (Index*Scaling) 1562
6.12.195 LSU Extension Store Quadruple Cross 2 Base + (Index*Scaling) 1562
6.12.196 LSU Extension Store Octuple Cross 4 Base + (Index*Scaling) 1562
6.12.197 ALU Make from Immediate 1562
6.12.198 ALU Make from Immediate Extended 1562
6.12.199 ALU Make from Immediate Extended Extended 1562
6.12.200 ALU Write Read Double Immediate 1563
6.12.201 ALU Write Read Double Immediate Extended 1563
6.12.202 ALU Write Read Double Immediate Extended Extended 1563
6.12.203 ALU Compare Write Read Immediate 1563
6.12.204 ALU Compare Write Read Immediate Extended 1563

Kalray – all rights reserved

6.12.205 ALU Compare Write Read Immediate Extended Extended 1563
6.12.206 ALU Extension Splat Double To Octuple Word Immediate 1564
6.12.207 ALU Extension Splat Double To Octuple Word Immediate Extended 1564
6.12.208 ALU Extension Splat Double To Octuple Word Immediate Extended Extended 1564
6.12.209 ALU Double Conditional Move Write Read Immediate 1564
6.12.210 ALU Double Conditional Move Write Read Immediate Extended 1564
6.12.211 ALU Double Conditional Move Write Read Immediate Extended Extended . 1564
6.12.212 ALU Bit Matrix Write Read Immediate 1565
6.12.213 ALU Bit Matrix Write Read Immediate Extended 1565
6.12.214 ALU Bit Matrix Write Read Immediate Extended Extended 1565
6.12.215 ALU Write Read Bitmask 1565
6.12.216 ALU Add PC to Immediate 1565
6.12.217 ALU Add PC to Immediate Extended 1565
6.12.218 ALU Add PC to Immediate Extended Extended 1566
6.12.219 ALU Write Read Immediate Word 1566
6.12.220 ALU Write Read Immediate Extended Word 1566
6.12.221 ALU Double Write Read Read 0 1566
6.12.222 ALU Double Write Read Read 0 Magic 1566
6.12.223 ALU Word Write Read Read 0 1566
6.12.224 ALU Word Write Read Read 0 Wrapped 1567
6.12.225 ALU Word Pair Write Read Read 0 1567
6.12.226 ALU Word Pair Write Read Read 0 Magic 1567
6.12.227 ALU Half Quadruple Write Read Read 0 1567
6.12.228 ALU Half Quadruple Write Read Read 0 Magic 1567
6.12.229 ALU Double Write Read Read 1 1567
6.12.230 ALU Double Write Read Read 1 Magic 1567
6.12.231 ALU Word Write Read Read 1 1568
6.12.232 ALU Word Write Read Read 1 Wrapped 1568
6.12.233 ALU Word Pair Write Read Read 1 1568
6.12.234 ALU Word Pair Write Read Read 1 Magic 1568
6.12.235 ALU Write Read Read Half Word Quadruple 1 1568
6.12.236 ALU Half Quadruple Write Read Read 1 Magic 1568
6.12.237 ALU Word to Double Write Read Read 2 1568
6.12.238 ALU Word to Double Write Read Read 2 Wrapped 1569
6.12.239 ALU Word to Double Write Read Read 3 1569
6.12.240 ALU Word to Double Write Read Read 2 Wrapped 1569
6.12.241 ALU Double Comparison Write Read Read 1569
6.12.242 ALU Double Write Read Read Special 1569
6.12.243 ALU Double Write Read Read Special Magic 1569
6.12.244 ALU Double Comparison Write Read Read 1569
6.12.245 ALU Double Comparison Write Read Read Wrapped 1570
6.12.246 ALU Write Read Read Special Word 1570
6.12.247 ALU Write Read Read Special Word Wrapped 1570
6.12.248 ALU Comparison Write Read Read Word Pair 1570

Kalray – all rights reserved

6.12.249 ALU Comparison Write Read Read Word Pair Magic 1570
6.12.250 ALU Write Read Read Special Word Pair 1570
6.12.251 ALU Write Read Read Special Word Pair Magic 1571
6.12.252 ALU Comparison Write Read Read Half Word Quadruple 1571
6.12.253 ALU Comparison Write Read Read Half Word Quadruple Magic 1571
6.12.254 ALU Write Read Read Special Half Quadruple 1571
6.12.255 ALU Write Read Read Special Half Quadruple Magic 1571
6.12.256 ALU Double Conditional Move Write Read Read 1571
6.12.257 ALU Bit Matrix Write Read Read 1571
6.12.258 ALU Bit Matrix Write Read Read Magic 1572
6.12.259 ALU Move Write Read Read Word Pair 1572
6.12.260 ALU Move Write Read Read Word Pair Magic 1572
6.12.261 ALU Move Write Read Read Half Word Quadruple 1572
6.12.262 ALU Move Write Read Read Half Word Quadruple Magic 1572
6.12.263 ALU Bit Write Read Double 1572
6.12.264 ALU Double Shift Rotate Write Read Immediate 1572
6.12.265 ALU Bit Write Read Word 1573
6.12.266 ALU Shift Rotate Write Read Immediate 1573
6.12.267 ALU Bit Write Read Word Pair 1573
6.12.268 ALU Word Pair Shift Rotate Write Read Immediate 1573
6.12.269 ALU Arithmetic Write Read Half Word Quadruple 1573
6.12.270 ALU Shift Rotate Write Read Immediate Half Word Quadruple 1573
6.12.271 ALU Double Shift Rotate Write Read Read 1573
6.12.272 ALU Shift Rotate Write Read Read 1574
6.12.273 ALU Double Write Read Read 1 Wrapped 1574
6.12.274 ALU Word Pair Shift Rotate Write Read Read 1574
6.12.275 ALU Word Pair Shift Rotate Write Read Read Magic 1574
6.12.276 ALU Shift Rotate Write Read Read Half Word Quadruple 1574
6.12.277 ALU Shift Rotate Write Read Read Half Word Quadruple Magic 1574
6.12.278 ALU Write Read Read Byte Octuple 0 1574
6.12.279 ALU Write Read Read Byte Octuple 0 Magic 1575
6.12.280 ALU Write Read Read Byte Octuple 1 1575
6.12.281 ALU Write Read Read Byte Octuple 1 Magic 1575
6.12.282 ALU Comparison Write Read Read Byte Octuple 1575
6.12.283 ALU Comparison Write Read Read Byte Octuple Magic 1575
6.12.284 ALU Write Read Read Special Byte Octuple 1575
6.12.285 ALU Write Read Read Special Byte Octuple Magic 1576
6.12.286 ALU Conditional Move Write Read Read Byte Octuple 1576
6.12.287 ALU Move Write Read Read Byte Octuple Magic 1576
6.12.288 ALU Shift Rotate Write Read Immediate Byte Octuple 1576
6.12.289 ALU Shift Rotate Write Read Read Byte Octuple 1576
6.12.290 ALU Shift Rotate Write Read Read Byte Octuple Magic 1576
6.12.291 ALU Floating-Point Comparison Write Read Read Scalar 1576
6.12.292 ALU Floating-Point Comparison Write Read Immediate Wrapped 1577

Kalray – all rights reserved

6.12.293 ALU Floating-Point Comparison Write Read Read Word Pair 1577
6.12.294 ALU Floating-Point Comparison Write Read Read Word Pair Magic 1577
6.12.295 ALU Floating-Point Comparison Write Read Read Half Word Quadruple . . 1577
6.12.296 ALU Floating-Point Comparison Write Read Read Half Word Quadruple Magic 1577
6.12.297 ALU Double Floating-Point Write Read 1577
6.12.298 ALU Floating-Point Write Read 1577
6.12.299 ALU Floating-Point Write Read Half Word Quadruple 1578
6.12.300 ALU Double Floating-Point Write Read Read 1578
6.12.301 ALU Floating-Point Write Read Word Long Latency 1578
6.12.302 ALU Floating-Point Write Read Word 1578
6.12.303 ALU Floating-Point Write Word Read Word Pair 1578
6.12.304 ALU Double Floating-Point Write Read Division 1578
6.12.305 ALU Floating-Point Comparison Negate Write Read Read Scalar 1578
6.12.306 ALU Floating-Point Comparison Negate Write Read Immediate Wrapped . . 1579
6.12.307 ALU Comparison Negate Write Read Read 1579
6.12.308 ALU Comparison Negate Write Read Immediate Wrapped 1579
6.12.309 ALU Reduce Write Read Read Word Pair 1579
6.12.310 ALU Word Pair Reduce Write Read Read Half Word Quadruple 1579
6.12.311 ALU Word Pair Reduce Write Read Read Half Word Quadruple 1579
6.12.312 ALU Move To Extension Double Word 0 1579
6.12.313 ALU Move To Extension Double Word 1 1580
6.12.314 ALU Move To Extension Double Word 2 1580
6.12.315 ALU Move To Extension Double Word 3 1580
6.12.316 ALU Move To Extension Quadruple Word 0 1580
6.12.317 ALU Move To Extension Quadruple Word 1 1580
6.12.318 ALU Extension Send Octuple Word to Channel 1580
6.12.319 ALU Extension Receive Octuple Word from Channel 1580
6.12.320 ALU Extension Send and Receive Octuple Word to/from Channels 1581
6.12.321 ALU No Operation 1581
6.12.322 MAU Floating-Point Multiply Add Word 1581
6.12.323 MAU Floating-Point Multiply Add Word Pair 1581
6.12.324 MAU Floating-Point Multiply Add Double 1581
6.12.325 MAU Floating-Point Multiply Add Word Quadruple 1581
6.12.326 MAU Floating-Point Multiply Add Double Pair 1581
6.12.327 MAU Floating-Point Multiply Complex Word 1582
6.12.328 MAU Floating-Point Multiply Word Complex Pair 1582
6.12.329 MAU Floating-Point Multiply Complex Double 1582
6.12.330 MAU Floating-Point Matrix Multiply Word 1582
6.12.331 MAU Floating-Point Matrix Multiply Add Word 1582
6.12.332 MAU Floating-Point Matrix Multiply Word Pair 1582
6.12.333 MAU Floating-Point Matrix Multiply Add Word Pair 1582
6.12.334 MAU Floating-Point Dual Multiply Add Word 1583
6.12.335 MAU Floating-Point Dual Multiply Add Word Pair 1583
6.12.336 MAU Floating-Point Dual Multiply Add Double 1583

Kalray – all rights reserved

6.12.337 MAU Floating-Point Dual Multiply Add Word Quadruple 1583
6.12.338 MAU Floating-Point Dual Multiply Add Double Pair 1583
6.12.339 MAU Floating-Point Multiply Add Word Complex 1583
6.12.340 MAU Floating-Point Multiply Add Word Complex Pair 1583
6.12.341 MAU Floating-Point Multiply Add Complex Double 1584
6.12.342 MAU Floating-Point Dual Multiply Dual Add Word 1584
6.12.343 MAU Floating-Point Dual Multiply Dual Add Word Pair 1584
6.12.344 MAU Floating-Point Dual Multiply Dual Add Double 1584
6.12.345 MAU Floating-Point Dual Multiply Dual Add Word Quadruple 1584
6.12.346 MAU Floating-Point Dual Multiply Dual Add Double Pair 1584
6.12.347 MAU Floating-Point Double Double 1584
6.12.348 MAU Double Double Double Double 0 1585
6.12.349 MAU Double Double Double Double 0 Magic 1585
6.12.350 MAU Quadruple Quadruple Double Double 1585
6.12.351 MAU Quadruple Quadruple Double Double Magic 1585
6.12.352 MAU Quadruple Double Double 0 1585
6.12.353 MAU Quadruple Double Double 0 Magic 1585
6.12.354 MAU Double Double Double 0 1585
6.12.355 MAU Double Double Double 0 Magic 1586
6.12.356 MAU Quadruple Quadruple Double Double 1 1586
6.12.357 MAU Quadruple Double Double 1 1586
6.12.358 MAU Quadruple Quadruple Double Double 2 1586
6.12.359 MAU Quadruple Quadruple Quadruple 1586
6.12.360 MAU Quadruple Quadruple Quadruple Quadruple 1586
6.12.361 MAU CRC 1586
6.12.362 MAU CRC Wrapped 1587
6.12.363 MAU Double Pair Pair 1587
6.12.364 MAU Single Single Single Single 1587
6.12.365 MAU Single Single Single Single Wrapped 1587
6.12.366 MAU Single Single Single Single 1587
6.12.367 MAU Single Single Single Wrapped 1587
6.12.368 MAU Quadruple Quadruple Double Double 3 1587
6.12.369 MAU Quadruple Double Double 3 1588
6.12.370 MAU Integer Division and Remainder 1588
6.12.371 MAU Copy Quadruple Word 1588
6.12.372 MAU Floating-Point Double Double Double Double 1588
6.12.373 MAU Floating-Point Double Double Double 1588
6.12.374 MAU Floating-Point Double Double Double 1 1588
6.12.375 MAU Floating-Point Single Single Single Single 1588
6.12.376 MAU Floating-Point Single Single Single 1589
6.12.377 MAU Floating-Point Quadruple Quadruple Double Double 1589
6.12.378 MAU Floating-Point Quadruple Double Double 1589
6.12.379 MAU Floating-Point Quadruple Quadruple Quadruple 2 1589
6.12.380 MAU Floating-Point Quadruple Quadruple Quadruple Quadruple 1589

Kalray – all rights reserved

6.12.381 MAU Floating-Point Quadruple Quadruple Quadruple 0 . . . 1589
6.12.382 MAU Floating-Point Quadruple Quadruple Quadruple 1 . . . 1589
6.13 Instruction Encoding Tables . . . 1590
6.14 Instruction Reservation Tables . . . 1612
6.14.1 Reservation Table ALL . . . 1612
6.14.2 Reservation Table ALU_FULL . . . 1613
6.14.3 Reservation Table ALU_FULL.X . . . 1613
6.14.4 Reservation Table ALU_FULL.Y . . . 1614
6.14.5 Reservation Table ALU_LITE . . . 1614
6.14.6 Reservation Table ALU_LITE.X . . . 1615
6.14.7 Reservation Table ALU_LITE.Y . . . 1615
6.14.8 Reservation Table ALU_LITE_CRWH . . . 1616
6.14.9 Reservation Table ALU_LITE_CRWL . . . 1617
6.14.10 Reservation Table ALU_NOP . . . 1617
6.14.11 Reservation Table ALU_TINY . . . 1618
6.14.12 Reservation Table ALU_TINY.X . . . 1618
6.14.13 Reservation Table ALU_TINY.Y . . . 1619
6.14.14 Reservation Table ALU_TINY_CRRP . . . 1619
6.14.15 Reservation Table ALU_TINY_CRRP_CRWL_CRWH . . . 1620
6.14.16 Reservation Table ALU_TINY_CRWH . . . 1620
6.14.17 Reservation Table ALU_TINY_CRWL . . . 1621
6.14.18 Reservation Table ALU_TINY_CRWL_CRWH . . . 1621
6.14.19 Reservation Table ALU_TINY_CRWL_CRWH.X . . . 1622
6.14.20 Reservation Table ALU_TINY_CRWL_CRWH.Y . . . 1622
6.14.21 Reservation Table BCU . . . 1623
6.14.22 Reservation Table BCU_CRRP_CRWL_CRWH . . . 1623
6.14.23 Reservation Table BCU_TINY_AUXW_CRRP . . . 1624
6.14.24 Reservation Table BCU_TINY_TINY_MAU_XNOP . . . 1625
6.14.25 Reservation Table BCU_XFER . . . 1625
6.14.26 Reservation Table EXT . . . 1626
6.14.27 Reservation Table LSU . . . 1626
6.14.28 Reservation Table LSU.X . . . 1627
6.14.29 Reservation Table LSU.Y . . . 1627
6.14.30 Reservation Table LSU_AUXR . . . 1628
6.14.31 Reservation Table LSU_AUXR.X . . . 1628
6.14.32 Reservation Table LSU_AUXR.Y . . . 1629
6.14.33 Reservation Table LSU_AUXR_AUXW . . . 1629
6.14.34 Reservation Table LSU_AUXR_AUXW.X . . . 1630
6.14.35 Reservation Table LSU_AUXR_AUXW.Y . . . 1630
6.14.36 Reservation Table LSU_AUXW . . . 1631
6.14.37 Reservation Table LSU_AUXW.X . . . 1632
6.14.38 Reservation Table LSU_AUXW.Y . . . 1632
6.14.39 Reservation Table LSU_CRRP . . . 1633
6.14.40 Reservation Table LSU_CRRP.X . . . 1634

Kalray – all rights reserved

6.14.41 Reservation Table LSU_CRRP.Y . . . 1634
6.14.42 Reservation Table MAU . . . 1635
6.14.43 Reservation Table MAU.X . . . 1635
6.14.44 Reservation Table MAU.Y . . . 1636
6.14.45 Reservation Table MAU_AUXR . . . 1636
6.14.46 Reservation Table MAU_AUXR.X . . . 1637
6.14.47 Reservation Table MAU_AUXR.Y . . . 1638
6.15 Instruction Constraints . . . 1639
6.15.1 Instruction Bundling Constraints . . . 1639
6.15.2 Instruction Scheduling Constraints . . . 1639

7 Application Binary Interface 1642
7.1 Introduction . . . 1642
7.2 Memory Addressing Model . . . 1642
7.3 Data Layout Rules . . . 1643
7.3.1 Native C Types . . . 1643
7.3.2 C Aggregate Types . . . 1643
7.3.3 Bit-Field Layout . . . 1643
7.4 Stack Frame Organization . . . 1644
7.4.1 Stack Frame Addressing . . . 1644
7.4.2 Stack Frame Regions . . . 1644
7.5 Nested Functions Implementation . . . 1646
7.5.1 Regarding cache alignement and cache invalidation . . . 1647
7.6 Function Arguments and Result . . . 1647
7.7 Function Linkage . . . 1648
7.7.1 Register Usage Conventions . . . 1648
7.7.2 Function Call Sequence . . . 1649
7.8 System Call Interface . . . 1650
7.9 Relocation List . . . 1651

8 Application Optimization Guide 1664
8.1 Uses of the Bit Matrix Multiplies 8×8 . . . 1664
8.1.1 Bit Matrix 8×8 Transposition and Multiplication . . . 1664
8.1.2 Kalray kv3 Bit Matrix 8×8 Instructions . . . 1665
8.1.3 Byte and Half-Word Manipulations . . . 1666
8.1.4 Parallel Bit Manipulations . . . 1666
8.1.5 Multiplication by Constants in GF(2^8) . . . 1668
8.1.6 Basis Change in GF(2^8) . . . 1669
8.1.7 Conversions for Bit Slicing . . . 1669

Kalray – all rights reserved

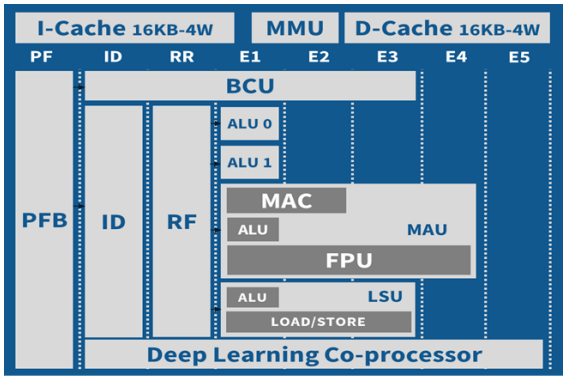


Figure 1: Kalray kv3 VLIW core block diagram.

1 Core Overview

The Kalray kv3 core implements a 32-bit and 64-bit 6-issue Very Long Instruction Word (VLIW) architecture with a 8-stage instruction pipeline. Groups of instructions that issue together are encoded by the compiler into *instruction bundles* (or "bundles"). A distinguishing feature of VLIW execution is that a swap of two registers can be done by two copy instructions placed inside the same bundle.

The Kalray kv3 core architecture has 64×64-bit general-purpose registers (GPRs) that can be fully operated on both in 32-bit and 64-bit execution mode. It includes arithmetic support for both 32-bit and 64-bit scalars as well as SIMD on quadruple 16-bit data and 32-bit paired data. The Kalray kv3 core architecture is byte-addressable little endian. The architecture is also a bit-little endian, in the sense that bit 0 of any data is always the least significant bit (little endian), and is shown at the rightmost position in figures.

1.1 Datapath Organization

The Kalray kv3 core includes two arithmetic and logic units, a multiply-accumulate/floating-point unit, a load/store unit, and a branch and control unit. These five execution units are primarily connected through a shared register file of 64×64-bit general-purpose registers, that allows reading up to nine 64-bit registers plus one 256-bit register quadruple, and writing up to two 64-bit registers plus two 256-bit register quadruple per cycle. The five execution units also share system function registers (SFRs) for program control and compute status. A Prefetch Buffer (PFB) is in charge of fetching the instructions and preparing them for dispatch to the execution units. The block diagram of the Kalray kv3 core is shown on Figure 1.

The Kalray kv3 core is associated with a coprocessor operating as a tightly-coupled accelerator (TCA). On the current Kalray kv3 core, the coprocessor includes a datapath and operators designed for deep learning acceleration. It features 48×256-bit registers, and associating them in pair or quadruples provide 512-bit and 1024-bit operands (Figure 2).

Kalray – all rights reserved

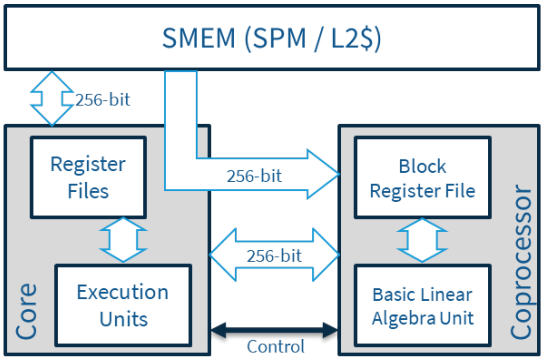


Figure 2: Kalray kv3 VLIW core tensor coprocessor.

1.1.1 Pipelined Implementation

The execution pipeline comprises five stages: E1, E2, E3, E4, and E5 (coprocessor); all executions begin in E1, with the exception of branches that are computed and taken as soon as possible, at ID or RR stage, to reduce the branch penalty. Most operands are read or bypassed for execution at the end of RR. This execution pipeline allows arithmetic and load/store instructions to execute for up to four cycles if needed. The results of instructions that complete earlier than E4 are made available for bypassing as operands to subsequent instructions.

The instruction pipeline of the Kalray kv3 core is interlocked, i.e., run-time stalls occur whenever an instruction tries to access a register that is not data-ready.

1.1.2 Prefetch Buffer (PFB)

Within the Kalray kv3 core, a prefetch buffer (PFB) is responsible for issuing instruction bundles to the core. The PFB manages 4 lines of 4×32-bit and receives blocks of 4×32-bit from the instruction cache every clock cycle, assuming that these blocks are in the instruction cache (fetch hit) and do not cross a level-1 cache line boundary (64-byte).

The PFB attempts to fetch ahead in order to keep its 4 lines of 4×32-bit full. When a branch is taken, the PFB is flushed and a new fetch starts from the target address. After a branch, the PFB takes one cycle to fetch the next bundle from the cache. Branching to a bundle that is split over two cache lines requires two cycles to fetch the bundle.

The PFB also provides 0-cycle overhead hardware loop support, allowing to automatically jump from the end to the beginning of a hardware loop without branch penalty.

1.1.3 Branch and Control Unit (BCU)

The Kalray kv3 core has a branch and control unit (BCU). This unit supports PC-relative and indirect branches or calls. The branch unit also supports hardware looping, also known as zero overhead loops. Due to instruction pipelining, all taken branches (not including hardware looping) involve run-time stalls: one cycle for unconditional direct branches and two cycles for conditional and indirect branches. Not taken conditional branches involve no stalls. Branch stalls

Kalray – all rights reserved

are not visible at the architecture level, that is, there are no delayed branches.

In case of MPPA® clusters, each Kalray kv3 PE core is connected to its direct predecessor and successor by wires that transport events for direct synchronization. The BCU executes all instructions related to event operations.

See Section 6.4 for BCU instructions.

1.1.4 Arithmetic and Logic Units (ALUs)

The Kalray kv3 core has two main arithmetic and logic units (ALU0 and ALU1). Each unit is capable of executing one instruction per cycle on 64-bit integers or packed 32-bit data. The results of the ALUs are 64-bit wide and can be used as operands of the next instruction bundle; This is equivalent to a pipeline depth of one cycle.

Integer signed, unsigned, and signed saturated instructions are available on each ALU, and also single cycle floating-point operations such as compare, min and max. In addition, ALU0 is capable of executing specific instructions such as PC-relative addition, 64-bit±64-bit with carry, and direct computation of 32-bit binary floating-point $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$. These reciprocal instructions have a latency of 15 cycles, but are fully pipelined.

See Section 6.6 for ALU instructions.

1.1.5 Multiply-Accumulate Unit (MAU)

The Kalray kv3 core multiply-accumulate unit (MAU) is pipelined with a depth of two cycles, executing one instruction every cycle. The multiply unit takes up to four 64-bit operands (two single 64-bit registers and one 128-bit register pair) and produces a single 64-bit or a pair of two 64-bits results at E2.

The MAU supports 64-bit by 64-bit integer multiplication with accumulation on 64 or 128 bits. It also supports multiplicative SIMD operations on vectors of two or four packed 32-bits words. Finally, it supports carryless multiplications for use in Galois field arithmetic, and flexible CRC computations.

In order to speed up integer intensive codes, the MAU also includes a 64-bit tiny ALU unit that is able to execute a subset of the 64-bit (and SIMD 32-bit) arithmetic and logic instructions if the MAU is not used for multiplication/accumulation instructions in the same bundle.

See Section 6.7 for MAU instructions.

1.1.6 Floating-Point Unit (FPU)

The Kalray kv3 core FPU includes a dual multiply-add unit (FMA) in single precision and a multiply-add unit in double precision. All floating-point instructions operate in conformance with the IEEE754-2008 standard for binary floating-point numbers, including sub-normals and four rounding modes. Correctly rounded division and square root instructions are available by executing Newton-Raphson iterations on values provided by seed generators for the 32-bit and 64-bit floating-point $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$.

The FPU of the Kalray kv3 core supports 32-bit binary floating-point dot-product of 32-bit pairs accumulated into 64-bit binary floating-point results. These instructions allow to evaluate

Kalray – all rights reserved

dot-product and polynomial evaluations in mixed precision. Finally, the FPU support a 2×2 matrix-multiply operation on pairs of 32-bit binary floating-point numbers, resulting into a 32-bit binary floating-point quadruple (128-bit operands held in register pairs).

As the FPU shares hardware with the MAU (including register file read ports and multipliers), FPU instructions cannot be bundled with integer MAU instructions.

See Section 6.8 for FPU instructions.

1.1.7 Load/Store Unit (LSU)

The Kalray kv3 core has a single load/store unit; This unit is pipelined with a latency of three cycles in the data cache, executing one instruction every cycle. The load/store unit can take up to three operands (two operands for the effective address computation, and one operand for the store instructions). Every clock cycle, the core is able to load or store 256 bits to/from a register quadruple, 128 bits to/from a register pair, or 64 bits to/from a register. When reading less than 64 bits from memory, the load instructions zero-extend or sign-extend the memory datum to the 64 bits of the destination register. The coprocessor may only load or store 256 bits to/from a coprocessor register.

The LSU also supports 3 different atomic operations (fetch-and-add, compare-and-swap, load-and-clear) for both 32-bit and 64-bit data sizes.

The Kalray kv3 core supports the following five addressing modes:

- A base address register plus a 10-bit signed, unscaled immediate constant.
- A base address register plus a 37-bit signed, unscaled immediate constant.
- A base register plus or minus an index register, optionally scaled by the datum size in memory.
- A base register, access conditionally executed depending on a test of another register.
- A base register plus a 27-bit signed, unscaled immediate constant, access conditionally executed depending on a test of another register.

In non device memory areas, the Kalray kv3 core supports memory accesses with misaligned effective addresses for regular loads/stores. Atomic instructions and accesses to device memory areas must be aligned on the size of the accessed object. If not, the Kalray kv3 core will take a DMISALIGN trap.

In order to speed up integer intensive codes, the LSU also includes a 64-bit light ALU unit that is able to execute a subset of the 64-bit (and SIMD 32-bit) arithmetic and logic instructions if the LSU is not used for loads/stores instructions in the same bundle.

See Section 6.5 for LSU instructions.

1.2 Data Types and Representations

The Kalray kv3 core instruction set architecture (ISA) efficiently supports 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit scalar types, and also 16-bit or 32-bit data types packed into 64-bit containers.

Kalray – all rights reserved

Traditional SIMD computing involving 16-bit or 32-bit numbers packed in 128-bit containers found in other architectures is emulated on the Kalray kv3 core by dual-issuing instructions that operate on data types packed into 64-bit registers. The non-packed data types are interpreted as signed integers, unsigned integers, fractional numbers, and floating-point numbers.

1.2.1 Signed Integers

An integer is expressed as a binary number of 2's complement and is 8-, 16-, 32-, 64-bit, or 128-bit long. Regardless of its length, the bit 0 of an integer is always the least significant bit. Because 2's complement is used, the most significant bit is used as a sign bit.

Data Length	Value Range
8 bits	-2^{7} to $+2^{7}-1$
16 bits	-2^{15} to $+2^{15}-1$
32 bits	-2^{31} to $+2^{31}-1$
64 bits	-2^{63} to $+2^{63}-1$
128 bits	-2^{127} to $+2^{127}-1$

1.2.2 Unsigned Integers

An unsigned integer is a binary number whose most significant bit has a positive weight and is 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit long.

Data Length	Value Range
8 bits	0 to $+2^{8}-1$
16 bits	0 to $+2^{16}-1$
32 bits	0 to $+2^{32}-1$
64 bits	0 to $+2^{64}-1$
128 bits	0 to $+2^{128}-1$

1.2.3 Floating-Point Numbers

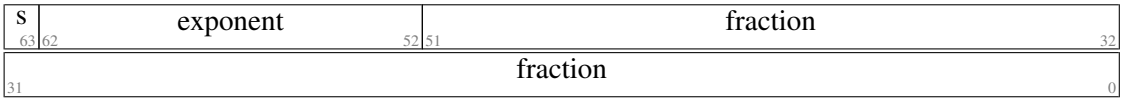
The Kalray kv3 core represents floating-point numbers in little-endian binary format, according to the IEEE754 2008 standard.

For 16-bit binary floating-point numbers, the representation is as follows:

For 32-bit binary floating-point numbers, the representation is as follows:

For 64-bit binary floating-point numbers, the representation is as follows:

Kalray – all rights reserved



1.2.4 Data Transfers

Data transfers between the general purpose registers of the core and data memory can be of 8, 16, 32, 64, 128, or 256 bits:

- 8-bit, 16-bit and 32-bit data can be loaded as scalars, one at a time, using LBS/LBZ, LHS/LHZ, and LWZ/LWS instructions respectively. They are then sign-extended/zero-extended to 64 bits as appropriate before being stored into the target register.
- When loading 64-bit data, they are directly stored into the target register.
- When loading 128-bit data, they are directly stored into the target register pair (R_i and R_{i+1}, with i even).
- When loading 256-bit data, they are directly stored into the target register quadruple (R_i and R_{i+3}, with i even).

1.2.5 Materialization of Constants

The Kalray kv3 core allows to construct 64-bit constants in any general purpose register, by using the MAKE instruction. The following table summarizes the size of immediate constants depending on the type of instruction format:

Instruction Type	Immediate Constant	Instruction Size
MAKE	16-bit signed	32-bit
MAKE	37-bit	64-bit
MAKE	64-bit	96-bit
ALU, MAU, LSU	10-bit signed	32-bit
ALU, MAU, LSU	37-bit	64-bit

1.3 VLIW Instruction Bundles

1.3.1 Instruction Bundle Memory Structure

On the Kalray kv3 core, the bundles are composed of instructions, each of which is encoded with one to three 32-bit instruction words called *syllables*. A valid Kalray kv3 core bundle consists of a sub-sequence of up to eight syllables in the following order (increasing memory addresses):

```
BCUi TCAi ALU0i ALU1i MAUi LSUi ALU0x ALU1x MAUx LSUx
```

That is, instruction syllables are ordered by type, with BCU first, TCA second, ALU third, etc. In case of immediate extensions, the corresponding syllables appear at the end of the bundle in the same order (BCU and TCA instructions cannot have immediate extensions). Please refer to Section 6.2 for the details of instruction bundle encoding.

Kalray – all rights reserved

1.3.2 Instruction Bundles in Assembly Language

At the assembly language level, the instructions may appear in any order and the immediate extensions are not apparent. These extensions are inferred from the number of bits required to encode the immediate values. For instance, a valid bundle appears the assembly language level could appear as sub-sequence of:

```
        BCU
        TCA
        ALU0
        ALU1
        MAU
        LSU
        ;;
```

Precisely, each instruction is separated from the next one either by an end of line or by a ‘;’ character. The end of the bundle is marked by a ‘;;’ alone in its line. Like with most assembly languages, a single ‘#’ character introduces a comment that extends until the end of the line.

1.3.3 Instruction Bundle Resource Constraints

The other rules to ensure bundle validity are related to micro-architecture resource constraints. Please refer to Section 6.15 for detailed instruction bundling constraints.

The Kalray kv3 VLIW core execution model ensures that whenever a register appears as source operand and target operand of different instructions inside a bundle, the old register value is seen by the source operands. Precisely, the Write-After-Read (WAR) dependence latency is zero, the Write-After Write (WAW) latency is one, and the Read-After-Write (RAW) dependence latency is strictly positive.

1.4 Memory Subsystem

The Kalray kv3 core memory model is such as two accesses of the same nature (cached or uncached) issued by the same core to the same memory location are guaranteed to complete in program order. Accesses to different memory locations are not guaranteed to complete in any particular order. A FENCE instruction can be used to force all previously issued data memory accesses to complete before a new data memory access can be issued by the core.

Furthermore, misaligned write accesses crossing a 64-byte boundary may complete in a non-atomic way, i.e. an other core reading at the same misaligned address shortly after the write from the first core might see the update of the bottom part but not that of the top part or conversely. Then again, the execution of FENCE by a core guarantees that all the previous accesses of this core (be they misaligned or not) have completed before it can emit new data accesses.

Data L1 cache coherency is implemented using a directory in the shared memory (SMEM) of the clusters grouping 16 Kalray kv3 cores. As a result, a store from core 1 is guaranteed to be seen by a core 2 polling on the same SMEM address with loads after some time (equal to the time taken by the store to reach the memory plus a fixed number of cycles). If core 1 executes a FENCE after its store, it is guaranteed that the store has been made visible to core 2 before any new data access can be issued from core 1.

Kalray – all rights reserved

No hardware coherency mechanism exists with respect to the instruction cache. Instruction cache lines need to be manually invalidated (using IINVAL* instructions) to support, e.g. self-modifying code.

The Kalray kv3 core memory subsystem includes the following components:

- Processor cache arbiter (PCA).
- Instruction cache (IC).
- Data cache (DC).

1.4.1 Processor Cache Arbiter (PCA)

The Kalray kv3 core features separate instruction and data caches that support, respectively, the fetch of instructions and the load/store data accesses. These two independent caches are connected to the outside memory system by a simple arbiter: the PCA.

1.4.2 Instruction Cache (IC)

The instruction cache of the Kalray kv3 core receives fetch requests from the PFB and returns groups of up to four 32-bit syllables that will later be assembled into bundles by the prefetch buffer. It has 16K bytes capacity, is 4-way set associative with LRU (Least Recently Used) replacement policy and 64-byte lines.

When the instruction words requested by the PFB are not in the cache, they are fetched from the memory system (with a critical-word-first scheme) and stored into the cache. During this time the core may stall, depending on the state of the PFB. The requested syllables are then returned to the PFB.

To invalidate the instruction cache safely, the core must execute the IINVAL instruction followed by the BARRIER instruction. This will invalidate the whole instruction cache, and trigger a re-fetch that will flush the PFB and miss in the invalidated instruction cache.

The instruction cache is globally disabled by clearing the ICE bit of the PS register, see Section 2.3. The different instruction cache policies (cache enable, cache bypass) are set by the Memory Management Unit, see Section 4.

1.4.3 Data Cache (DC)

The data cache receives load/store/atomic requests from the LSU and returns the requested data to the core. It is write-through no write-allocate, has 16K bytes capacity, is 4-way set associative with LRU replacement policy and 64-byte lines.

Store requests are directly sent to the memory, updating the cache by the way if they hit. In case a cached store misses, no line allocation is done in the cache and the store is simply posted to the memory.

If a store is aligned, uncached (by way of PS.DCE = 0, see Section 2.3) or MMU cache policy (see Section 4) and PS.USE = 0 , then it is considered as “blocking”: the core will stop executing until it receives the answer of the memory system for that store: this allows to generate precise exceptions in case of memory bus error and thus ease the debug of (e.g.) code accessing devices.

Kalray – all rights reserved

In all other circumstances (misaligned or cached store or PS.USE = 1), stores are “posted”: the core does not wait for their answer and can continue executing, including submission of new LSU requests to the data cache.

When data to be loaded by the core are not in the cache (load miss), they are fetched from memory and copied into the cache (line allocation). During this time the core will stall if it tries to process an instruction that has a data dependency on the loaded data or if it tries to access the data cache again (hit-under-miss is not supported). As for the stores, an aligned uncached load with PS.USE = 0 will be considered as blocking.

If executing an load instruction with uncached modifier, or if PS.DCE = 0, or if the address of a data access falls in a device or uncached memory area (be it the default device memory map when the MMU is off or a device/uncached page with MMU on), then the access is considered as uncached: it is simply forwarded to the memory and, if applicable, the response is forwarded to the core, leaving the cache contents unchanged.

Kalray – all rights reserved

2 Architectural State

2.1 General Purpose Registers

The Kalray kv3 core provides 64 × 64-bit wide general purpose registers (GPRs) R_i ($i = 0$ to 63) used for for all scalar and SIMD data types that fit in 64-bit. In particular, integers, addresses, Boolean conditions, single-precision and double-precision floating-point. GPRs are used for:

- The target of load instructions or the source of store instructions.
- The source or target of arithmetic, logic, shift, bit-manipulation instructions.
- The source or target of move to or from the system function registers (SFRs).
- The base address used for the effective address calculation of a load/store instruction.
- Used by addressing modes as index or condition registers.

Data types larger than 64 bits are held into adjacent pairs or quadruples of general purpose registers.

The 64 GPRs are reset to 0x0 after RESET.

2.1.1 GPR Loads and Stores

Any GPR can be loaded with 64-bit data from memory. Data smaller than a 64-bit double-word (a byte, half-word or word) can also be loaded from the memory into any GPR. They are first zero- or sign-extended to 64-bit as appropriate before being moved into the 64-bit target register.

The 64-bit content of any GPR can be stored to memory. A datum smaller than 64 bits stored in a GPR can also be moved into memory. In that case only the least significant bits of a GPR are stored to memory.

For loads and store instructions, there is no difference between 32/64-bit integer data, or 32/64-bit floating-point data. As a result, the instruction set does not distinguish between 32/64-bit integer data memory accesses, or 32/64-bit floating-point data memory accesses.

It is also possible to load 128-bit or 256-bit data into respectively a pair or a quadruple of GPR. Register pairs must start at an even register number, which receives the 64 MSBs. Likewise, register quadruples must start at register number multiple of 4, which receives the 64 MSBs.

2.1.2 GPRs as Address Registers

The effective addresses of load/store instructions are calculated according to the addressing mode specified by the instructions. All addressing modes specify at least one address register for the base address, and a second one in case of indexed or conditional addressing modes. Consequently, any GPR can be used as either of the following:

- A base address of the addressing mode.
- An index register in case of indexed addressing modes.

Kalray – all rights reserved

- A condition register in case of conditional memory accesses.

When the Kalray kv3 core executes in 32-bit address mode, virtual addresses generated by the LSU, as well as on the fetch side, are always truncated to 32-bits. In particular, this means that PC can never be observed with a value above $2^{32} - 1$, also it is really a 41-bit SFR.

2.2 Execution Privilege Levels

The Kalray kv3 core architecture defines 4 privilege levels (PL) from 0 to 3, with decreasing execution privileges, PL0 being the most privileged. Each of these PLs owns its own set of banked SFRs and is entered either:

- When taking an exception from a less privileged (higher numbered) PL or from itself.
- When returning from a higher privileged (lower numbered) PL or itself, by executing RFE.

For example, PL1 can be entered when taking a trap/it/scall/dbg exception from PL3, PL2 or PL1. PL2 can be returned to by executing RFE from PL0, PL1 or PL2.

After reset, the processor starts executing in PL0, which owns all the resources of the core, including RW rights on all (writable) SFRs, MMU manipulation rights, and all instructions execution rights. PL0 may then choose to delegate ownership of some resources to less privilege levels, that may themselves delegate them further below.

A PL that owns a resource will always receive the exceptions raised by less privilege levels trying to access this resource. It will then take the appropriate action (for instance: end the offending process, or take some action on behalf of it) before (potentially) returning control to the trapping level.

2.3 System Function Registers

The System Function Registers (SFRs) are used for program and system control and as status registers. The SFRs are numbered from 0 to 63. Those listed in Table 1 are defined while the others are reserved for future use.

The SFRs are used for program/system control and as status registers. They can be used as:

- The source or target (implicit or explicit) of specific instructions.
- The source or target of a move to or from the GPRs.

The SFRs cannot be the target of load instructions or the source of store instructions. The value read from the reserved bits in the various SFRs is 0b0. Attempting to write to the reserved bits in the various SFRs has no effect. However, it may raise the hardware trap PRIVILEGE if the permissions are not sufficient.

After RESET, all SFRs are initialized. SFRs are described in the following subsections.

Privileged resources can only be accessed when the core is in privileged mode. When attempting to read privileged registers in the user mode, the result is 0. Attempts to write cause a PRIVILEGE or OPCODE trap, according to the following rules: in privileged mode the trap is always OPCODE. In the user mode it is PRIVILEGE if such access is allowed in the system mode, OPCODE if it is impossible also in privileged mode.

Kalray – all rights reserved

©2025 Kalray

ID	Names	Index	Description
PC	$pc $s0	0	Program Counter
PS	$ps $s1	1	Processor State
PCR	$pcr $s2	2	Processing Identification
RA	$ra $s3	3	Return Address
CS	$cs $s4	4	Compute Status
CSIT	$csit $s5	5	Compute Status arithmetic interrupt
AESPC	$aespc $s6	6	Arithmetic Exception Saved PC
LS	$ls $s7	7	Loop Start Address
LE	$le $s8	8	Loop Exit Address
LC	$lc $s9	9	Loop Counter
IPE	$ipe $s10	10	Inter Processor Event
MEN	$men $s11	11	Misc External Notifications
PMC	$pmc $s12	12	Performance Monitor Control
PM<i>	$pm0-$pm3 $s13-$s16	13 - 16	Performance Monitor <i>[0, 3]
PMSA	$pmsa $s17	17	Performance Monitor Saved Address
TCR	$tcr $s18	18	Timer Control
T0V	$t0v $s19	19	Timer 0 value
T1V	$t1v $s20	20	Timer 1 value
T0R	$t0r $s21	21	Timer 0 reload value
T1R	$t1r $s22	22	Timer 1 reload value
WDV	$wdv $s23	23	Watchdog Value
WDR	$wdr $s24	24	Watchdog Reload
ILE	$ile $s25	25	Interrupt Line Enable
ILL	$ill $s26	26	Interrupt Line Level
ILR	$ilr $s27	27	Interrupt Line Request
MMC	$mmc $s28	28	Memory Management Control
TEL	$tel $s29	29	TLB Entry Low
TEH	$teh $s30	30	TLB Entry High
IXC	$ixc $s31	31	Inter-Extension Communications
SYO	$syo $s32	32	Syscall Owners
HTO	$hto $s33	33	Hardware Trap Owners
ITO	$ito $s34	34	Interrupt Owners
DO	$do $s35	35	Debug Owners
MO	$mo $s36	36	Miscellaneous Owners
PSO	$pso $s37	37	PS register fields Owners
TPCM<i>	$tpcm0-$tpcm1 $s38-$s39	38 - 39	Trace PC message register <i>[0, 1]
RES<i>	$res40-$res40 $s40-$s40	40 - 40	OCE (Debug) Control for hardware watchpoints and breakpoints 0 and 1
DBA<i>	$dba0-$dba1 $s41-$s42	41 - 42	Breakpoint Address <i>[0, 1]
DWA<i>	$dwa0-$dwa1 $s43-$s44	43 - 44	Watchpoint Address <i>[0, 1]
MES	$mes $s45	45	Memory Error Status
WS	$ws $s46	46	Wake-Up Status
DC<i>	$dc0-$dc3 $s47-$s50	47 - 50	OCE (Debug) Control for hardware watchpoints and breakpoints <i>[0, 3
DBA<i>	$dba2-$dba3 $s51-$s52	51 - 52	Breakpoint Address <i>[2, 3]
DWA<i>	$dwa2-$dwa3 $s53-$s54	53 - 54	Watchpoint Address <i>[2, 3]
TPCM<i>	$tpcm2-$tpcm2 $s55-$s55	55 - 55	Trace PC message register <i>[2, 2]
TPCMC	$tpcmc $s56	56	Trace PC message control register
PM<i>	$pm4-$pm7 $s57-$s60	57 - 60	Performance Monitor <i>[4, 7]
PMC<i>	$pmc2-$pmc2 $s61-$s61	61 - 61	Performance Monitor Control <i>[2, 2]
SRHPC	$srhpc $s62	62	Last sequence rupture helper PC when uninferrable
FRCC	$frcc $s63	63	Free running cycle counter
SPC_PL<i>	spc_pl0-spc_pl3 $s64-$s67	64 - 67	Shadow PC PL <i>[0, 3]
SPS_PL<i>	sps_pl0-sps_pl3 $s68-$s71	68 - 71	Shadow PS PL <i>[0, 3]
EA_PL<i>	ea_pl0-ea_pl3 $s72-$s75	72 - 75	Effective Address PL<i>[0, 3]
EV_PL<i>	ev_pl0-ev_pl3 $s76-$s79	76 - 79	Exception Vector PL <i>[0, 3]
SR_PL<i>	sr_pl0-sr_pl3 $s80-$s83	80 - 83	System Register PL <i>[0, 3]
ES_PL<i>	es_pl0-es_pl3 $s84-$s87	84 - 87	Exception Syndrome PL <i>[0, 3]
SID_PL<i>	sid_pl0-sid_pl3 $s88-$s91	88 - 91	Stream ID PL <i>[0, 3]
SR1_PL0	$sr1_pl0 $s92	92	System Register 1 PL 0
SR1_PL1	$sr1_pl1 $s93	93	System Register 1 PL 1
SR1_PL2	$sr1_pl2 $s94	94	System Register 1 PL 2
SR1_PL3	$sr1_pl3 $s95	95	System Register 1 PL 3
SYOW	$syow $s96	96	Alias to SYO register
HTOW	$htow $s97	97	Alias to HTO register
ITOW	$itow $s98	98	Alias to ITO register
DOW	$dow $s99	99	Alias to DO register
MOW	$mow $s100	100	Alias to MO register
PSOW	$psow $s101	101	Alias to PSO register
RES<i>	$res102-$res103 $s102-$s103	102 - 103	Reserved
TPCC_PL<i>	$tpcc_pl0-$tpcc_pl3 $s104-$s107	104 - 107	Trace PC Context
RES<i>	$res108-$res127 $s108-$s127	108 - 127	Reserved
SPC	$spc $s128	128	Shadow PC alias on SPC_PL<i>
RES<i>	$res129-$res131 $s129-$s131	129 - 131	Reserved
SPS	$sps $s132	132	Shadow PS alias on PS_PL<i>
RES<i>	$res133-$res135 $s133-$s135	133 - 135	Reserved

Kalray – all rights reserved

2.3.1 Program Counter (PC)

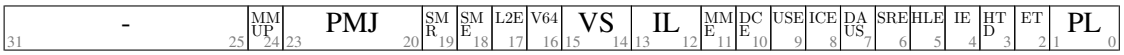
The Program Counter (PC) contains the address of the current instruction. As the bundles are at least four bytes wide, the least significant two bits of PC are always cleared. Write to this register is not allowed.

PC is the source for the calculations of the real addresses in all PC-relative branches, as well as the initialization of PC-relative loop start/end registers.

PC is automatically copied into SPC when an exception (hardware trap, interrupt, system call) occurs.

After RESET, the PC is initialized with a configurable address depending on input pins, but bits 0 to 11 have to be set to 0x000.

2.3.2 Processing Status (PS)



The Processing State (PS) register is composed of the following bit-fields:

PL Privilege Level

Current privilege level, forced to PL$< i >$ on exception.

After RESET, the core is in privilege level 0.

ET Exception Taken

The ET bit is used to detect unrecoverable errors (double trap): no exception should be taken while this bit is set otherwise the execution is halted.

The ET bit is set when an exception (SCALL, taken interrupt, trap) is taken. It can also be set by software. It is the software responsibility to clear it in exception handler prologue if exception nesting is required.

After RESET, ET is set.

HTD Hardware Trap Disable

If HTD is set and ET is clear, traps other than parity and ECC errors do not branch to exception handler and do not modify PM, IE and ET. The EC field and the EA register are always updated. HTD is set by software or by the break instruction.

before		after		Debug enable	Exception type	Comments
ET	HTD	ET	HTD			
0	0	1	0	†	any	`SPS` ← `PS`, `SPC` ← `PC`, `PC` ← `EV`
0	1	1	1	†	SCALL/int/trap (ECC, Parity)	same as above
0	1	0	1	†	trap(not ECC, not Parity)	`PS.EC` ← `cause`, `EA` ← `@address`, bundle canceled
1	0/1	1	1	1	any	trap caught by debugger
1	0	1	0	0	any	core halt and notification
1	1	1	1	0	any	core halt and notification

†: regular debug behavior

Kalray – all rights reserved

After RESET, HTD is cleared, so all hardware traps are enabled.

Notes Notification of double exception is defined outside the core (status bit in watchdog). All exceptions disable interrupts, to avoid taking another interrupt when software is not ready. The code of exception handlers must be at least partly mapped in TLB.

IE Interrupt Enable

If set, the interrupts are enabled. This bit is automatically cleared when the PS is copied into the SPS upon acknowledging an exception or interrupt.

After RESET, IE is cleared, so interrupts are disabled.

HLE Hardware Loop Enable

If set, the hardware loops are enabled. This bit is automatically cleared when the PS is copied into the SPS by exception or interrupt processing.

After RESET, HLE is cleared, hardware loops are disabled.

SRE Software REserved bit

Can be used by software as convenience.

After RESET, SRE is cleared.

DAUS Data Accesses Use SPS settings

The PS.DAUS field allows to see the hardware TLBs, and more generally the memory system, through the eyes of a different PL than the current one, for data accesses and PROBETLB.

When PS.DAUS is set to 1, all the subsequent data accesses are made with the MMU and data cache settings of SPS (that is an alias to SPS_PL$< i >$ with $< i >$ the current PL), instead of those of PS. Fetch accesses are not sensitive to this bit and will always use the settings of PS. This means that data accesses are presented to the MMU and the memory system using SPS.MME, SPS.MMUP, SPS.V64, SPS.VS, SPS.PMJ, SPS.DCE, SPS.USE and SPS.L2E instead of PS.MME, PS.MMUP, PS.V64, PS.VS, PS.PMJ, PS.DCE, PS.USE and PS.L2E respectively (whatever the value of SPS.DAUS). The ASN used will always be MMC.ASN whatever the value of the PS.DAUS. As soon as PS.DAUS is cleared, data accesses are back to normal, using PS MMU and data cache settings. This may be useful when (e.g.) trying to debug an OS from a more privileged layer, itself running with a different MMU configuration (on/off, 32/64, different VS ...).

After RESET, DAUS is cleared.

ICE Instruction Cache Enable

If set, the L1 instruction cache is enabled.

After RESET, ICE is cleared, so the instruction cache is disabled.

Kalray – all rights reserved

USE Uncached Streaming Enable

If set, uncached data accesses work in streaming mode (not stalling). If unset, non-streaming (stalling) accesses are used.

After RESET, USE is cleared, so the streaming mode is disabled.

DCE Data Cache Enable If set, the L1 data cache is enabled.

After RESET, DCE is cleared, so the data cache is disabled.

MME Memory Management Enable

If set, the memory management unit is enabled.

After RESET, MME is cleared, so the memory management is disabled.

IL Interrupt Level

An interrupt priority must be strictly higher than the core Interrupt mask Level (IL) to be taken. An IL value of 0xF prevents any interrupt to be accepted. This implies that a level 0 interrupt cannot be accepted.

After RESET, IL is set to 0xF, so interrupts are disabled.

VS Virtual Space.

See Section 4.9

After RESET, VS is cleared.

V64 (Virtual) 64 bits mode.

If set, inform that the core uses 64-bit addresses.

After RESET, V64 is cleared.

L2E L2 cache enable

If set, makes data and fetch accesses use the L2 cache when applicable, that is if the MMU page cache policy (or default mapping cache policy when the MMU is disabled) for the access is cached and the L2 cache is globally enabled for the cluster.

After RESET, L2E is cleared.

SME Step Mode Enable

If set, inform that the core has enabled the step mode.

After RESET, SME is cleared.

SMR Step Mode Ready

If set, inform that the core step mode is ready.

After RESET, SMR is cleared.

Kalray – all rights reserved

PMJ Page size Mask in JTLB

This is a 4 bits mask where bit $< i >$ configures if the $< i >$th supported page size may be present in the JTLB or not. Bit 0 corresponds to the smallest page size (4KB) and bit 3 to the largest (1GB). For example, PS.PMJ = 0xD means that 4KB, 2MB and 512MB pages are supported but not the 64KB ones.

When a page size is supported, a page of that size may be installed in the JTLB by the execution of the appropriate TLBWRITE maintenance instruction and the hardware will take this size into account when performing a JTLB look-up. Enabling/disabling some page sizes in JTLB trades performance (micro-TLB miss penalty via JTLB lookup latency) versus comfort of programming.

After RESET, PMJ is cleared.

MMUP Privileged in MMU

Set "Privileged" (1) or "User" (0) for the current level PL inside a virtual space (VS). See Section 4.9

After RESET, MMUP is set.

2.3.3 Processing Configuration Register (PCR)

The Processing Configuration Register (PCR) provides the revision number of the core, but also the hardware options of the current core instance and some global settings like Coprocessor, L1 cache coherency or ECC exception mode.

63 - 58	57 TPCIS	56 TPCUMP	55 TPCE	54 DSEM	53 L1CE	52 COE	51 - 44 BM	43 - 40 ST	39 - 32 SV

31 - 28 CMA	27 - 24 CAR	23 - 16 MID	15 - 8 CID	7 - 0 PID

The Processing Configuration Register (PCR) is composed of the following bit-fields:

PID Processing IDentifier

Processing identifier in compute cluster: [0..15] for the processing engine cores, 16 for the resource manager core.

CID Cluster IDentifier

Cluster identifier in system. For compute clusters: [0..15].

Set by pins on reset.

MID MPPA IDentifier

Revision identifier of the MPPA.

Set by pins on reset.

CAR Core ARchitecture revision

Revision identifier of the hardware architecture.

After RESET, CAR is 0x0 for the Kalray kv3 core.

Kalray – all rights reserved

CMA Core MicroArchitecture revision

Revision identifier of the hardware implementation of the architecture.

After RESET, CMA is 0x0 for Coolidge V1.

SV SoC version

Revision identifier of the System On Chip.

Set by pins on reset (Coolidge V1 = 0x0).

ST SoC type version

Revision identifier of the System On Chip type.

Set by pins on reset (MPPA_80 = 0x0).

BM Boot Mode.

To be completed.

Set by pins on reset.

COE COprocessor Enable

If set, enable coprocessor instructions.

After RESET, COE is enabled.

L1CE L1 Cache Coherency Enable

If set, enables the hardware L1 data cache coherency mechanism.

After RESET, L1CE is cleared.

DSEM Data Simple ECC Exception Mode

If set, the core will take a DSECCERROR hardware trap on uncached aligned loads/stores when PS.USE = 0 and the memory system signals a SECC error. If cleared, the same circumstances will raise an interrupt instead.

After RESET, DSEM is cleared.

TPCE Trace PC enable

If set, PC trace is enabled. After RESET, TPCE is cleared.

TPCUM Trace PC Uninferable Mode

If set, PC trace reports inferable jumps (call & goto instructions) as uninferable for call-grind support. After RESET, TPCUM is cleared.

TPCIS Trace PC Ignore Stall

If set, the PC trace stall is ignored in the processor. After RESET, TPCIS is set.

PCR RESET value depends on the hardware configuration for identifiers.

Kalray – all rights reserved

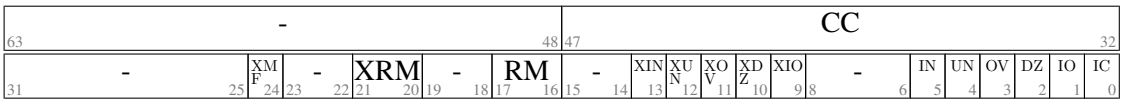
©2025 Kalray

2.3.4 Return Address (RA)

When a function call is performed the value of the next PC is saved to the Return Address (RA) register. When an RET instruction is executed, the RA register (containing the return address) is copied into the PC register.

The RA register has to be saved/restored within a non-leaf function (that is, a function which calls another function), the calling code must take care of the RA register by explicitly copying to a free general-purpose register or to the stack before calling any function, and by explicitly restoring it from the register or the stack before returning.

2.3.5 Compute Status (CS)



The Compute Status (CS) register is composed of the following bit-fields:

IC Integer Carry

The integer carry produced and consumed by instructions ADDCD and SBFCD.

IO IEEE 754 Invalid Operation

Invalid Operation condition of the IEEE 754 binary floating-point standard in core.

DZ IEEE 754 Divide by Zero

Divide by Zero condition of the IEEE 754 binary floating-point standard in core.

OV IEEE 754 Overflow

Overflow condition of the IEEE 754 binary floating-point standard in core.

UN IEEE 754 Underflow

Underflow condition of the IEEE 754 binary floating-point standard in core.

IN IEEE 754 Inexact

Inexact condition of the IEEE 754 binary floating-point standard in core.

XIO Extension IEEE 754 Invalid Operation

Invalid Operation condition of the IEEE 754 binary floating-point standard in coprocessor.

XDZ Extension IEEE 754 Divide by Zero

Divide by Zero condition of the IEEE 754 binary floating-point standard in coprocessor.

XOV Extension IEEE 754 Overflow

Overflow condition of the IEEE 754 binary floating-point standard in coprocessor.

Kalray – all rights reserved

XUN Extension IEEE 754 Underflow

Underflow condition of the IEEE 754 binary floating-point standard in coprocessor.

XIN Extension IEEE 754 Inexact

Inexact condition of the IEEE 754 binary floating-point standard in coprocessor.

RM IEEE 754 Rounding Mode

Rounding Mode of the IEEE 754 binary floating-point standard for the core: to nearest even (0b00), toward $+\infty$ (0b01), toward $-\infty$ (0b10), toward zero (0b11).

XRM Extension IEEE 754 Rounding Mode

Rounding Mode of the IEEE 754 binary floating-point standard for the coprocessor: to nearest even (0b00), toward $+\infty$ (0b01), toward $-\infty$ (0b10), toward zero (0b11).

CC Carry Counter

2.3.6 Compute Status Interrupt (CSIT)

The Compute Status Interrupt (CSIT) register is composed of the following bit-fields:

ICIE Integer Carry Interrupt Enable

Raise interrupt 13 if an instruction (such as ADDCD) outputs a carry and CS.AEIR = 0

IOIE IEEE 754 Invalid Operation Interrupt Enable

Raise interrupt 13 if a floating point instruction raises the IO flag and CS.AEIR = 0

DZIE IEEE 754 Divide by Zero Interrupt Enable

Raise interrupt 13 if a floating point instruction raises the DZ flag and CS.AEIR = 0

OVIE IEEE 754 Overflow Interrupt Enable

Raise interrupt 13 if a floating point instruction raises the OV flag and CS.AEIR = 0

UNIE IEEE 754 Underflow Interrupt Enable

Raise interrupt 13 if a floating point instruction raises the UN flag and CS.AEIR = 0

INIE IEEE 754 Inexact Interrupt Enable

Raise interrupt 13 if a floating point instruction raises the IN flag and CS.AEIR = 0

XIOIE Extension IEEE 754 Invalid Operation Interrupt Enable

Raise interrupt 13 if a coprocessor floating point instruction raises the IO flag and CS.AEIR = 0

Kalray – all rights reserved

©2025 Kalray

XDZIE Extension IEEE 754 Divide by Zero Interrupt Enable

Raise interrupt 13 if a coprocessor floating point instruction raises the DZ flag and CS.AEIR = 0

XOVIE Extension IEEE 754 Overflow Interrupt Enable

Raise interrupt 13 if a coprocessor floating point instruction raises the OV flag and CS.AEIR = 0

XUNIE Extension IEEE 754 Underflow Interrupt Enable

Raise interrupt 13 if a coprocessor floating point instruction raises the UN flag and CS.AEIR = 0

XINIE Extension IEEE 754 Inexact Interrupt Enable

Raise interrupt 13 if a coprocessor floating point instruction raises the IN flag and CS.AEIR = 0

AEIR Arithmetic Exception Interrupt Raised

This bit is automatically set to 1 by hardware if one of the above arithmetic flags, or one of the below coprocessor arithmetic flags is raised by an instruction and its corresponding *IE bit is set and CS.AEIR = 0. At this point, CS.AEC is updated with the code corresponding to the exception that fired, the AESPC register may receive the PC of the bundle responsible for the exception, and Interrupt Line 13 is asserted. The processor might then take IT 13 or not depending on its state and on Interrupt Controller settings.

AEC Arithmetic Exception Code

When AEIR is set to 1 by hardware, this field is updated with the code of the arithmetic exception that fired. If a given bundle triggers several (interrupt-enabled) arithmetic exceptions, then only one IT is generated and the exception with the lowest numeric code (among those that fired as early as possible in the pipeline) is chosen to fill AEC.

Exception Class	AEC
IC	000
IO	001
DZ	010
OV	011
UN	100
IN	101

SPCV SPC Valid

When AEIR is set to 1 by hardware, this field is set to 1 if the PC of the triggering instruction was saved in AESPC, and to 0 otherwise. This is dependent on micro-architectural conditions. Typically, flags output late in the pipeline, such as inexact, underflow and overflow of the floating point inverse instruction will not register their PC.

Kalray – all rights reserved

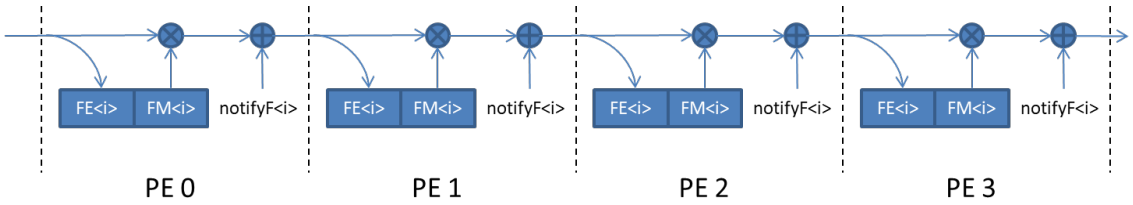


Figure 3: Forward event propagation between PE cores in a cluster.

2.3.7 Arithmetic Exception Saved PC (AESPC)

This register is automatically updated by hardware when an arithmetic exception interrupt is triggered (be it then taken by the core or not) with the PC of the bundle that raised the exception. More details about this mechanism can be found in the CS register section.

2.3.8 Hardware Loop Registers (LS, LE, LC)

Loop Start (LS) register Specifies the address of the first instruction bundle to be executed in the hardware loop.

Loop Exit (LE) register Specifies the address of the first instruction bundle after the hardware loop. Thus, the specified bundle is not part of the hardware loop. In case LE is equal to 0x0, the hardware loop is disabled.

Loop Count (LC) register Specifies the number of times the hardware loop is iterated. As a special case, when LC contains zero and the PC is within LS inclusive and LE exclusive, the hardware loop iterates forever. This case only occurs when the LOOPDO instruction is given the value 0.

2.3.9 Inter Process Event (IPE)

This register maintains the events received from neighboring cores when they execute SYNCGROUP instruction. This register also configures the event propagation mode between cores.

BM (63..48)	FM (47..32)
BE (31..16)	FE (15..0)

FE Forward Events

These 16 independent bits record events notified or transmitted by the predecessor PE core in the cluster (PE core numbered $n-1 \mod 16$ for PE core n).

BE Backward Events

Kalray – all rights reserved

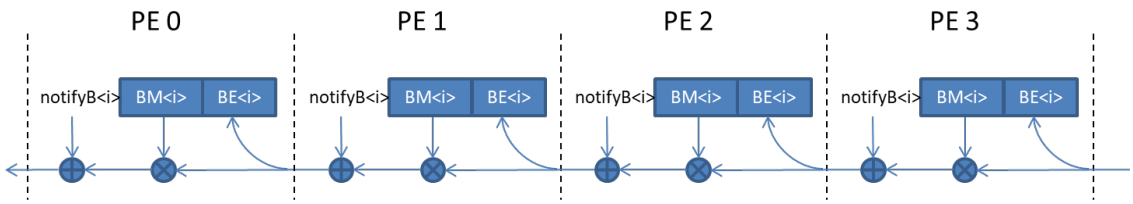


Figure 4: Backward event propagation between PE cores in a cluster.

These 16 independent bits record events notified or transmitted by the successor PE core in the cluster (PE core numbered $n+1 \mod 16$ for PE core n).

FM Forward Mode

These 16 independent configuration bits control the transmission of forward events between the current PE core n and its successor $n+1 \mod 16$. When bit FM_i of PE core n is set, any forward event i received by PE core n is also transmitted to PE core $n+1 \mod 16$, even if PE core n is not executing a SYNCGROUP instruction (Figure 3).

BM Backward Mode

These 16 independent configuration bits control the transmission of backward events between the current PE core n and its predecessor $n-1 \mod 16$. When bit FM_i of PE core n is set, any backward event i received by PE core n is also transmitted to PE core $n-1 \mod 16$, even if PE core n is not executing a SYNCGROUP instruction (Figure 4).

2.3.10 Miscellaneous External Notifications (MEN)

This register receives and registers miscellaneous external notifications incoming from the system. In the current Coolidge implementation of the cluster, only the RM (aka Secure Core) receives such external notifications, corresponding to l2 cache job FIFO notifications on bits [15:0]. Bits [63:16] are reserved and cannot be written (read as zero).

2.3.11 Performance Monitor Registers (PM*)

Please refer to Section 5.3 (Performance Monitoring) for details about Performance Monitor feature and associated control/status registers.

2.3.12 Real Time Clock and Watchdog Registers (T*) (W*)

Please refer to Section 5.2 (Timers and Watchdog) for details about Timers and Watchdog feature and associated control registers.

2.3.13 Memory Translation Registers (MMC, TEL and TEH)

Please refer to Section 4 (Memory Management Unit) section for details on the memory translation registers.

Kalray – all rights reserved

2.3.14 Syscall Owners (SYO)

This SFR defines which PL is the target of syscall (one configuration per quarter of the 4096 scall numbers).
This SFR is written through SYOW register.

31 - 8	7 - 6	5 - 4	3 - 2	1 - 0
-	Q3	Q2	Q1	Q0

Q0 Quarter 0: number of the PL that owns syscalls 0 to 1023

After RESET, Q0 is cleared.

Q1 Quarter 1: number of the PL that owns syscalls 1024 to 2047

After RESET, Q1 is cleared.

Q2 Quarter 2: number of the PL that owns syscalls 2048 to 3071

After RESET, Q2 is cleared.

Q3 Quarter 3: number of the PL that owns syscalls 3072 to 4095

After RESET, Q3 is cleared.

Notes This can easily be used by an hypervisor to define hypervisor calls, through the ownership of a slice of syscalls.

2.3.15 Hardware Trap Owners (HTO)

This SFR defines which PL is the target of hardware traps (one configuration per hardware trap).
This SFR is written through HTOW register.

A double exception is a special hardware trap defined as a horizontal exception attempt with PS.ET = 1. This may lead to a loss of context so it is treated specially:

- If the double exception happens in a PL that is less privileged than the PL owning the double exception, then the trap is taken to the owner PL.
- Otherwise, if the double exception happens in the PL owning the double exception itself, or in a more privileged one, then the trap is not taken to the owner PL. Instead execution is stopped, the processor power controller enters SLEEPING state and the double exception is signaled to the outside world under the form of a watchdog alert. The processor will not resume execution without the intervention of the outside environment, that will have to wake it up and acknowledge the watchdog (by reboot).

31 - 26	25 - 24	23 - 22	21 - 20	19 - 18	17 - 16	15 - 14	13 - 12	11 - 10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
-	PLO	VSFR	DE	A2CL	W2CL	PROT	NOMAP	SECCG	DECCG	DSYS	PSYS	DMIS	OPC

Kalray – all rights reserved

OPC PL that owns opcode hardware trap.

After RESET, OPC is set to level 0.

DMIS PL that owns data misaligned hardware trap.

After RESET, DMIS is set to level 0.

PSYS PL that owns program memory system hardware trap.

After RESET, PSYS is set to level 0.

DSYS PL that owns data memory system hardware trap.

After RESET, DSYS is set to level 0.

DECCG PL that owns the Double ECC hardware trap group: PDECC + DDECC + PPAR + DPAR + TPAR.

After RESET, DECCG is set to level 0.

SECCG PL that owns the Single ECC hardware traps group: PSECC + DSECC.

After RESET, SECCG is set to level 0.

NOMAP PL that owns the no-mapping hardware trap

After RESET, NOMAP is set to level 0.

PROT PL that owns protection hardware trap

After RESET, PROT is set to level 0.

W2CL PL that owns the write to clean hardware trap

After RESET, W2CL is set to level 0.

A2CL PL that owns the atomic to clean hardware trap

After RESET, A2CL is set to level 0.

DE PL that owns the double exception hardware trap

After RESET, DE is set to level 0.

VSFR PL that owns the VSFRs (Virtual SFRs) hardware trap

After RESET, VSFR is set to level 0.

PLO PL that owns the PL (privilege level) overflow hardware trap

After RESET, PLO is set to level 0.

Notes The privilege trap is the one that is triggered on resource rights violations attempts. As such, it has no owner bits in the HTO register itself as such a trap is, by definition, always taken to the registered owner PL of the resource.

Kalray – all rights reserved

Notes About the VSFR hardware trap: this trap is triggered when a less privileged PL tries to read or write one of SFRs[3840,4095]. It fills the same info in the owner ES_PL$< i >$ as a read/write to any other non-existing SFR (GET/SET/WFX*/RSWAP code + GPR idx + SFR idx), except that the HTC is set to a dedicated VSFR cause. It is there to provide a hook for virtual SFR accesses.

2.3.16 Interrupts Owners (ITO)

This SFR defines which PL is the target of interrupt (one configuration per interrupt).
This SFR is written through ITOW register.

Own an interrupt line means owning the related ILE + ILR + ILL fields (trap to the owner PL in case of less-privileged write attempt) + be the target of the interrupt exception when that interrupt fires.

IT31	IT30	IT29	IT28	IT27	IT26	IT25	IT24	IT23	IT22	IT21	IT20	IT19	IT18	IT17	IT16
63 62	61 60	59 58	57 56	55 54	53 52	51 50	49 48	47 46	45 44	43 42	41 40	39 38	37 36	35 34	33 32
IT15	IT14	IT13	IT12	IT11	IT10	IT9	IT8	IT7	IT6	IT5	IT4	IT3	IT2	IT1	IT0
31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0

IT_$< i >$ PL that owns interrupt line number $< i >$ with $0 \leq i \leq 31$.

After RESET, IT_$< i >$ is set to level 0.

2.3.17 Debug Owner (DO)

This SFR defines which PL is the owner for each debug resource.
This SFR is written through DOW register.

Own a debug resource means owning the related debug SFR / SFR fields (trap to the owner level in case of less-privileged write attempt) and be the target of the debug exception when that debug resource fires.

-	BI3	BI2	BI1	BI0	W3	W2	B3	B2	W1	W0	B1	B0
31 24	23 22	21 20	19 18	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0

B0 PL that owns the breakpoint 0. That groups the ownership of the following SFR / SFR field: DC0.BE and DBA0

After RESET, B0 is set to level 0.

B1 PL that owns the breakpoint 1. That groups the ownership of the following SFR / SFR field: DC1.BE and DBA1

After RESET, B1 is set to level 0.

B2 PL that owns the breakpoint 2. That groups the ownership of the following SFR / SFR field: DC2.BE and DBA2

After RESET, B2 is set to level 0.

Kalray – all rights reserved

B3 PL that owns the breakpoint 3. That groups the ownership of the following SFR / SFR field: DC3.BE and DBA3

After RESET, B3 is set to level 0.

W0 PL that owns the watchpoint 0. That groups the ownership of the following SFR / SFR field: DC0.WE, DC0.WSZ, DC0.WTYP and DWA0

After RESET, W0 is set to level 0.P

W1 PL that owns the watchpoint 1. That groups the ownership of the following SFR / SFR field: DC1.WE, DC1.WSZ, DC1.WTYP and DWA1

After RESET, W1 is set to level 0.

W2 PL that owns the watchpoint 2. That groups the ownership of the following SFR / SFR field: DC2.WE, DC2.WSZ, DC2.WTYP and DWA2

After RESET, W2 is set to level 0.

W3 PL that owns the watchpoint 3. That groups the ownership of the following SFR / SFR field: DC3.WE, DC3.WSZ, DC3.WTYP and DWA3

After RESET, W3 is set to level 0.

BI0 PL that owns the breakpoint instruction with id 0.

After RESET, BI0 is set to level 0.

BI1 PL that owns the breakpoint instruction with id 1.

After RESET, BI1 is set to level 1.

BI2 PL that owns the breakpoint instruction with id 2.

After RESET, BI2 is set to level 2.

BI3 PL that owns the breakpoint instruction with id 3.

After RESET, BI3 is set to level 3.

2.3.18 Miscellaneous Owners (MO)

This SFR defines which PL is the owner of various Kalray kv3 core resources, including some SFRs / SFR fields and instructions.
This SFR is written through MOW register.

For instructions, the registered PL owner is able to execute the instruction, as are more privileged PLs. Less privileged PLs trap (privilege) to the owner PL if trying to execute the instruction.

63 - 50	49 SRHPC 48	47 PM7 46	45 PM6 44	43 PM5 42	41 PM4 40	39 DISW 38	37 TPCM 36	35 COMM 34	33 PMIT 32
31 PM3 30	29 PM2 28	27 PM1 26	25 PM0 24	23 WD 22	21 T1 20	19 T0 18	17 CSIT 16	15 MES 14	13 MEN 12	11 MSG 10	9 PCR 8	7 SYNC 6	5 STOP 4	3 RFE 2	1 MMI 0

Kalray – all rights reserved

MMI Memory Management Instructions. PL that owns the PROBETLB, READTLB and WRITETLB instructions.

After RESET, MMI is set to level 0.

RFE PL that owns the RFE instruction.

After RESET, RFE is set to level 0.

STOP PL that owns the STOP instruction.

After RESET, STOP is set to level 0.

SYNC PL that owns the SYNCGROUP instruction. This field gives also ownership on IPE.

After RESET, SYNC is set to level 0.

PCR PL that owns the PCR register.

After RESET, PCR is set to level 0.

MSG PL that owns the MMU SFR group. This means owning MMC, TEL and TEH.

After RESET, MSG is set to level 0.

MEN PL that owns the MEN register.

After RESET, MEN is set to level 0.

MES PL that owns the MES register.

After RESET, MES is set to level 0.

CSIT PL that owns the CSIT register. This field gives also ownership on AESPC.

After RESET, CSIT is set to level 0.

T0 PL that owns the timer 0. This means owning T0V, T0R and timer 0 related bits in TCR (i.e.: TCR.T0CE, TCR.T0IE, TCR.T0ST and TCR.STI0).

After RESET, T0 is set to level 0.

T1 PL that owns the timer 1. This means owning T1V, T1R and timer 1 related bits in TCR (i.e.: TCR.T1CE, TCR.T1IE, TCR.T1ST and TCR.STI1).

After RESET, T1 is set to level 0.

WD PL that owns the watchdog. This means owning WDC, WDR and WDOG related bits in TCR (i.e.: TCR.WIE, TCR.WUI, TCR.WIS and TCR.SWI).

After RESET, WD is set to level 0.

PM0 PL that owns the Performance Monitor 0. This means owning PMC.PM0C and PM0.

After RESET, PM0 is set to level 0.

Kalray – all rights reserved

PM1 PL that owns the Performance Monitor 1. This means owning PMC.PM1C and PM1.

After RESET, PM1 is set to level 0.

PM2 PL that owns the Performance Monitor 2. This means owning PMC.PM2C and PM2.

After RESET, PM2 is set to level 0.

PM3 PL that owns the Performance Monitor 3. This means owning PMC.PM3C and PM3.

After RESET, PM3 is set to level 0.

PMIT PL that owns the Performance Monitor Interrupt Mechanism. This means owning PMC.SAT, PMC.SAF, PMC.SAV and PMSA. Ownership of PMC.PM$< i >$IE is obtained by a PL if it owns both PM$< i >$ and PMIT.

After RESET, PMIT is set to level 0.

COMM PL that owns the inter TCA Communication mechanism. This means owning XSENDO, XRECVO, XSENDRECVO instructions and IXC register.

After RESET, COMM is set to level 0.

TPCM PL that owns the Trace PC Message mechanism. This means owning TPCM0, TPCM1, TPCM2 and TPCMC regsiters.

After RESET, TPCM is set to level 0.

DISW PL that owns the DINVALLSW instruction.

After RESET, DISW is set to level 0.

2.3.19 Processor State Owner (PSO)

This SFR defines which PL is the owner of each bit field of PS.
This SFR is written through PSOW register.
For PL$< i >$, be the owner of a PS bit field means:

- Be able to read it / write it
- Be trapped to by less privileged PLs trying to write it
- Be able to read/write the corresponding bit in SPS_PL$< j >$, with $j \geq i$
- Be trapped to by less privileged PLs trying to write it

More privileged PL (those with $j < i$) are also able to read/write the considered PS bit, as well as its version in SPS_PL$< k >$ with $k \geq j$.

-								MMUP	PMJ3	PMJ2	PMJ1	PMJ0	SMR	SME	L2E	V64
63							50	49 48	47 46	45 44	43 42	41 40	39 38	37 36	35 34	33 32
VS1	VS0	IL1	IL0	MME	DCE	USE		ICE	DAUS	SRE	HLE	IE	HTD	ET	PL1	PL0
31 30	29 28	27 26	25 24	23 22	21 20	19 18		17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0

Kalray – all rights reserved

PL_0 PL that owns PS.PL[0].
After RESET, PL_0 is set to level 0.

PL_1 PL that owns PS.PL[1].
After RESET, PL_1 is set to level 0.

ET PL that owns the PS.ET.
After RESET, ET is set to level 0.

HTD PL that owns PS.HTD.
After RESET, HTD is set to level 0.

IE PL that owns PS.IE.
After RESET, IE is set to level 0.

HLE PL that owns PS.HLE.
After RESET, HLE is set to level 0.

ICE PL that owns PS.ICE.
After RESET, ICE is set to level 0.

USE PL that owns PS.USE.
After RESET, USE is set to level 0.

DCE PL that owns PS.DCE.
After RESET, DCE is set to level 0.

MME PL that owns PS.MME.
After RESET, MME is set to level 0.

IL_0 PL that owns PS.IL[0].
After RESET, IL_0 is set to level 0.

IL_1 PL that owns PS.IL[1].
After RESET, IL_1 is set to level 0.

VS_0 PL that owns PS.VS[0].
After RESET, VS_0 is set to level 0.

VS_1 PL that owns PS.VS[1].
After RESET, VS_1 is set to level 0.

V64 PL that owns PS.ICE.
After RESET, V64 is set to level 0.

Kalray – all rights reserved

L2E PL that owns PS.L2E.

After RESET, L2E is set to level 0.

SME PL that owns PS.SME.

After RESET, SME is set to level 0.

SMR PL that owns PS.SMR.

After RESET, SMR is set to level 0.

PMJ_0 PL that owns PS.PMJ[0].

After RESET, PMJ_0 is set to level 0.

PMJ_1 PL that owns PS.PMJ[1].

After RESET, PMJ_1 is set to level 0.

PMJ_2 PL that owns PS.PMJ[2].

After RESET, PMJ_2 is set to level 0.

PMJ_3 PL that owns PS.PMJ[3].

After RESET, PMJ_3 is set to level 0.

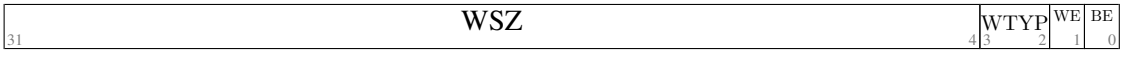
MMUP PL that owns PS.MMUP.

After RESET, MMUP is set to level 0.

2.3.20 Debug Control (DC0)

This SFR controls the hardware watchpoint 0 and the hardware breakpoint 0.
The watchpoints types (field WTYP) are:

Value	Watchpoint type
00	Disabled
01	Write watchpoint
10	Read watchpoint
11	Access watchpoint



BE Breakpoint 0 Enable.

After RESET, BE is cleared to 0.

WE Watchpoint 0 Enable.

After RESET, WE is cleared to 0.

Kalray – all rights reserved

WTYP Watchpoint 0 type.

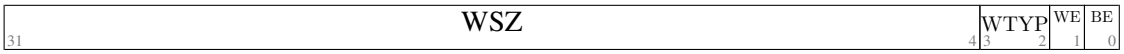
After RESET, WTYP is set to 0.

WSZ Watchpoint 0 size.

After RESET, WSZ is set to 0.

2.3.21 Debug Control (DC1)

This SFR controls the hardware watchpoint 1 and the hardware breakpoint 1.
The watchpoints types (field WTYP) are identical to those of DC0.



BE Breakpoint 1 Enable.

After RESET, BE is cleared to 0.

WE Watchpoint 1 Enable.

After RESET, WE is cleared to 0.

WTYP Watchpoint 1 type.

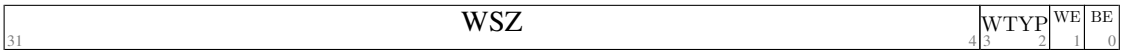
After RESET, WTYP is set to 0.

WSZ Watchpoint 1 size.

After RESET, WSZ is set to 0.

2.3.22 Debug Control (DC2)

This SFR controls the hardware watchpoint 2 and the hardware breakpoint 2.
The watchpoints types (field WTYP) are identical to those of DC0.



BE Breakpoint 2 Enable.

After RESET, BE is cleared to 0.

WE Watchpoint 2 Enable.

After RESET, WE is cleared to 0.

WTYP Watchpoint 2 type.

After RESET, WTYP is set to 0.

WSZ Watchpoint 2 size.

After RESET, WSZ is set to 0.

Kalray – all rights reserved

2.3.23 Debug Control (DC3)

This SFR controls the hardware watchpoint 3 and the hardware breakpoint 3.
The watchpoints types (field WTYP) are identical to those of DC0.

BE Breakpoint 3 Enable.

After RESET, BE is cleared to 0.

WE Watchpoint 3 Enable.

After RESET, WE is cleared to 0.

WTYP Watchpoint 3 type.

After RESET, WTYP is set to 0.

WSZ Watchpoint 3 size.

After RESET, WSZ is set to 0.

2.3.24 Debug Breakpoint Address (DBA*)

This SFR defines virtual PC to match for breakpoints.

DBA0 Virtual PC to match for breakpoint 0.

DBA1 Virtual PC to match for breakpoint 1.

DBA2 Virtual PC to match for breakpoint 2.

DBA3 Virtual PC to match for breakpoint 3.

2.3.25 Debug Watchpoint Address (DWA*)

This SFR defines virtual address to match for watchpoints.

DWA0 Virtual address to match for watchpoint 0.

DWA1 Virtual address to match for watchpoint 1.

DWA2 Virtual address to match for watchpoint 2.

DWA3 Virtual address to match for watchpoint 3.

Kalray – all rights reserved

2.3.26 Memory Error Status (MES)

This SFR defines Memory Error Status.

PSE SECC detected on memory answer to the instruction cache.

PILSY Program cache Invalidated Line following pSYs error.

PILDE Program cache Invalidated Line following pDEcc error.

PILPA Program cache Invalidated Line following pPArity error.

DSE SECC detected on memory answer to the data cache.

DILSY Data cache Invalidated Line following dSYs error.

DILDE Data cache Invalidated Line following dDEcc error.

DILPA Data cache Invalidated Line following dPArity error.

DDEE Data DEcc Error.

DSYE Data dSYs Error.

2.3.27 Wake-up Status (WS)

This register contains the value of the three wake-up bits (WU0, WU1 and WU2) that wake-up the core from / prevent the core to enter AWAIT, SLEEP and STOP idle mode respectively. Please refer to Section 3.4 for details about the idle modes and their entry/exit conditions.

2.3.28 Shadow Program Counter (SPC_PL*)

There is one SPC_PL$< i >$ per privilege level (i=[0,3]).

The Shadow Program Counter (SPC) is updated with the current PC value when a trap, syscall or interrupt is taken. It allows to update the PC in order to jump to the corresponding handler, while the source PC of the exception is clearly identified. Before clearing PS.ET, thus allowing the hardware to take new exceptions, SPC should be saved onto the stack.

At the end of the exception handler, the RFE instruction updates PC from SPC, allowing the preempted code to resume. If SPC has been saved onto the stack, then of course it is mandatory to restore SPC from the stack before executing the RFE instruction.

Kalray – all rights reserved

2.3.29 Shadow Processing Status (SPS_PL*)

There is one SPS_PL$< i >$ per privilege level (i=[0,3]).

The Shadow Processing Status (SPS) is updated with the current PS value when a trap, syscall or interrupt is taken. It allows to update the PS for the corresponding handler, while the original PS, corresponding to the state before the exception, is clearly identified. Before clearing PS.ET, thus allowing the hardware to take new exceptions, SPC should be saved onto the stack.

A the end of the exception or interrupt handler, the RFE instruction updates PS from SPS, allowing the preempted code to resume in the original state. If SPS has been saved onto the stack, then of course it is mandatory to restore SPS from the stack before executing the RFE instruction.

2.3.30 Excepting Address (EA_PL*)

There is one EA_PL$< i >$ per privilege level (i=[0,3]).

The Excepting Address (EA) register gives the effective address involving memory instructions which has caused the hardware trap. For other hardware traps, this register contains zero. It can be used in the exception handler to identify the origin of the hardware trap. EA is a read only register.

2.3.31 Exception Vector (EV_PL*)

There is one EV_PL$< i >$ per privilege level (i=[0,3]). Exceptions taken to PL$< j >$ make the core branch to:

- EV_PL$< j >$ + 0 for debug exceptions.
- EV_PL$< j >$ + 0x40 for trap exceptions.
- EV_PL$< j >$ + 0x80 for interrupt exceptions.
- EV_PL$< j >$ + 0xc0 for syscall exceptions.

The writable bits of EV are [40:8] (256B aligned), while bits [7:0] read as zero.

2.3.32 System-Reserved Registers (SR_PL*)

There is one SR_PL$< i >$ per privilege level (i=[0,3]).

The Kalray core offers one register for system per privilege level.

2.3.33 Debug Support Registers

Please refer to Section 5.4 (On Chip Emulation) section for details on the debug registers.

Kalray – all rights reserved

2.3.34 Exception Syndrome Register (ES_PL*)

ED (63..32)	
ED (31..4)	EC (3..0)

There is one ES_PL$< i >$ per privilege level (i=[0,3]).

The Exception Syndrome register is a SFR that is automatically updated by the hardware with status information when the Kalray kv3 core takes an exception or executes a bundle in step mode. The first field of ES, named **EC**, for **Exception Class**, can take the following possible values and associated meanings:

ES.EC Value	Exception Class
0b0000	Debug exception
0b0001	Hardware trap exception
0b0010	Interrupt exception
0b0011	Syscall exception
0b1000	Watch dog error
0b1001	Double exception happening because of hardware trap
0b1010	Double exception happening because of interrupt
0b1011	Double exception happening because of syscall

Codes 1001 to 1011 are used when a double exception fires. In that case, the hardware will fill the exception details of the second exception in ES.ED, as if this second exception had really fired (for debug purposes). Bit 3 of ES.EC is used to distinguish between regular exceptions and double exception traps. With respect to branching target based on EV_PL*, double exceptions are considered as hardware traps. Codes 0100, 0101, 0110 and 0111 are not set by hardware. They may be used by software, e.g. for virtualization purposes.

Each time ES.EC is updated by hardware with an exception class code, the rest of the SFR, named **ED** for **Exception Details**, is simultaneously updated with different pieces of information that are relevant with respect to that particular exception class. As a result, ES.ED will contain the concatenation of

- OAPL, ORPL, PTAPL, PTRPL, HTC, SFRT, SFRI, GPRP, SFRP, DHT, DRX, DAF, RWX, NTA, UCA, AS, BS, DRI and PIC when taking a hardware trap
- OAPL, ORPL, PTAPL, PTRPL, ITN, ITL and ITI when taking an interrupt
- OAPL, ORPL, PTAPL, PTRPL, SN when taking a syscall
- OAPL, ORPL, PTAPL, PTRPL, DC, BN, WN, SFRI, GPRP, SFRP, DHT, DRX, DAF, RWX, NTA, UCA, AS, BS, DRI and PIC when executing a system stepi

The meanings and encodings of these fields are detailed below, for each exception class.

Debug Exception fields in ES (EC: 0000):

Kalray – all rights reserved

©2025 Kalray

PIC	DRI	BS	AS	UCA	NTA	RWX	DAF	DRX	DHT	SFRP
63 60	59 54	53 50	49 44	43	42	41 39	38	37	36	35 32

SFRP	GPRP	SFRI	-	WBN	DCV2	PTRPL	PTAPL	ORPL	OAPL	EC
31 27	26 21	20 18	17	16 15	14 12	11 10	9 8	7 6	5 4	3 0

OAPL (Origin Absolute PL): Privilege level before trap

After RESET, OAPL is cleared.

ORPL (Origin Relative PL): Relative Privilege level before trap

After RESET, ORPL is cleared.

PTAPL (Primary Target Absolute PL): Normal absolute trap target level (no double trap).

After RESET, PTAPL is cleared.

PTRPL (Primary Target Relative PL): Normal relative trap target level (no double trap).

After RESET, PTRPL is cleared.

DCV2 (Debug Cause): code number of the debug exception taken by the core

Value	Debug Cause
000	Hardware breakpoint
001	Hardware watchpoint
010	Step
011	DSU break
100	Software breakpoint

WBN (Watchpoint / breakpoint Number): Number of the hardware watchpoint, hardware breakpoint or the software breakpoint that triggered the debug exception: 0 to 3 (significant only when DCV2 = 000, 001 and 100, if not 0)

SFRI (SFR Instruction): Encoding of the SFR access instruction

Value	Instruction
000	Not a BCU valid instruction: no SFR access instruction.
010	GET
011	IGET
100	SET
101	WFXL
110	WFXM
111	RSWAP

After RESET, SFRI is cleared.

GPRP (GPR Pointer): GPR index

After RESET, GPRP is cleared.

SFRP (SFR Pointer): SFR index

After RESET, SFRP is cleared.

Kalray – all rights reserved

DHT (Disabled Hardware Trap): Set when step over a disable Hardware Trap (significant only when DCV2 = 10, if not 0)

DRX (Data Register eXtension): DRI (Data Register Index) targets TCA register file.

DAF (Data Access Format): Set in case of lv transpose (lv.c*)

RWX (Read Write Execute):

Value	Interpretation
000	No data memory access in the stepped bundle (canceled non-trapping loads and non executed conditional ld/st are considered as such).
001	Reserved.
010	Data write memory access in the stepped bundle.
100	Data read memory access in the stepped bundle (DINVALL, DTOUCHL, DINVAL are considered as read accesses).
110	Data atomic memory access in the stepped bundle.
other	Reserved.

NTA Non-Trapping Access: if 1, the stepped bundle contains a speculative data memory access; if 0, the stepped bundle contains a regular data memory access (significant only when RWX = 010 or RWX = 100 or RWX = 110)

UCA (Un-Cached Access): if 1, the stepped bundle contains a *U LSU instruction; if 0, the stepped bundle contains a non *U LSU instruction (significant only when RWX = 010 or RWX = 100 or RWX = 110)

AS Access Size: size of the data access contained in the stepped bundle (significant only when RWX = 010 or RWX = 100 or RWX = 110)

Value	Interpretation
000001	Byte access.
000010	Half-word access.
000100	Word access (including atomic word size instructions).
001000	Double-word access (including atomic double-word size instructions).
010000	Quad-word access.
100000	Octo-word access.
111111	Cache line-size access (DZEROL).
100001	Cache maintenance access (DINVALL, DTOUCHL, DINVAL).
other	Reserved.

BS (Bundle Size): Size of the stepped bundle

Value	Interpretation
0000	Reserved.
0001	32 bits (1 syllable).
0010	64 bits (2 syllables).
0011	96 bits (3 syllables).
0100	128 bits (4 syllables).
0101	160 bits (5 syllables).
0110	192 bits (6 syllables).
0111	224 bits (7 syllables).
1000	256 bits (8 syllables).

Kalray – all rights reserved

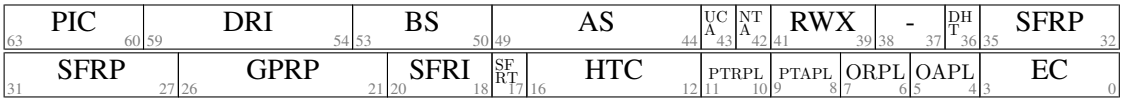
©2025 Kalray

DRI (Data Register Index): Index of the GPR holding the source or destination operand of the data memory access in the bundle (significant only when RWX = 010 or RWX = 100 or RWX = 110)

PIC (Privileged Instruction Code): Code of the privileged instruction of the stepped bundle

Value	Instruction
0000	No privileged instruction
0001	TLBWRITE
0010	TLBREAD
0011	TLBPROBE
0100	RFE
0101	STOP
0110	SYNCGROUP
0111	XSENDO or XRECVO or XSENDRECVO
1000	DINVALSW
1111	Multiple privilege instruction

Hardware traps fields in ES (EC: 0001):



OAPL (Origin Absolute PL): Privilege level before trap

After RESET, OAPL is cleared.

ORPL (Origin Relative PL): Relative Privilege level before trap

After RESET, ORPL is cleared.

PTAPL (Primary Target Absolute PL): Normal absolute trap target level (no double trap).

After RESET, PTAPL is cleared.

PTRPL (Primary Target Relative PL): Normal relative trap target level (no double trap).

After RESET, PTRPL is cleared.

HTC (Hardware Trap Cause): code number of the hardware trap taken by the core

Kalray – all rights reserved

Value	Hardware Cause
00000	Reserved
00001	OPCODE
00010	PRIVILEGE
00011	DMISALIGN
00100	PSYSERROR
00101	DSYSERROR
00110	PDECC
00111	DDECC
01000	PPAR
01001	DPAR
01010	PSECC
01011	DSECC
01100	NOMAPPING
01101	PROTECTION
01110	WRITETOCLEAN
01111	ATOMICTOCLEAN
10000	TPAR
10001	DOUBLE_EXCEPTION
10010	VSFR
10011	PL_OVERFLOW

SFRT (SFR Trap): Trap caused by a SFR access

After RESET, SFRT is cleared.

SFRI (SFR Instruction): Encoding of the SFR access instruction

Value	Instruction
000	Not a BCU valid instruction: no SFR access instruction.
010	GET
011	IGET
100	SET
101	WFXL
110	WFXM
111	RSWAP

After RESET, SFRI is cleared.

GPRP (GPR Pointer): GPR index

After RESET, GPRP is cleared.

SFRP (SFR Pointer): SFR index

After RESET, SFRP is cleared.

DHT (Disabled Hardware Trap): Set when taken a disable Hardware Trap

DRX (Data Register eXtension): DRI (Data Register Index) targets TCA register file.

DAF (Data Access Format): Set in case of lv transpose (lv.c*)

RWX (Read Write Execute):

Kalray – all rights reserved

Value	Interpretation
000	Not a memory trap.
001	Fetch side memory trap.
010	Data side memory trap on a write access (DZEROL is considered as a write access).
100	Data side memory trap on a read access (including DINVALL, DTOUCHL, DINVAL, WPURGE).
110	Data side memory trap on an atomic access.
other	Reserved.

NTA Non-Trapping Access: if 1, trap was caused by a non-trapping data memory access; if 0, trap was caused by a regular data memory access (significant only when RWX = 010 or RWX = 100 or RWX = 110)

UCA (Un-Cached Access): If 1, trap was caused by an uncached memory access instruction (significant only when RWX = 010 or RWX = 100 or RWX = 110)

AS Access Size: size of the data access that caused the trap (significant only when RWX = 010 or RWX = 100 or RWX = 110)

Value	Interpretation
000001	Byte access.
000010	Half-word access.
000100	Word access (including atomic word size instructions).
001000	Double-word access (including atomic double-word size instructions).
010000	Quad-word access.
100000	Octo-word access.
111111	Cache line-size access (DZEROL).
100001	Cache maintenance access (DINVALL, DTOUCHL, DINVAL).
other	Reserved.

BS (Bundle Size): Size of the bundle that caused the trap

Value	Interpretation
0000	Reserved.
0001	32 bits (1 syllable).
0010	64 bits (2 syllables).
0011	96 bits (3 syllables).
0100	128 bits (4 syllables).
0101	160 bits (5 syllables).
0110	192 bits (6 syllables).
0111	224 bits (7 syllables).
1000	256 bits (8 syllables).

DRI (Data Register Index): Index of the GPR holding the source or destination operand of the data memory access that caused the trap (significant only when RWX = 010 or RWX = 100 or RWX = 110)

PIC (Privileged Instruction Code): Code of the privileged instruction of the trapping instruction when $es.sfrt is set and $es.htc is PRIVILEGE. Else code of the privileged instruction of the trapped bundle.

Kalray – all rights reserved

Value	Instruction
0000	No privileged instruction
0001	TLBWRITE
0010	TLBREAD
0011	TLBPROBE
0100	RFE
0101	STOP
0110	SYNCGROUP
0111	XSENDO or XRECVO or XSENDRECVO
1000	DINVALSW
1111	Multiple privilege instruction (could not appear when $es.sfrt is set and $es.htc is PRIVILEGE)

Interrupts fields in ES (EC: 0010):

OAPL (Origin Absolute PL): Privilege level before trap

After RESET, OAPL is cleared.

ORPL (Origin Relative PL): Relative privilege level before trap

After RESET, ORPL is cleared.

PTAPL (Primary Target Absolute PL): Normal absolute trap target level (no double trap).

After RESET, PTAPL is cleared.

PTRPL (Primary Target Relative PL): Normal relative trap target level (no double trap).

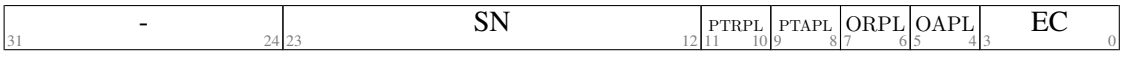
After RESET, PTRPL is cleared.

ITN (InTerrupt Number): number of the interrupt taken by the core

ITL (InTerrupt Level): level of the interrupt taken by the core

ITI (InTerrupt Info): copy of the MES SFR content when an interrupt related to a memory error is taken by the Kalray kv3 core (namely: IT nb 12 for SECCs, 16 for Data asynchronous memory errors, or 17 for i/d line invalidations)

System calls fields in ES (EC: 0011):



OAPL (Origin Absolute PL): Privilege level before trap

After RESET, OAPL is cleared.

ORPL (Origin Relative PL): Relative privilege level before trap

After RESET, ORPL is cleared.

Kalray – all rights reserved

PTAPL (Primary Target Absolute PL): Normal absolute trap target level (no double trap).

After RESET, PTAPL is cleared.

PTRPL (Primary Target Relative PL): Normal relative trap target level (no double trap).

After RESET, PTRPL is cleared.

SN (Syscall Number): copy of the `sysnumber` argument of the SCALL instructions

2.3.35 Write aliases to read only SFRs owner registers

SYOW SYscall Onwers Write: Alias to SYO. Used to write read only SYO according to SYO and current PL.

HTOW Hardware Trap Onwers Write: Alias to HTO. Used to write read only HTO according to HTO and current PL.

ITOW InTerrupt Onwers Write: Alias to ITO. Used to write read only ITO according to ITO and current PL.

DOW Debug Onwers Write: Alias to DO. Used to write read only DO according to DO and current PL.

MOW Miscellaneous Onwers Write: Alias to MO. Used to write read only MO according to MO and current PL.

PSOW Processor State Owners Write: Alias to PSO. Used to write read only PSO according to PSO and current PL.

2.3.36 Aliases to banked SFR registers

SPC Shadow Program Counter: Alias to SPC_PL[PS.PL].

SPS Shadow Processor State: Alias to SPS_PL[PS.PL].

EA Effective Address: Alias to EA_PL[PS.PL].

EV Exception Vector: Alias to EV_PL[PS.PL].

SR System Register: Alias to SR_PL[PS.PL].

ES Exception Syndrome: Alias to ES_PL[PS.PL].

Kalray – all rights reserved

3 System Control

This chapter gives an overview of the core system control, including privilege level transitions, and idle modes management.

3.1 Privilege Level Transitions

When an exception fires (respectively when a RFE instruction is executed), in addition to other actions, PS is saved to (respectively restored from) SPS_PL$< j >$, where $< j >$ is the destination level (respectively the origin level) of the exception (respectively RFE). This allows exception nesting across different PLs without loss of context. The case $j = i$ allows to define what is called "horizontal exceptions" (respectively "horizontal RFE") by contrast to the classic "vertical exceptions" (respectively "vertical RFE").

One motivation for the horizontal exceptions/RFE is the implementation of user-level interrupts, through the use of individual interrupts ownership defined in ITO.

Four categories of exceptions are defined: interrupts, hardware traps, system calls and debug. They are reported differently in ES_pl$< j >$ and branch to different addresses, all based on EV_PL$< j >$, where $< j >$ is the exception destination level.

When one of these exception fires, the execution is rerouted to the destination level $< j >$ (defined by the resource ownership, see Section 3.2) of the exception, and the following actions are taken automatically and atomically by hardware:

- PS $\leftarrow$ SPS_PL$< j >$ except for the PL field and the FOE0/1 bits[1] set by hardware[2].
- SPS_PL$< j > \leftarrow$ PS (atomic with the preceding step, this is really a swap).
- PC $\leftarrow$ handler PC based on EV_PL$< j >$ (see Section 2.3.31).
- SPC_PL$< j > \leftarrow$ PC (atomic with the preceding step, this is really a swap).
- ES_PL$< j > \leftarrow$ filled with relevant information by hardware.
- EA_PL$< j > \leftarrow$ filled with the exception address, as usual.

The PC saved in SPC is the PC of the first non-executed bundle before the exception is taken. In case of system call and interrupt, this is the PC of the logical successor of the bundle executing while exception was accepted. In case of hardware trap and debug exceptions, this is the PC of the bundle that raised the exception.

When a RFE instruction is executed from PL$< i >$, the following actions are taken automatically and atomically by hardware:

- PS $\leftarrow$ SPS_PL$< i >$.

[1] FOE0/1 means Forced on Exception to 0 or 1, such as PS.IE and PS.ET.

[2] This allows to (partly) restore a remnant state of a destination level when entering it (e.g. caches enabled in OS independently from the setting in User, or 32/64-bit modes set differently). To avoid unnecessary software manipulations, this partial restoration is not performed in case $< i >=< j >$ (horizontal exception).

Kalray – all rights reserved

- SPS_PL$< i >$ $\leftarrow$ PS except for the FOE0/1 bits, SPS_PL$< i >$.IL and SPS_PL$< i >$.PL fields that are left unchanged (back-up of the remnant state of PL$< i >$, atomic with the preceding step, this is really a swap)[3].
- PC $\leftarrow$ SPC_PL$< i >$.

For some hardware traps and debug exceptions, resuming execution should be at the bundle that logically follows the one that triggered the exception, instead of the trapping bundle itself. This is achieved by explicitly increasing the SPC saved by exception taking by the size of the trapping bundle, before executing RFE. To support this idiom, instructions that may raise a PRIVILEGE trap (SET, WFX*, RSWAP, TLB*) have a bundling constraint that ensures that they are alone in their bundle, so the size of their bundle is always 4.

3.2 Privilege Level Ownership

Owner PLs are defined for various resources of the processor: instructions, SFRs or their bit-fields, and exceptions. After reset, PL0 is the owner of all the resources of the processor. The ownership of each owner-controllable resource is defined in one of the six SFRs: SYO, HTO, ITO, DO, MO PSO. When a PL owns a resource/mechanism, it means that:

- the owner PL and more privileged PLs can use/read/modify the resource (if applicable);
- less privileged PLs cannot use/read/modify the resource and if trying they will:
  - trap (PRIVILEGE) to the owner PL if trying to do a SFR write (SET, WFX*, RSWAP, TLB*) on a SFR or SFR field belonging to a more privileged PL.
  - read zeroes (RAZ) in the relevant fields (non owned fields) if trying to read a SFR (GET, IGET), except for:
    - PCR: read-only for all non privileged PLs (PLs that are less privileged than the owner of PCR).
    - PS: all fields are read-only for non privileged PLs, even ones not owned by the current PL in PSO.
    - SPS_PL$< j >$: all fields are read-only for PL$< i >$, with $i \leq j$, even ones not owned by PL$< i >$ in PSO.
    - SYO, HTO, ITO, DO, MO and PSO: read-only for all non privileged PLs.
  - trap (PRIVILEGE) to the owner PL if trying to execute an instruction belonging to a more privileged PL.

In case of simultaneous OPCODE and PRIVILEGE traps (because the targeted SFR/SFR field cannot be accessed with the instruction used and it belongs to a more privileged PL), the OPCODE trap is taken (e.g. if PL$< i >$ tries to execute WFXL on EA_PL$< j >$ with $i > j$).

For the exceptions, the notion of ownership (specified by SYO/HTO/IT/DO) means that, when the core is executing in PL$< i >$ and encounters an exception that belongs to PL$< j >$, then:

[3] Similarly to exception taking, this is not performed for horizontal RFE.

Kalray – all rights reserved

- if $j \leq i$ (owner more privileged or equal to current PL, normal case), it will take that exception to PL$< j >$;
- else ($j > i$, owner less privileged than current PL, pathological case), it will take that exception to PL$< i >$ (itself).

3.3 Privilege Level Manipulations

When setting an ownership, the value used by the WFX*/SET instruction will always be relative to the PL currently executing. This means that writing 0 signifies 'myself (+0), the current PL', 1 really means 'myself + 1', and so on until +3. So, for example, if the processor is running in PL1 and writes 2 to an owner field (e.g. SYOW.QO), it means: 'assign ownership of Q0 to PL3'.

When reading from ownership registers, the value returned by the HW will be absolute, e.g. 2 means PL2, whatever the current PL.

To help distinguish the two, aliases have been defined for the SYO, HTO, ITO, DO, MO PSO (called *O SFRs) by appending 'W' to their name (called *OW SFRs). The *O SFRs SFRs will always be used for reads (GET, IGET) and return an absolute encoding of owner PLs, while *OW SFRs will be used for writes (SET, WFX*). The *O SFRs are read-only for all levels and the *OW SFRs are WFX-able field by field for the owner of the fields (and cannot be read). RSWAP is forbidden and will always trap OPCODE if tried on a *O or *OW SFR as it is considered as a read and a write.

This aliasing of *O SFRs answers the needs to:

- know if we are the owner of a resource or it is owned by a more privileged PL than us. This is made possible by reading *O and comparing it with PS.PL;
- delegate ownership of a resource to less privileged PL. Doing so in a relative way:
  - Gives a more robust and intuitive way to manipulate permissions: cannot try to delegate to a more privileged PL.
  - Eases the porting of a given software layer (e.g. an OS) to a different PL than that for which it was originally written (opening the door to dynamic level relocation).

As explained above, relative PLs are used when writing to *O ownership SFRs through their *OW aliases. They are also used when writing to the PS.PL field (but not reading; reading PS.PL will output an absolute PL value).

The WFX* instructions targeting relative PL fields in the SFRs mentioned above use the following semantics:

Write Mask	Read Mask	Effect
00	00	None
00	01	+0
00	10	+0
00	11	+0
01	xx	+1
10	xx	+2
11	xx	+3

Kalray – all rights reserved

So, for example, the following code sequence:

```
make $r0 = 0x0000000100000300
;;
wfxl $itow = $r0
;;
```

This attempts to attribute ownership (current level) +1 to IT0, and self ownership (+0 = current level) to IT4, leaving the other IT ownership's untouched. That may in turn trigger a privilege or PL overflow trap (explained below), depending on the current value of ITO and the current PL.

When software writes a relative PL value to *OW or PS.PL, it may be subject to 'PL Overflow'. This is a hardware trap (whose owner is itself defined in HTO) that will fire if the sum of the current PL and the relative PL value it tries to write exceeds 3, as the hardware is in the incapacity of turning the relative information into a valid absolute one. For example, if PL1 is the owner of ITO.IT_12 and it tries to delegate it to PL +3 (by writing 3 in ITOW.it_12) then, the write will not be made and the PLO trap will fire (unless, of course, if PL1 was not the owner (or more privileged) of IT_12 to start with, in which case, its write attempt is not even examined and the PRIVILEGE trap is raised).

In SPS*.PL, the PL value is stored under its relative form. When populated by the hardware upon exception entry, it means 'this is the PL I took the exception in, relative to the target PL'. As a result, when a trap handler (e.g.) executing in PL$< i >$ reads SPS, it can know where the exception originated from: $< i >$ (= itself, meaning horizontal exception) if SPS.PL = 0, $< i+1 >$ if SPS.PL = 1, etc. The relative aspect of the information is not dynamic, it is built once and for all when SPS.PL is updated (be it by the hardware or by a SET/WFX*). Consequently, reading (e.g.) SPS_PL2.PL as '01' while executing in PL0 means PL2 + 1 = PL3, and not PL1.

When writing to SPS*.PL, the hardware does not apply any particular rules to build the new value (contrary to those explained above for *OW fields). In case of WFX*, it simply applies the clear/set masks to the former value, as it does for any regular SFR field. It is acceptable for the software to write a relative value that may lead to PL overflow. As the check will only be made by the HW when RFE is executed. It is at this point that the relative information stored into SPS*.PL will be turned into an absolute one in PS.PL, doing PS.PL = current_PL + SPS*.PL.

The general rule is that the hardware enforces PL overflow check only at the time a relative PL information is turned into an absolute one. This happens when:

- Writing to *O registers through their *OW aliases, as ownerships live in absolute mode in the real *O registers.
- Writing to PS.PL, as the real PS.PL register contains an absolute PL.
- Executing RFE, as the real PS.PL register contains an absolute PL.

The only places where a relative PL can be found in a permanent way (that is, really stored in the hardware, not just used as a way to convey information during the execution of a SET/WFX*) are SPS*.PL and ES.*RPL.

Kalray – all rights reserved

3.4 Idle Modes and Transitions

The core offers three different idle modes where it stops executing, waiting for some conditions to resume execution. They are named after the instructions that are used to enter them: AWAIT, SLEEP and STOP (formerly known as, respectively, IDLE0, IDLE1 and IDLE2).

These three idle modes all save power by gating the main CPU clocks. They differ by the conditions upon which they are entered/left by the core, and by their sensitivity to the reset_on_wakeup cluster power controller bit.

The entry/exit in/from each mode is controlled by an associated bit in the Wakeup Status SFR (WS). Precisely, **AWAIT mode** is entered by the core when it executes the AWAIT instruction and WS.WU0 = 0. Executing AWAIT with WS.WU0 = 1 is equivalent to a NOP (no stall happens).

When in AWAIT idle mode, the core will wake-up and resume execution as soon as WS.WU0 rises. WS.WU0 is automatically set to 1 by hardware when one of the following conditions happens:

- A PE2PE event comes in
- Any new interrupt comes in (rising edge), including interrupts that can not fire immediately because they are individually disabled, or their level is less or equal to PS.IL, or interrupts are globally disabled (PS.IE = 0)
- The core receives a wake-up signal from the cluster power controller or the DSU.

If woken up from AWAIT by en eligible interrupt, the core will take it straight away, setting SPC to the bundle following the AWAIT instruction. Otherwise, a core woken up from AWAIT will simply resume execution at the bundle following the AWAIT instruction.

Similarly, **SLEEP mode** is entered by the core when it executes the SLEEP instruction and WS.WU1 = 0. Executing SLEEP with WS.WU1 = 1 is equivalent to a NOP (no stall happens).

When in SLEEP idle mode, the core will wake-up and resume execution as soon as WS.WU1 rises. WS.WU1 is automatically set to 1 by hardware when one of the following conditions happens:

- A new eligible interrupt comes in.
- The core receives a wake-up signal from the cluster power controller or the DSU.

If woken up from SLEEP by en eligible interrupt, the core will take it straight away, setting SPC to the bundle following the SLEEP instruction. Otherwise, a core woken up from SLEEP will simply resume execution at the bundle following the SLEEP instruction.

Finally, **STOP mode** is entered by the core when it executes the STOP instruction and WS.WU2 = 0. Executing STOP with WS.WU2 = 1 is equivalent to a NOP (no stall happens). When in STOP idle mode, the core will wake-up and resume execution as soon as WS.WU2 rises. WS.WU2 is automatically set to 1 by hardware when the core receives a wake-up signal from the cluster power controller or the DSU. When woken up from STOP by such a signal, the core will:

Kalray – all rights reserved

- hardware reset itself and reboot from RESET_PC if the corresponding reset_on_wakeup bit of the cluster power controller is set
- resume execution at the bundle following the STOP instruction otherwise

STOP is the only idle mode that is sensitive to reset_on_wakeup, AWAIT and SLEEP always ignoring it (acting as if it were 0).

Events and interrupts received during any idle mode are always registered in EV4 and ILR SFRs, whether they wake the core up or not. Similarly, WS.WU* bits are raised by the events detailed above and registered, whether they wake the core up or not.

When software wishes to enter an idle mode to save power while waiting for certain conditions, it is recommended to follow the sequence below:

1. First blindly clear the relevant WS.WU* bit.
2. Then check the appropriate software structures/conditions indicating whether there are tasks to perform.
3. If yes, treat them and then go back to (1).
4. Else, execute the needed AWAIT, SLEEP, STOP instruction.

In that way, if there is a race between step (2) and a new event/interrupt signaling that there is work to be done, the core will either go through those idle modes as a NOP, or be woken up quickly after entering it. This avoids scenarios where an idle mode is entered and the core never wakes up because the wake-up event was cleared by mistake before.

Kalray – all rights reserved

4 Memory Management Unit

4.1 MMU Overview

4.1.1 MMU Main Features

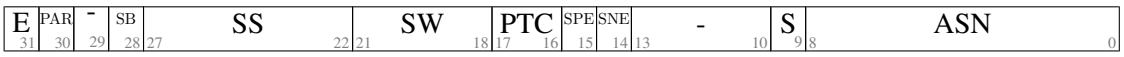
The Kalray kv3 core Memory Management Unit (MMU) is enabled by setting the MME bit of the Processing Status (PS) register. The Kalray kv3 core MMU assumes a unified address space for instructions and data. It supports three primary functions:

- Perform address translation from 32-bit (41-bit in 64 bit addressing mode) virtual addresses, optionally associated to a 9-bit Address Space Number (ASN), to 40-bit physical addresses.
- Ensure memory protection between processes, by associating protection attributes with each virtual memory page.
- Enforce memory caching policies for each virtual memory page.

The overall virtual to physical address translation flow is depicted in Figure 5.

Operating systems rely on a page table data structure to translate the virtual addresses of a process to physical addresses. The Kalray kv3 core includes Translation Lookaside Buffers (TLBs) to cache the most recent translations. The Kalray kv3 core MMU is designed for software page table walk and TLB contents maintenance. The Kalray kv3 core MMU supports 4 page sizes, namely 4KB, 64KB, 2MB and 512 MB. For each TLB entry, the (virtual and physical) addresses must be aligned on the page size (least significant bits of PN and FN will be considered as zero by hardware).

4.1.2 Memory Management Control (MMC) Register



The MMC register is a 32-bit system register that configures the MMU:

E Error flag

Set by TLB maintenance instructions under certain conditions (see Section 4.3).

PAR PARity error flag

Set by TLB maintenance instructions under certain conditions (see Section 4.3).

SB Select Buffer

Buffer (0 for JTLB, 1 for LTLB) of the entry operated on by the TLB maintenance instructions.

Kalray – all rights reserved

©2025 Kalray

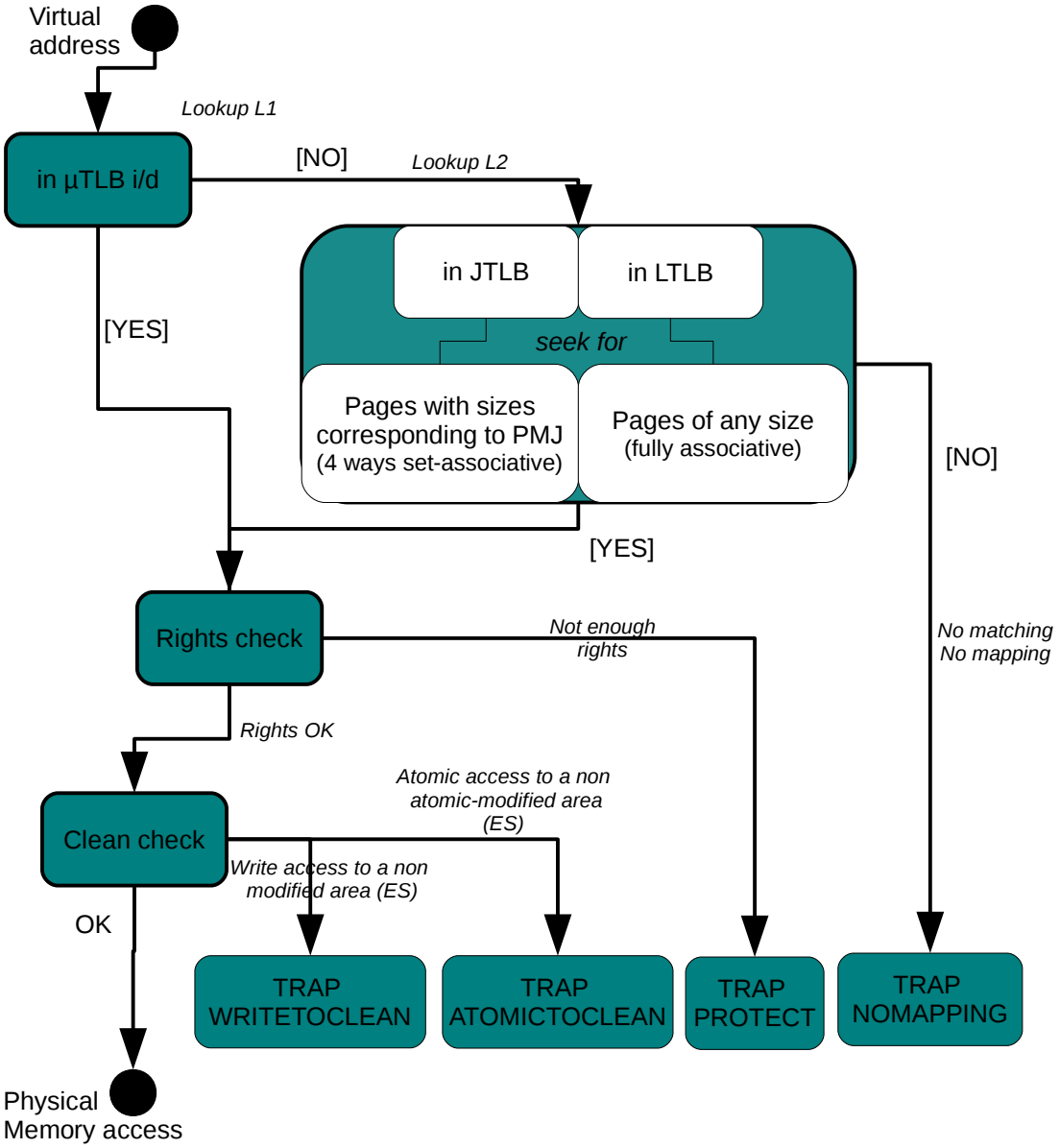


Figure 5: MMU address translation flow.

Kalray – all rights reserved

SS Select Set

Set number of the entry operated on by the TLB maintenance instructions (0 to 63 for the JTLB, 0 for the LTLB).

SW Select Way

Way number of the entry operated on by the TLB maintenance instructions (0 to 3 for the JTLB, 0 to 15 for the LTLB).

PTC Protection Trap Cause

This field is updated by the HW when a PROTECTION trap is taken and details the scenario that produced the trap, i.e. it precises the nature of the memory access that triggered the PROTECTION trap. Its value is interpreted as follows:

PTC Value	0	1	2	3
Trapping access nature	reserved	read	write	execute

where "execute" corresponds to an instruction fetch access, "read" to a data read instruction (loads, DINVALL, DTOUCHL), and "write" to a data write access (stores, DZEROL and atomic instructions).

SPE Speculative PROTECTION trap Enable

Enable speculative memory instructions to trigger the PROTECTION trap.

SNE Speculative NOMAPPING trap Enable

Enable speculative memory instructions to trigger the NOMAPPING trap.

S Speculative trap

Automatically updated by hardware on PROTECTION and NOMAPPING traps: set to 1 if the trap was caused by a speculative memory instruction (non-trapping load or DTOUCHL), 0 otherwise.

ASN Address Space Number

Contains the Address Space Number of the current process running on the Kalray kv3 core. Non global TLB entries (i.e. entries with bit G = 0) must have their ASN field matching MMC.ASN for a (data or instruction) core memory access to match them (in addition to the virtual address matching the PN field of the entry).

4.1.3 TLB Entry Low (TEL) and High (TEH) Registers

The TEL and TEH registers are 64-bit system registers whose formats correspond respectively to the High part [80:40] and the Low part [39:0] of the TLB Format (TLBF) specified in Section 4.2. The high parts of TEL and TEH (respectively bits [63:40] and [63:41]) are reserved: writing to them has no effect, they are always read as 0 and have no effect on maintenance instructions. TLBF represents the format of the TEL:TEH register pair (concatenated without the reserved high parts). These registers are used to:

Kalray – all rights reserved

- Prepare an entry for writing into the architectural TLBs with the TLBWRITE instruction.
- Return an entry read from the architectural TLBs with the TLBREAD or TLBPROBE instructions.
- Supply the virtual address and ASN to the TLBPROBE instruction.
- Return the slot ($\{buffer, set, way\}$ triplet) in which a TLB entry hit by a TLBPROBE instruction resides.

4.1.4 MMU Trap Causes

The traps that can be raised by the MMU correspond to the following PS.EC values:

PSYSERROR The mmu is off (PS.MME = 0) and the (fetched) virtual address is out of the physical address range (0 to 1TB -1) so it cannot be flat-mapped

DSYSERROR The mmu is off (PS.MME = 0) and the (data) virtual address is out of the physical address range (0 to 1TB -1) so it cannot be flat-mapped

TPAR Parity error encountered while performing a lookup in JTLB. This trap will not be raised if a matching entry is found in the LTLB.

NOMAPPING No valid architectural TLB entry matches the virtual address (TLB miss).

PROTECTION A valid architectural TLB entry matches the virtual address, however the protection attributes do not allow the memory access.

WRITETOCLEAN A valid architectural TLB entry matches the virtual address, the protection attributes authorize the memory store/DZEROL/atomic instruction execution, however the entry ES value is “Present” (CP = uncached).

ATOMICTOCLEAN A valid architectural TLB entry matches the virtual address, the protection attributes authorize the memory atomic instruction execution (alwc, aldc, acws, acds, afwa or afda), however the entry ES value is “Modified” (CP = 10).

DMISALIGN Data misaligned access to a device area (CP = 00) or data misaligned access spanning over two pages with different cache policies

The MMU traps are prioritized according to the above order with PSYSERROR first.

For all MMU traps, the virtual address is available in the EA register. In case an unaligned bundle, or data access, spans over two pages, EA contains:

- the (virtual) address of the beginning of the bundle (respectively the data access effective address) if the lower page traps,
- the (virtual) address of the higher page otherwise.

In case a misaligned bundle, or data access, spans over two pages and the two pages are trapping, the page holding the lower virtual address imposes its trap.

Kalray – all rights reserved

4.2 Translation Lookaside Buffers (TLBs)

4.2.1 Architectural and Micro TLBs

The Kalray kv3 core architectural resources include a fully associative locked TLB (LTLB) with 16 entries, and a 4-way set-associative joined TLB (JTLB) with 256 entries (64 entries per way). The union of the LTLB and the JTLB is referred to as the architectural TLBs (see Figure 6). Each TLB entry contains 81 bits, respecting the format detailed below.

In order to improve the energy efficiency and the performance of the core, two fully associative caches of the architectural TLBs of 4 entries each are implemented, one for data accesses (μDTLB) and the other for instructions fetches (μITLB). Indeed, the entries are then closer to the translation process. These micro-architectural TLBs (referred to as micro-TLBs) are managed by hardware and are transparent to software. They are automatically invalidated whenever any LTLB or JTLB entry is changed. When a data or instruction memory access is requested by the core, the μDTLB and the μITLB are respectively looked up. If no matching entry is found (micro-TLB miss), the architectural TLBs are sought.

TLB entries should be installed by software in the architectural TLBs using TLBWRITE instructions. When writing TLB entries into the architectural TLBs, the software only controls:

- in which TLB (LTLB or JTLB) the entry is written, via MMC.SB (Select Buffer)
- in which way of a particular TLB the entry is written, via MMC.SW (Select Way)

MMC.SS (Select Set) is used only by TLBREAD and is ignored by TLBWRITE. As a consequence, the hardware will always decide unilaterally in which set of the target TLB an entry goes. This will be set 0 for the LTLB as it is the only one available, and set $< i >$ in JTLB, where $< i >$ corresponds to the "index" of the virtual page of the entry. For a page of size 2^n (with $n \geq 12$), this index is determined by the value of the bit slice comprising bits n to n+5 of the virtual address corresponding to the start of the page (that corresponds in turn to the bits n-12 to n-12+5 of the entry's TLBF.PN field). For example, a 4KB page that has to be installed in the JTLB will be written to set number p, where p is the value of the 6 lower significant bits of the entry's TLBF.PN field (bits 12 to 17 of the virtual address corresponding to the start of the page). As a result, a 4KB page starting at address 0x00001000 will be written to the set number 1 in the JTLB (the way number (0 to 3) being specified by software through MMC.SW). A 64KB page that has to be installed in the JTLB will be written to architectural set number p, where p is the value of the bits 16 to 21 of the entry's TLBF.PN field. As a result, a 64KB page starting at address 0x001e0000 will be written to the set number 30 in the JTLB.

When a TLBWRITE is successful, the hardware updates MMC.SS with the number of the set in which it installed the TLB entry.

This organization constraint is in relation with the way the JTLB is implemented in hardware: it is really a 4-ways set associative cache (refilled by software), meaning that the hardware will look up virtual addresses at precise indexes (set number) only. This in turn allows a fast lookup of the JTLB, typically spending one cycle by page size enabled in PS.PMJ.

The Kalray kv3 core does not support virtual pages overlapping or multimapping: in case several entries are written to the architectural TLBs that could match the same memory access, the behaviour is undefined.

Kalray – all rights reserved

©2025 Kalray

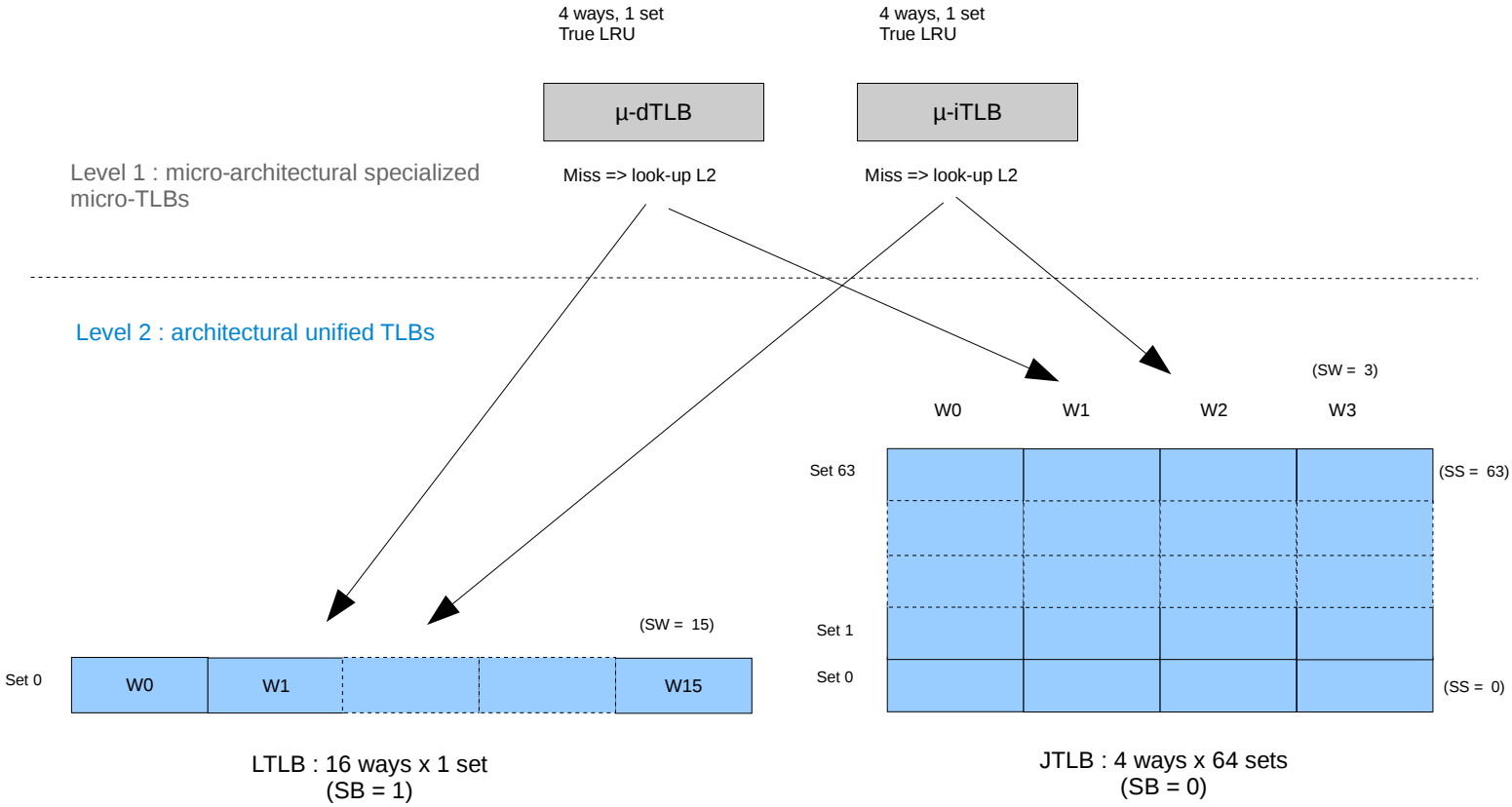
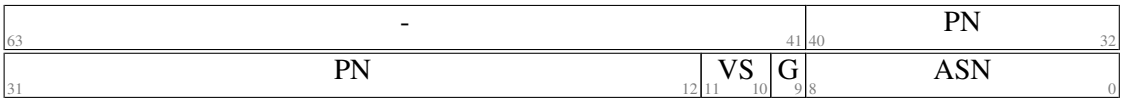


Figure 6: Architectural and microTLBs organization.

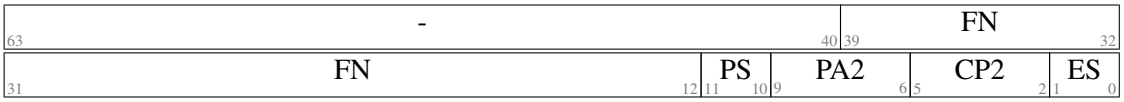
Kalray – all rights reserved

4.2.2 TLB Format (TLBF)

TEH:



TEL:



The Kalray kv3 core defines the format of the 81 bits of a TLB entry as:

PN Page Number

These bits are matched against the virtual address bits [40:n], where n is such as 2^n equals the page size. PN need to be always aligned on page size. For page sizes greater than 4KB, bits $[n-12-1{:}0]$ of PN must be 0. The nine highest bits of PN (corresponding to bits [40:32] of TEH) must always match the virtual address bits [40:32], even when the core executes in 32 bit addressing mode. This implies that they should be written as 0 if willing to map pages for 32 bit addressing mode.

VS Virtual Space

One virtual space per privilege level. Virtual space is used by hardware during lookup of the MMU TLB entries for normal fetch and data virtual accesses.

G Global page indicator

When G = 1, the page is global, meaning that its ASN need not match MMC.ASN for a virtual address to hit in the page

ASN Address Space Number

Address Space Number of the page: must match MMC.ASN for an access to hit in the page (unless G = 1).

FN Frame Number

FN contains the physical address that will be substituted to the page address when a virtual address translation occurs. Only bits [39:n], where n is such as 2^n equals the page size, will be substituted. The bottom bits of the virtual address (bits $[n-1{:}0]$) will remain unchanged and constitute the offset inside the page. For page sizes greater than 4KB, bits $[n-12-1{:}0]$ of FN are always ignored (considered as 0) by hardware. This corresponds to the fact that virtual pages (and their associated physical frames) are always aligned on the page size.

Kalray – all rights reserved

PS Page Size

This field sets the page size among the four available, with the following encoding:

PS Value	0	1	2	3
Page Size	4KB	64KB	2MB	512MB

PA2 Protection Attributes

The sixteen values encoded in PA2 are interpreted as different access protections depending on the PS.MMUP Privilege Mode bit:

PA2 Value	0	1	2	3	4	5	6	7
PS.MMUP = 0	NA	NA	NA	NA	NA	R	R	R
PS.MMUP = 1	NA	R	RW	RX	RWX	R	RW	RX
PA2 Value	8	9	10	11	12	13	14	15
PS.MMUP = 0	R	RW	RW	RX	RX	RWX		
PS.MMUP = 1	RWX	RW	RWX	RX	RWX	RWX		

Access protections include No Access (NA), Read Only (RO), Read Write (RW), Read Execute (RX), Read Write execute (RWX).

CP2 Cache Policy

The values encoded in CP2 (= Cache Policy) are interpreted as follows:

CP2	**Type**	**L2 policy (I + D)**	**L1D policy**	**L1I policy**
0000	Memory	L2 uncached	L1D uncached	L1I uncached
0001	Memory	L2 uncached	L1D uncached	L1I cached
0010	Memory	L2 uncached	L1D cached	L1I uncached
0011	Memory	L2 uncached	L1D cached	L1I cached
0100	Memory	L2 cached	L1D uncached	L1I uncached
0101	Memory	L2 cached	L1D uncached	L1I cached
0110	Memory	L2 cached	L1D cached	L1I uncached
0111	Memory	L2 cached	L1D cached	L1I cached
1000	Device	L2 uncached	L1D uncached	L1I uncached
other	Reserved	unpredictable	unpredictable	unpredictable

The Uncached policy connects loads, stores and fetch accesses directly to the memory system, without side-effects on the caches.

The Device policy prevents the speculative memory instructions (non-trapping load or DTOUCHL) from accessing memory. Non-trapping loads happening in Device pages always return zero to the core. In addition, the Device policy implies that all accesses must be aligned. In case of misaligned memory access with Device policy, a DMISALIGN trap is generated. An access spanning over two pages with different CP also generate a DMISALIGN trap.

ES Entry Status

The four values encoded in ES are interpreted as follows:

ES Value	0	1	2	3
	Invalid	Present	Modified	A-Modified

Kalray – all rights reserved

A zero ES value disables the entry. Setting ES value to “Modified” prevents the WRITETOCLEAN trap from happening when a store/DZEROl/atomic instruction writes data to this page (but atomic instructions hitting such pages will take the ATOMICTOCLEAN trap). Setting ES value to “A-Modified” prevents the WRITETOCLEAN trap from happening when a store/DZEROl instruction writes data to this page, and also prevents the ATOMICTOCLEAN trap from happening when an atomic instruction accesses this page (as a result no *TOCLEAN traps can be taken on such a page).

When TLB entries are stored in memory as part of a page table, the ES value zero (Invalid) frees the 30 upper bits of the corresponding TLB entry. These bits can be used by system software to encode a page offset in a swap file. A non-zero ES value indicates the translation is valid.

4.2.3 Configuration of supported Page Sizes

All page sizes can be installed and used at any time in the LTLB. On the other hand, PS.PMJ (Page size Mask in JTLB) constrains the page sizes that can be used currently in the JTLB, as described in section 4.1.2. All page sizes can be installed in the JTLB even a page whose size is not enabled in PS.PMJ (SPS.PMJ if PS.DAUS is set). Disabling page sizes in PS.PMJ without first invalidating all valid entries of the corresponding sizes in the JTLB is forbidden and may lead to undefined behaviour.

Note The reset value of the LTLB entries is all 0 (so, in particular, TLBF.ES = 0, meaning the page is not valid). The reset value of the JTLB entries is architecturally undefined. As a consequence, all JTLB entries should always be written by software (using the TLBWRITE maintenance instruction) before the MMU is enabled for the first time.

4.3 TLB Maintenance Instructions

4.3.1 Writing to Architectural TLB Entries

The TLBWRITE instruction reads the TEL:TEH register pair as well as the MMC.SB and MMC.SW fields and copies TEL:TEH in the TLB entry slot of the selected buffer (JTLB if MMC.SB = 0, LTLB if MMC.SB = 1), at the correct set determined by hardware, in the way selected by MMC.SW, as explained in section 4.2.1. Error conditions of the TLBWRITE instruction are:

- A MMC.SW value out of the architectural TLB ranges (i.e.: ≥ 4 with MMC.SB = 0)
- TEH.PN or TEL.FN values that are not aligned on TEL.PS

On error, the MMC.E bit is set, MMC.SS is not updated and the write to the architectural TLB is not done. If the TLBWRITE is successful, MMC.E is cleared and MMC.SS is updated with the set number in which the hardware wrote the TLB entry.

Kalray – all rights reserved

4.3.2 Reading from Architectural TLB Entries

The TLBREAD instruction reads the MMC.SB, MMC.SS and MMC.SW fields. The architectural TLB entry corresponding to this triplet is copied into the TEL:TEH register pair. For hardware optimization reasons, following a successful TLBREAD, only bits [40-12:n-12] of the TEH.PN field (corresponding to bits [40:n] of TEH) are guaranteed to be correct (i.e. reflect those of the entry read), with n such as 2^n equals the page size of the entry read. Bits [n-12-1:0] of TEH.PN are undefined and should not be interpreted / relied upon. So, for example, for a 64KB page, bits [3:0] of PN (corresponding to bits [15:12] of TEH) are undefined. These bits are not significant (as their weight is inferior to the entry's page size) and never used by hardware during the memory access matching process.

In case

- the triplet does not point to an existing TLB entry (e.g.: $\{SB, SS, SW\} = \{1, 53, 4\} = \{LTLB, set53, way4\}$)
- or the TLBREAD reads a JTLB entry and a parity error is detected somewhere in the corresponding set

then the MMC.E bit is set and the read is not performed (TEL:TEH are not updated). MMC.PAR is cleared in the first case (TLBREAD to a non existing entry) and set in the second one (parity error in JTLB).

If the TLBREAD is successful (that is none of the above error conditions happens), then MMC.E and MMC.PAR are cleared and TEL:TEH are updated with the value of the TLB entry read.

4.3.3 Probing Architectural TLB Entries

The TLBPROBE instruction reads the TEH.PN and MMC.ASN fields, as well as PS.VS (SPS.VS if PS.DAUS is set) and searches for a matching entry in the architectural TLBs. On success, the $\{SB, SS, SW\}$ triplet of the matching entry is written to the MMC.SB, MMC.SS and MMC.SW fields, the matched entry is copied into TEL and TEH, and MMC.E and MMC.PAR are cleared.

A matching entry in the architectural TLBs must be valid, must have the same VS and PN field values, and the same ASN value (unless it is global).

Error cases for a TLBPROBE instruction are:

- A parity error is encountered in JTLB during the lookup and no LTLB entry matches. In this case MMC.SB, MMC.SS and MMC.SW fields are not updated (nor TEL and TEH) and the MMC.E and MMC.PAR bits are set.
- no matching entry is found in the architectural TLBs (and no parity error is detected). In this case, MMC.SB, MMC.SS and MMC.SW fields are not updated (nor TEL and TEH) , MMC.E is set and MMC.PAR is cleared.

4.4 Special Memory Accesses

Kalray – all rights reserved

4.4.1 Speculative Memory Instructions

Speculative memory instructions (such as non trapping loads and DTOUCHL) can not trigger DSYSERROR nor DMISALIGN traps. In case one of them is performed at a virtual address where a regular load would have triggered such traps, it is cancelled (not sent to the memory system) and silently brings back 0x0 to the register file.

Speculative memory instructions are subject to the TPAR trap like any other access.

Speculative memory instructions can not trigger the WRITETOCLEAN trap nor the ATOMICTOCLEAN trap, as only memory instructions that write data to memory can trigger these traps.

Speculative memory instructions can trigger NOMAPPING and PROTECTION traps if the bits MMC.SNE and MMC.SPE are respectively set. If a speculative memory access is performed at a virtual addresses where a regular load would have triggered one of these traps, and the corresponding control bit of MMC is cleared, the access is cancelled (not sent to the memory system) and silently brings back 0x0 to the register file.

4.4.2 Unaligned Accesses

Unaligned memory accesses that overlap a boundary between two pages have the potential to trigger two MMU traps. When this condition occurs, the two traps are serialized: the first is triggered immediately, and the second is triggered when the memory access instruction is restarted (with the page corresponding to the first half newly mapped).

As a consequence, on traps caused by the MMU on a (respectively) data access or instruction bundle spanning over two pages, the EA SFR (Exception Address) will be set to the effective address of the access (respectively bundle) if the first page is not mapped or generates trapping conditions, and to the address of the start of the upper page otherwise. For fetch-side MMU traps, SPC will always be set to the address of the start of the bundle.

Because, when the MMU is on, the NOMAPPING trap has the highest priority (apart from the TPAR trap), the two potentially missing pages will be mapped first before any other MMU trap occurs on a misaligned access.

When a data access spans over two pages that are valid but specify different cache policies, a DMISALIGN trap is generated (unless the access is speculative). In this case, EA is set to the address of the upper page (unless the bottom page uses a device cache policy, in which case, EA is set to the effective address of the access).

4.4.3 Global Cache maintenance instructions

Global data cache maintenance instructions (DINVAL) and global instruction cache maintenance instructions (IINVAL) bypass the MMU as they do not target particular addresses. Hence no translation is needed for them, and no right checks of any kind is made.

4.4.4 Line Cache maintenance instructions

The IINVALS instruction acts on indexes of the instruction cache (not real addresses) by invalidating lines whose indexes match the lowest bits of the virtual address provided. As a result, it

Kalray – all rights reserved

does not need any translation and also bypasses the MMU.

DTOUCHL and DINVALL are considered respectively as non trapping and regular load byte with respect to all the MMU behaviour (translation, permission checks, traps).

DZEROL is considered as a write access (store byte) with respect to all the MMU behaviour (translation, permission checks, traps).

4.4.5 Atomic instructions

Atomic instructions (ALCLRW, ALCLRD, ALADDW, ALADDD, ACSWAPW, ACSWAPD) are considered as read/write accesses. As a result, they need both “R” and “W” rights in the PA2 field to execute without triggering a PROTECTION trap.

4.5 32-bits and 64-bits modes

The Kalray kv3 core supports two modes of execution: 32-bits and 64-bits mode. The behaviour of the MMU is exactly the same for the two execution modes except on one particular point: when a data access crosses the first 4GB boundary (e.g. a LD at address 0xFFFF_FFFC), then:

- in 32 bit addressing mode, the MMU will consider that the second half of the access happens at virtual address 0;
- in 64 bit addressing mode, the MMU will consider that the second half of the access happens at virtual address 0x1_0000_0000.

4.6 TLB Shootdown Procedure

In case of multiple cores sharing the page table data structures or physical page frames, the TLB entries inside those cores must be kept consistent. In particular, any change in a page table descriptor by one core must be propagated to the other cores. This propagation can be achieved by a so-called TLB shootdown procedure, where the TLB entries corresponding to a physical address are flushed in those cores.

The Kalray kv3 core does not provides specific resources for helping a TLB shootdown procedure.

4.7 Default memory space mapping

When the MMU is disabled (PS.MME = 0), all the memory accesses from the core are flat-mapped, that is the MMU outputs a physical address identical to the virtual address it received. The memory space is globally considered as unprotected (as if PA2 = 13: RWX, RWX). All addresses are considered as in Device space (CP2 = 8, implying no misalignment is allowed) except five pre-defined regions that are treated as fully cached (CP2 = 7). These five regions are:

- The 0x00_0000_0000 to 0x00_007F_FFFF range (= 0 MB to 8 MB = ”local SMEM”)
- The 0x00_0100_0000 to 0x00_017F_FFFF range (= 16 MB to 24 MB = ”cluster 0 SMEM”)
- The 0x00_0200_0000 to 0x00_027F_FFFF range (= 32 MB to 40 MB = ”cluster 1 SMEM”)

Kalray – all rights reserved

- The 0x00_0300_0000 to 0x00_037F_FFFF range (= 48 MB to 56 MB = ”cluster 2 SMEM”)
- The 0x00_0400_0000 to 0x00_047F_FFFF range (= 64 MB to 72 MB = ”cluster 3 SMEM”)
- The 0x00_0500_0000 to 0x00_057F_FFFF range (= 80 MB to 88 MB = ”cluster 4 SMEM”)
- The 0x00_00D2_4000 to 0x00_00D2_BFFF range (= 32 kB wide, starting at OTP Boot Rom start address)
- The 0x00_1800_0000 to 0x00_1FFF_FFFF range (= 384 MB to 512 MB = ”XIP Flash”)
- The 0x00_8000_0000 to 0x00_FFFF_FFFF range (= 2GB to 4GB = ”alias local DDR”)
- The 0x01_0000_0000 to 0x10_FFFF_FFFF range (= 4GB to 68GB = ”local DDR”)
- The above memory ranges plus 256GB, 512GB and 768GB (other MPPA in a multi-chip configuration, or the same MPPA through RaW checker)

Furthermore, fetch or data accesses to virtual addresses above 0xFF_FFFF_FFFF (1TB - 1) will respectively trigger PSYSERROR and DSYSERROR traps (as there is no corresponding flat-mapped physical address in the current implementation of the physical memory map).

4.8 Local Cluster address space

The local cluster address space is aliased : it can be seen from 2 address ranges, the ”local” one [0MB; 16MB[and the ”absolute” one [(cluster_id+1)*16MB; (cluster_id+2)*16MB[. To ensure the L1-coherency and prevent the same memory data to be cached twice with 2 aliased addresses, the MMU always (even when off) translates the absolute address space of the current cluster to the local one, whatever the FN of MMU pages.

4.9 Virtual Spaces

In order to provide a clean separation between (e.g.) an hypervisor and a guest OS running on the same Kalray kv3 core, (configurable) separate virtual spaces (VS) are added to the architecture, one per PL.

This requires that 2 bits (VS[1:0]), living in PS and SPS_PL*, be used by hardware during the lookup of the MMU TLB entries for normal fetch and data virtual accesses. In a similar way as a process runs with a particular ASN, at a certain time, a Privilege Level PL$< i >$ always executes within a particular VS. During the TLB lookup, for a match to occur between the requested virtual address and a TLB entry, the VS of the access has to match that of the entry.

Starting from the most privileged PL (PL0), each PL will be able to choose: its own VS and PS.MME bit value; and whether less privileged PLs can choose their VS and control their PS.MME bit or not by making use of the VS / MME ownership bits in PSO.

Inside a VS, the usual matching rules apply with respect to the ASN and G bits. The PS.MMUP bit controls if the current PL is considered privileged or not inside its VS with respect to the RWX attributes of the TLB entries. The PS.V64 bit controls if the current PL operates in 64 bit (V64 =

Kalray – all rights reserved

1) or 32 bit addressing mode. When the PL changes (following the firing of an exception or the execution of a RFE), the current VS naturally changes to that of the target ring.

In the case of an hypervisor running in PL0, an OS in PL1 and a User in PL2, an expected setup for Virtual Spaces would be to have one for PL0 (e.g. SPS_PL0.VS = 0) and a different one common to PL1 and PL2 (e.g. SPS_PL1.VS = SPS_PL2.VS = 1), with PL0 owning *PS*.VS in PSO.

Kalray – all rights reserved

IT	Source
0	Timer 0
1	Timer 1
2	Watchdog
3	Performance monitors
4	APIC line 0
5	APIC line 1
6	APIC line 2
7	APIC line 3
12	SECC error from memory system
13	Arithmetic exception (carry and IEEE 754 flags)
16	Data Asynchronous Memory Error (DAME), raised for DECC/DSYS errors
17	CLI (Cache Line Invalidation) for L1D or L1I following DECC/DSYS/Parity errors

Table 2: Wiring of interrupt lines.

5 Integrated Peripherals

5.1 Interrupt Controller

5.1.1 Interrupt Controller Overview

The Kalray kv3 core includes a hardware interrupt controller (ITC), with the following features:

- 32 independent interrupt sources, with priority management between the 32 interrupts allowing interrupt nesting.
- Fully configurable for each interrupt, with 2-bit priority level for each privilege level (PL), and individual interrupt source enable bit.
- Interrupt status bit displaying the pending interrupts.

The Kalray kv3 core directly manages 32 interrupt lines. Among them, some are dedicated to hardware sources, while others are not and could be used by the software to implement virtual interrupts coming from virtual devices (raising interrupts by software is performed by writing to ILR). Table 2 summarizes the wired interrupt connections in the MPPA Coolidge processor.

5.1.2 Interrupt Triggering

Each of the 32 interrupt lines is owned by a particular PL, as configured in ITO. It also has a priority level, encoded in ILL (interrupt line level) on 2 bits, that is relative to the owner privilege level (PL). The concatenation of the interrupt line owner PL (on the MSBs) and the interrupt line relative level (on the LSBs) forms a quadruplet of bits that defines the interrupt line absolute priority. The interrupt controller then arbitrates between all the pending (ILR[i] = 1) and enabled (ILE[i] = 1) interrupts and submits the one with the highest absolute priority. Then:

Kalray – all rights reserved

- If the processor is currently running in a privileged mode that is less privileged than the owner of the candidate interrupt, the interrupt is always taken (regardless of PS.IE and PS.IL) and control is transferred to the interrupt owner PL.
- If the processor is currently running in a privileged mode that is more privileged than the owner of the candidate interrupt, the interrupt is not taken (regardless of PS.IE and PS.IL) and remains pending.
- If the processor is currently running in the candidate interrupts owner PL, the interrupt is taken if and only if PS.IE = 1 and it_rel_level $>$ PS.IL, where ir_rel_level is the relative priority level of the interrupt on 2 bits, as defined in ILL. If the interrupt is taken, then a horizontal interrupt happens in the current PL.

This mechanism guarantees that:

- A more privileged PL is always able to interrupt a less privileged one.
- A less privileged PL never interrupts a more privileged one.
- Inside a given PL, interrupt nesting happens only in a controlled way, using the PS.IE and PS.IL fields.

5.1.3 ITC Control SFRs

The following SFRs control the behavior of the ITC:

ILR is a 32-bit SFR. Each bit, when set, indicates that the corresponding interrupt line has a pending request.

ILE is a 32-bit SFR. Each bit, when set, indicates that the corresponding interrupt line is enabled, so in case the corresponding ILR bit is set, the interrupt is eligible for arbitration.

ILL is a 64-bit SFR that contains 32 bit-fields of 2 bits each. This defines the priority of the corresponding interrupt for the current PL. The absolute priority of an interrupt is computed as NOT(ITO.IT$<i>$)*4 + ILL.IT$<i>$.

5.2 Timers and Watchdog

The Kalray kv3 core provides two 64-bit general purpose timers, as well as a dedicated Watchdog 64-bit counter.

5.2.1 Timers Value Registers (T0V and T1V)

T0V and T1V registers contain the current value of the timers and are reset to 0x0.

When a timer is enabled, it is decremented on each tick of the CPU clock, until it reaches 0. Upon the next tick, the timer value is loaded with its reload value (T0R or T1R), and the corresponding status bit is set in the TC register.

Kalray – all rights reserved

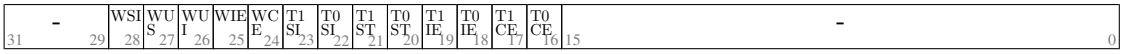
5.2.2 Timers Reload Registers (T0R and T1R)

T0R and T1R are the two 64-bit timers reload values, they are reset to 0x0.

Upon tick of CPU clock, when a timer is equal to 0x0, the corresponding reload value is transferred to the timer.

When such an underflow occurs, depending on a dedicated interrupt enable bit (located also in the TC register), a pulse on the corresponding interrupt line is generated.

5.2.3 Timers Control Register (TCR)



The TC register, also reset to 0x0, contains the following fields, listed below:

T0CE Timer 0 Count Enable

When set, enables T0 timer counting.

T1CE Timer 1 Count Enable

When set, enables T1 timer counting.

T0IE Timer 0 Interrupt Enable

When set, and if T0CE is also set, enables T0 interrupt generation.

T1IE Timer 1 Interrupt Enable

When set, and if T1CE is also set, enables T1 interrupt generation.

T0ST Timer 0 Status

T0 status bit, set automatically by hardware when T0 has expired.

T1ST Timer 1 Status

T1 status bit, set automatically by hardware when T1 has expired.

T0SI Stop Timers in Idle

When set, Timers 0 does not count when the core is in idle state (await, sleep or stop).

T1SI Stop Timers in Idle

When set, Timers 1 does not count when the core is in idle state (await, sleep or stop).

WCE Watchdog Counting Enable

Enables WDV counting.

WIE Watchdog Interrupt Enable

Enables the interrupt generation (IT(2) of RM core and PE cores) when the WDV register underflows (wraps back to 0x0).

Kalray – all rights reserved

WUI Watchdog Underflow Inform

Informs the control logic that an underflow occurs while the status bit is set (this is the watchdog function enable).

WUS Watchdog Underflow Status

This is a status bit (set on WDV underflow) which can be cleared only by software.

WSI Watchdog Stop in Idle

When set, the watchdog does not count when the core is in idle state (await, sleep or stop).

5.2.4 Timers Programming

As an extra cycle is needed when a timer reaches 0x0, the duration between two underflows of a timer is equal to the reload value plus one. For example, a reload value of 9 will cause an underflow every 10 ticks.

Timer 0 interrupt is connected to the interrupt line 0 of the interrupt controller.

Timer 1 interrupt is connected to the interrupt line 1 of the interrupt controller.

5.2.5 Watchdog Programming

The purpose of the Watchdog timer function is to allow detecting unexpected behavior, such as software or hardware deadlocks. This watchdog timer function is implemented using several resources:

- WDV (Watchdog Value) holds the current watchdog counter value. WDV is a 64-bit counter, counting backwards on the clock ticks (it uses the same clock as the 2 timers, that is the main CPU clock). It can be modified through SET operations only (WFX* cannot be used), in privilege mode.
- WDR (Watchdog Reload) holds the watchdog reload value. WDR is a simple 64-bit register. It can be modified through SET operations only (WFX* cannot be used), in privilege mode.
- Some bits of TC control the watchdog timer functions:
  - WCE (Watchdog Counting Enable) enables WDV counting.
  - WIE (Watchdog Interrupt Enable) enables the interrupt generation (IT(2) of RM core and PE cores) when the WDV register underflows (wraps back to 0x0).
  - WUI (Watchdog Underflow Inform) enables the control logic that an underflow occurs while the status bit is set (this is the watchdog function enable).
  - WUS (Watchdog Underflow Status) is a status bit (set on WDV underflow) which can be cleared only by software.

When TC configures the watchdog function (TC.WUI=1, TC.WCE=1, TC.WIE=1), then:

- On WDV underflow, TC.WUS is set, and the interrupt line 2 is asserted.

Kalray – all rights reserved

- WDV is reloaded with the WDR value, and restarts de-counting.
- The software (the interrupt routine for example) has to reset TC.WUS.
- If a new WDV underflow occurs before TC.WUS has been reset, then the control logic is informed, and:
  - The faulty core will come back to the reset state (and will restart executing its code if the wake-up conditions are met).
  - In case the faulty core is a PE, an interrupt will be sent to the Resource Manager, which will be able to identify the core through memory mapped resources.
  - In case the faulty core is the RM, no other core is aware of this situation.

  So, in order to prevent the watchdog to fire, TC.WUS has to be reset between every two WDV underflows.

The watchdog registers can also be used as a third simple timer. For this purpose, if TC.WUI=0, TC.WCE=1, TC.WIE=1, then the interrupt line 2 will be asserted on WDV underflows. No extra actions will start if TC.WUS is set when WDV underflows again.

5.3 Performance Monitoring

The Kalray kv3 core provides eight 64-bit counters that allow to monitor various performance events, with the ability to trigger interrupts on overflow.

5.3.1 Performance Monitor Counters

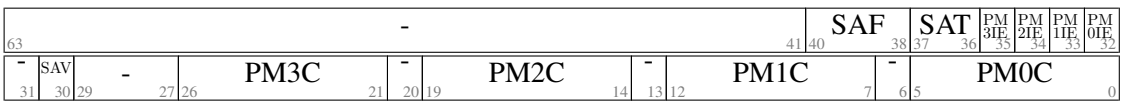
PM0, PM1, PM2, PM3, PM4, PM5, PM6 and PM7 registers contain the value of the counters of the performance monitors. If their trigger condition (depending on PMC value) is met, they are incremented (by one or more, depending on the selected event).

The PMC register allows to configure the counters counting conditions.

5.3.2 Performance Monitor Count Registers (PM0 to PM7)

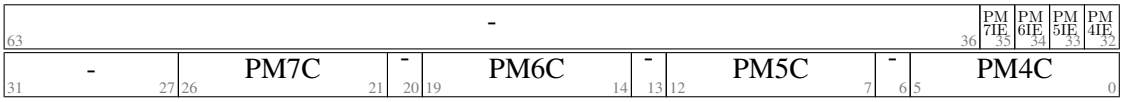
PM0, PM1, PM2, PM3, PM4, PM5, PM6 and PM7 are the eight 64-bit counters, they are reset to 0x0.

5.3.3 Performance Monitor Control Registers (PMC and PMC2)



Kalray – all rights reserved

©2025 Kalray



The PMC and PMC2 registers, contain the following fields:

PM0C PM0 Configuration

Contains the code of the event counted by PM0.

PM1C PM1 Configuration

Contains the code of the event counted by PM1.

PM2C PM2 Configuration

Contains the code of the event counted by PM2.

PM3C PM3 Configuration

Contains the code of the event counted by PM3.

SAF Saved Address From

This 3-bit field is set by the HW with the index of the PM which has overflowed

SAV Saved Address Valid

This bit is automatically set by hardware when a performance monitor for which PM< i >IE bit is set overflows (i.e. increments by one or more and wraps from 0xFFFFFFFF_FFFFFF.. to 0x00000000_000000..) When this happens:

- an interrupt is sent to the interrupt controller (on line 3)
- the four Performance Monitors stop counting and their current value remains “frozen”
- PMC.SAT is set to the relevant value
- the PMSA register may receive an address / PC

The eight Performance Monitors will start counting again as soon as the software clears PM.SAV

If a PM overflows while its PM< i >IE bit is not set or while PMC.SAV is already at 1, nothing happens: the overflowing PM just wraps around silently to 0x00000000_000000.. and continues counting. The other PMs continue counting too, PMC.SAT is not set, PMSA is left untouched and no interrupt is sent.

PM0IE Enables interrupt generation for PM0

PM1IE Enables interrupt generation for PM1

PM2IE Enables interrupt generation for PM2

PM3IE Enables interrupt generation for PM3

Kalray – all rights reserved

SAT Saved Address Type

This 2-bit field is automatically set by the HW when PM.SAV = 0 and a performance monitor for which PM$<i>$IE bit is set underflows. According to the nature of the counted event that caused the overflow, the PMSA register receives:

- the PC of the bundle that caused the overflowing event. In this case, PM.SAT is set to 10 (aka PC)
- an Address related to the event that caused the overflow. In this case, PM.SAT is set to 01 (aka A)
- nothing (PMSA left untouched). In this case, PM.SAT is set to 00 (aka None)

PM.SAT = 11 is a reserved value and will not be set by hardware.

PM4C PM4 Configuration

Contains the code of the event counted by PM4.

PM5C PM5 Configuration

Contains the code of the event counted by PM5.

PM6C PM6 Configuration

Contains the code of the event counted by PM6.

PM7C PM7 Configuration

Contains the code of the event counted by PM7.

PM4IE Enables interrupt generation for PM4

PM5IE Enables interrupt generation for PM5

PM6IE Enables interrupt generation for PM6

PM7IE Enables interrupt generation for PM7

The table below (Table 9) details all the events that can be counted by the kvx core, with their associated code and behavior on overflow.

Kalray – all rights reserved

©2025 Kalray

Code	Name	Description	SAT	Comment
0	PCC	Processor Clock Cycle	None	+1 at every clock cycle, even during idle modes.
1	ICC	Idle Clock Cycle	None	+1 at every cycle during which the main clock is cut because of idle mode.
2	EBE	Executed Bundle Event	PC	+1 for each bundle architecturally executed. Do not count bundles victim of a HW trap as they are not architecturally executed.
3	ENIE	Executed N Instructions Event	PC	+n for each bundle architecturally executed, with n = number of main syllables (32 bits) that are not IMMX.
4	ENSE	Executed N Syllables Event	PC	+n for each bundle architecturally executed, with n = number of 32 bits syllables in the bundle (included IMMX).
5	ICHE	ICache Hit Event	A	+1 for each ICache hit (micro architectural: a single bundle may be responsible for +2).
6	ICME	ICache Miss Event	A	+1 for each true ICache miss (idem). Does not increment on uncached accesses.
7	ICMABE	ICache Memory Accesses Burst Event	A	+1 each time the ICache requests a burst of data from the memory system, be it on a miss (refill) or because the access was uncached.
8	MNGIC	Memory Not Granting Instruction cache access Cycle	None	+1 at every cycle during which the memory system does not grant an Instruction Cache access.

Table 3: Performance counting conditions.

Kalray – all rights reserved

Code	Name	Description	SAT	Comment
9	MIMHE	MMU Instruction Micro-TLB Hit Event	A	+1 for every fetch access that hits in I-uTLB.
10	MIMME	MMU Instruction Micro-TLB Miss Event	A	+1 for every fetch access that misses in I-uTLB.
11	IATSC	Instruction Address Translation Stall Cycle	None	+1 at every cycle during which the ICache waits for an address translation from the MMU.
12	FE	Fetch Event	None	+1 each time a PFB fetch request is granted by the ICache.
13	PBSC	Prefetch Buffer Starvation Cycle	None	+1 at every cycle during which a PFB fetch request to the ICache is not granted by the latter.
14	PNVC	Pipeline No Valid Cycle	None	+1 at every cycle during which the first stage of the pipeline (Fetch—ID) has no valid bundle to execute and the main clock is running (not in idle mode).
15	PSC	Pipeline Starvation Cycle	None	+1 at every cycle during which the first stage of the pipeline (Fetch—ID) has no valid bundle to execute and is not stalled by a bundle at RR, nor canceled by a downstream branch, and the main clock is running (not in idle mode)..
16	TADBE	Taken Application Direct Branch Event	PC	+1 for each architecturally executed (not canceled by a later trap) direct branch that is taken (GOTO, CB, CALL, RET).
17	TABE	Taken Application Branch Event	PC	+1 for each architecturally executed (not canceled by a later trap) applicative branch that is taken (TADBE + ICALL, IGOTO).
18	TBE	Taken Branch Event	None	+1 for each branch taken by the core (TABE + SCALL, ITs, HWtraps, re-fetches on BARRIER, TLBWRITE, etc.). Only automatic hardware loop branch-back are not counted as they never result in branch penalty cycles.

Table 4: Performance counting conditions.

Kalray – all rights reserved

Code	Name	Description	SAT	Comment
19	MDMHE	MMU Data Micro-TLB Hit Event	PC	+1 for every data access that hits in d-uTLB
20	MDMME	MMU Data Micro-TLB Miss Event	PC	+1 for every data access that misses in d-uTLB
21	DATSC	Data Address Translation Stall Cycle	None	+1 at every cycle during which the DCache (and pipeline) wait for an address translation from the MMU
22	DCLHE	DCache Load Hit Event	PC	+1 for each architecturally executed load/dtouchl that hits in the L1 data cache (a bundle might be responsible for +2 if a misaligned load spans over two cache lines) Does not increment if the load/dtouchl is canceled (e.g. by a MMU trap)
23	DCHE	DCache Hit Event	PC	+1 for each architecturally executed load/DTOUCHL/store that hits in the L1 data cache (a bundle might be responsible for +2 if a misaligned load-/store spans over two cache lines) Does not increment if the access is canceled (e.g. by a MMU trap) or uncached
24	DCLME	DCache Load Miss Event	PC	+1 for each architecturally executed load/dtouchl that misses in the L1 data cache (a bundle might be responsible for +2 if a misaligned load spans over two cache lines) Does not increment if the load/dtouchl is canceled (e.g. by a MMU trap) or uncached
25	DCME	DCache Miss Event	PC	+1 for each architecturally executed load/DTOUCHL/store that misses in the L1 data cache (a bundle might be responsible for +2 if a misaligned load/store spans over two cache lines) Does not increment if the access is canceled (e.g. by a MMU trap) or uncached

Table 5: Performance counting conditions.

Kalray – all rights reserved

Code	Name	Description	SAT	Comment
26	DARSC	Data Access Related Stall Cycle	None	+1 at every cycle during which a stage of the pipeline is stalled because of a data access. This encompasses cycles lost to: • Waiting on r-a-w dependencies • Waiting on MMU data translation • Misalignment penalties • DCache not granting because busy processing a miss or memory system unresponsive • Outstanding loads CAM full But not cycles lost to blocking uncached accesses nor FENCE (these are counted in SISC).
27	LDSC	Load Dependency Stall Cycle	None	+1 at every cycle during which a stage of the pipeline is stalled because of a read-after-write dependency on an outstanding load.
28	DCNGC	Data Cache Not Granting Cycle	None	+1 at every cycle during which the data cache does not grant data accesses (be there a data access request from the pipeline or not).
29	DMAE	Data Misaligned Access Event	PC	+1 for each misaligned access that results in a performance penalty: cached accesses crossing a cache line border + uncached accesses crossing a SMEM protocol word boundary.
30	LCFSC	Load Cam Full Stall Cycle	None	+1 at every cycle during which the pipeline is stalled because the DCache outstanding load CAM is full.

Table 6: Performance counting conditions.

Kalray – all rights reserved

©2025 Kalray

Code	Name	Description	SAT	Comment
31	MNGDC	Memory Not Granting Data cache access Cycle	None	+1 at every cycle during which the memory system does not grant a Data Cache access.
32	MACC	Memory Accesses Conflict Cycle	None	+1 at every cycle during which a memory request from the ICache conflicts with a memory request from the DCache at the input of the caches arbiter.
33	TACC	TLB Accesses Conflict Cycle	None	+1 at every cycle during which a L2-TLB request from the data side or the instruction side is not granted by the L2-TLB because the latter is taking or processing a request from the other side, or a TLB management instruction (such as TLBWRITE).
34	IWC	Idle Wait Cycle	None	+1 at every cycle during which an AWAIT/SLEEP/STOP instruction stalls the pipeline, waiting for a wake-up (be the main clock cut yet or not).
35	WISC	Wait Instruction Stall Cycle	None	+1 at every cycle during which a WAITIT/SYNCGROUP instruction stalls in the pipeline, waiting for its unblocking condition to happen.
36	SISC	Synchronization Instruction Stall Cycle	None	+1 at every cycle during which the pipeline is stalled because of the execution of a FENCE or BARRIER or uncached blocking load/store instruction.
37	DDSC	Data Dependency Stall Cycle	None	+1 at every cycle during which a stage of the pipeline is stalled because of a read-after-write dependency (including on pending loads: this is a superset of LDSC).
38	SC	Stall Cycle	None	+1 at every cycle during which a stage of the pipeline is stalled, whatever the cause of the stall (all dependencies + synchronization instructions + pipeline flushes ...) and the main clock is running (not in idle mode).

Table 7: Performance counting conditions.

Kalray – all rights reserved

Code	Name	Description	SAT	Comment
39	ELE	Executed Load Event	PC	+1 for each architecturally executed load, be it cached or not (does not increment on conditional loads whose condition is not met, does not increment on dtouchl).
40	ELNBE	Executed Load N Bytes Event	PC	+n for each architecturally executed load be it cached or not, with n is the number of bytes loaded (idem).
41	ELUE	Executed Load Uncached Event	PC	+1 for each architecturally executed uncached load (does not increment on conditional loads whose condition is not met).
42	ELUNBE	Executed Load Uncached N Bytes Event	PC	+n for each architecturally executed uncached load, with n is the number of bytes loaded (idem).
43	ESE	Executed Store Event	PC	+1 for each architecturally executed store (does not increment on conditional stores whose condition is not met).
44	ESNBE	Executed Store N Bytes Event	PC	+n for each architecturally executed store, with n is the number of bytes stored.
45	EAE	Executed Atomic Event	PC	+1 for each architecturally executed atomic instruction.
46	CIRE	Coherency Invalidation Request Event	None	+1 each time the core receives a L1 DCache coherency line invalidation request from the directory.
47	CIE	Coherency Invalidation Event	None	+1 each time the core invalidates a line after receiving a L1 DCache coherency line invalidation request from the directory (subset of CIRE as all invalidation requests may not trigger an invalidation).

Table 8: Performance counting conditions.

Kalray – all rights reserved

Code	Name	Description	SAT	Comment
48	SE	Stop Event	None	STOP: Stops the counter.
49	RE	Reset Event	None	RESET = Special event that forces counter to 0.
50	FSC	Fetch Stall Cycle	None	+1 at every cycle during which a valid bundle is stalled at Fetch (ID) stage in the pipeline and the main clock is running (not in idle mode).
51	CPIRE	Coherency Precise Invalidation Request Event	A	+1 each time the core receives a precise L1 DCache coherency line invalidation request from the directory. In case of overflow (and PM<i>IE = 1), the address whose invalidation is requested is written to PMSA.
52	CPIE	Coherency Precise Invalidation Event	A	+1 each time the core invalidates a line after receiving a precise L1 DCache coherency line invalidation request from the directory (subset of CIRE as all invalidation requests may not trigger an invalidation). In case of overflow (and PM<i>IE = 1), the address whose invalidation is requested is written to PMSA.
53	HUPEVICT	Hit-Under-Prefetch Eviction Event	None	+1 each time a line gets evicted of HUP module due to a prefetch (dtouchl) when the HUP CAM is full.
54	HUPHIT	Hit-Under-Prefetch Hit Event	None	+1 each time the HUP starts transferring a line to the DCache.
55	L2LM	L2 cache Load Miss event	PC	+1 each time a full cached (L1 + L2) load miss in L2 cache. Note that the load also miss in L1 cache to be requested to L2.
56	L2LH	L2 cache Load Hit event	PC	+1 each time a full cached (L1 + L2) load hit in L2 cache. Note that the load also miss in L1 cache to be requested to L2.
57	L2SM	L2 cache Store Miss event	A	+1 each time a full cached (L1 + L2) store miss in L2 cache. Note that the store could miss or hit in L1 cache (L1 cache is write-through).

Kalray - all rights reserved

5.4 On Chip Emulation (OCE)

A *cluster* is a set of Kalray kv3 cores that share the same memory. At the cluster level a common debug unit, the *Debug System Unit* (DSU), is shared between all the cores, and an external controlling host. This shared approach allows to limit the area impact and the communication channel number between the cluster and the host.

Each core implements resources dedicated to the debug mechanisms:

- SSPS and SSPC registers, containing copies of SPS and SPC registers while diverting to debug handlers
- DV, the Debug Vector register, holding the base address of debug handlers
- OCE0 and OCE1, two general purpose SFRs dedicated to debug routine use
- OCEC and OCEA, two SFRs dedicated to debugger watchpoints.

Using these dedicated resources, debugging a core is a cooperative activity between the core and the host, using the DSU as communication mean.

Please refer to the *MPPA® Cluster Architecture* document for further details on the OCE.

5.5 Cycle counter

The Kalray kv3 core provides one 64-bit cycle counter that allow to monitor time, event if the processor is in idle state. This counter can be set to any value without ownership limitations, through SFR $FRCC (free running cycle counter).

5.6 Sequence Rupture helper PC

The Kalray kv3 core provides an SFR that logs the PC where a PC sequence rupture occurs, which cannot be traced back by other means. More specifically, this includes the following instructions:

- RET
- RFE
- GOTO
- IGOTO
- CB

The register is called SRHPC and is owner-protected through a field in $mo.

Kalray – all rights reserved

6 Instruction Set Description

6.1 Instruction Syntax

The instruction mnemonics are composed of a main part that indicates the sequence of operations, followed by an optional prefix that specifies the data type if it is not an integer word. Immediate operands always appear last in instruction syntax, except for memory instructions where the offset is located on the left of the base register. For instance:

```
addd    $r1 = $r2, $r3      // Integer add of $r2 and $r3 into $r1.
addd    $r1 = $r2, 127      // Integer add of $r2 and 127 into $r1.
addds   $r1 = $r2, $r3      // Saturated add of $r2 and $r3 into $r1.
lbs     $r1 = 127[$r2]      // Load byte sign-extended from $r2+127.
sd      127[$r2] = $p4      // Store double word $p4 into $r2+127.
```

Most instruction operands are either register or immediate values. The Kalray kv3 core also uses *modifier* operands, which have an alphanumeric name and start by a '.' character. These modifiers always follow the instruction mnemonic in the instruction syntax:

```
cb.dnez   $r1, label        // Branch to label if $r1 is not zero.
lwz.xs $r1 = $r2[$r3]       // Load word zero extend at $r3 + ($r2 * 4).
```

6.2 Instruction Bundle Encoding

6.2.1 Structure of Instruction Bundles

Instruction bundle encoding of the Kalray kv3 core follows a simple pattern. An instruction bundle is composed of up to 8 syllables (32-bit words), whose bit 31 (denoted as `parallel` or `P`) is dedicated to encode the end of the bundle (end of bundle when zero). All syllables also dedicate bits 29 and 30 to encode the *steering* information:

0b00 BCU instruction syllables or immediate extension syllable.

0b01 LSU instruction syllable.

0b10 MAU or FPU instruction syllable.

0b11 ALU0 or ALU1 instruction syllable.

An instruction bundle is encoded with the instruction syllables first, followed by the immediate extension syllables if any. The instruction syllables in a bundle are ordered:

- Instruction syllable that must issue on the BCU.
- Instruction syllable that must issue on the TCA.
- Instruction syllables that must issue on the ALU0, or both ALU0 and ALU1.
- Instruction syllables that may issue on the ALU0, the ALU1, or the tiny ALUs.
- Instruction syllable that must issue on the MAU or the FPU.
- Instruction syllable that must issue on the LSU.

Kalray – all rights reserved

6.2.2 Immediate Extensions in Instruction Bundles

The immediate extension syllables must appear after all the instruction syllables. In the immediate extension syllables, the steering bits are 0b00, while the `exnum` field in bits 27 and 28 encodes which execution unit should make use of the immediate extension:

0b00 Immediate extension for the instruction syllable issued on the ALU0.

0b01 Immediate extension for the instruction syllable issued on the ALU1.

0b10 Immediate extension for the instruction syllable issued on the MAU or the FPU.

0b11 Immediate extension for the instruction syllable issued on the LSU.

Excess immediate extension syllables for a particular instruction syllable are ignored. Immediate extension syllables for instruction syllables without immediate operand are also ignored. This feature can be exploited to pad bundles for the alignment of subsequent instruction bundles, as long as the total number of syllables does not exceed 8:

- If there is no ALU instruction, pad the end of bundle with the NOP syllable (0x00000000).
- If there is an ALU0 or double ALU instruction without immediate operand, pad the end of bundle with the NOP syllable.
- If there is an ALU0 or double ALU instruction with and extended immediate operand, pad the end of bundle with the NOP syllable.
- If there is an ALU0 or double ALU instruction with non-extended immediate operand, pad the end of bundle with an immediate extension syllable that sign-extends the original immediate operand.

Indeed, the NOP syllable is interpreted as an ALU0 immediate extension.

6.3 Instruction Execution

The execution of instructions is described using a pseudo-C language. This language retains the C syntax, however, it operates on bool(eans) and int(egers) of unbounded size. The pseudo-C builtin functions are listed in Table 10. The core stateful resources are listed in Table 12, including the register files that behave like arrays, and a memory interface MEM that implement the methods listed in Table 11. Finally, the core-specific functions are listed in Table 13.

Kalray – all rights reserved

Identifier	Description
_ABS	Absolute value
_BMM_*n*	Bit Matrix Multiply $n \times n$
_BMT_*n*	Bit Matrix Transpose $n \times n$
_CBS.*n*	Count bit set on *n* bits
_CLS.*n*	Count leading sign on *n* bits
_CLZ.*n*	Count leading zero on *n* bits
_CTZ.*n*	Count trailing zero on *n* bits
_MAX	Maximum
_MIN	Minimum
_ROL_*n*	Rotate Left modulo *n* bits
_ROR_*n*	Rotate Right modulo *n* bits
_SAT.*n*	Saturation to *n* bits
_SWAP_*n*	Swap the pairs of *n* bits slices
_SX_*n*	Sign extension on *n* bits
_THROW	Throw hardware trap
_ZX_*n*	Zero extension on *n* bits

Table 10: Pseudo-C builtin functions.

Identifier	Description
.cas	Memory compare and swap
.dflush	Memory data cache flush
.dflushl	Memory data cache flush line
.dinval	Memory data cache invalidate
.dinvall	Memory data cache invalidate line
.dpurge	Memory data cache purge
.dpurgel	Memory data cache purge line
.dtouchl	Memory data cache touch line
.dzerol	Memory data cache zero line
.fence	Memory fence
.iinval	Memory instruction cache invalidate
.iinvall	Memory instruction cache invalidate line
.dtouchl	Memory instruction cache touch line
.load	Memory load
.loadd	Memory load double
.loadu	Memory load unsigned
.loadud	Memory load unsigned double
.store	Memory store
.storeu	Memory store uncached
.wflush	Memory flush write buffer

Table 11: Memory interface methods.

Kalray – all rights reserved

Identifier	Description
SFR[]	System Function Registers
GPR[]	General Purpose Registers
PGR[]	Paired General Registers
QGR[]	Quadrupled General Registers
XCR[]	Extension Compute Registers
XBR[]	Extension Block Registers
XVR[]	Extension Vector Registers
XTR[]	Extension Tile Registers
XMR[]	Extension Matrix Registers
MEM	Memory interface name

Table 12: Pseudo-C symbols.

Identifier	Description
bwlu_*n*	Bit-wise look-up on *n* bits
comparison_*n*	Comparison on *n* bits
condition_*n*	Condition test on *n* bits
fabs_*n*	Floating point absolute value on *n* bits
fadd_*n*_*n*	Floating point add on *n* bits
fdma_*n*_*m*	Floating point multiply add on *n* bits, result on *m* bits
fdms_*n*_*m*	Floating point multiply subtract on *n* bits, result on *m* bits
ffma_*n*_*m*	Floating point fuse multiply add on *n* bits, result on *m* bits
ffman_*n*_*m*	Floating point fuse multiply add negate on *n* bits, result on *m* bits
ffms_*n*_*m*	Floating point fuse multiply subtract on *n* bits, result on *m* bits
ffmsn_*n*_*m*	Floating point fuse multiply subtract negate on *n* bits, result on *m* bits
fixed_*n*	Floating point conversion to fixed point on *n* bits
fixedu_*n*	Floating point conversion to unsigned fixed point on *n* bits
float_*n*	Fixed point conversion to floating point on *n* bits
floatu_*n*	Unsigned fixed point conversion to floating point on *n* bits
floatcomp_*n*	Floating point compare on *n* bits
fmax_*n*	Floating point maximum on *n* bits
fmin_*n*	Floating point minimum on *n* bits
fmul_*n*_*m*	Floating point multiply on *n* bits, result on *m* bits
fmuln_*n*_*m*	Floating point multiply negate on *n* bits, result on *m* bits
fnarrow_*n*_*m*	Floating point narrow from *n* bits to *m* bits
fneg_*n*	Floating point negate on *n* bits
fsbf_*n*_*m*	Floating point subtract *n* bits, result on *m* bits
fsdiv_*n*	Floating point seed for division on *n* bits
fsisr_*n*	Floating point seed for inverse square root on *n* bits
fwiden_32_64	Floating point widen single to double

Table 13: Pseudo-C external functions.

Kalray – all rights reserved

6.4 BCU Instructions

6.4.1 AWAIT — Wait for any event or interrupt.

Encoding Format BCU_IPC (Section 6.12.50), bcucode3 = 01000

P	00	01	1111	01000	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `await`

Description Wait until **WS.WU0** equals one (cf Idle Modes chapter). The main core clock is gated during the waiting period to reduce power consumption.

Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  idle(0);
```


Kalray – all rights reserved

6.4.2 BARRIER — Instruction Pipeline Barrier

Encoding Format BCU_IPC (Section 6.12.50), bcucode3 = 01011

P	00	01	1111	01011	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `barrier`

Description Flush the core instruction pipeline and memory system.

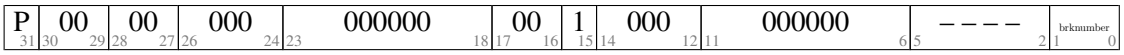
Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  barrier();
```


Kalray – all rights reserved

6.4.3 BREAK — Debugger Breakpoint

Encoding Format BCU_BREAK (Section 6.12.4), silent = 1



Syntax `break` *brknumber*

Description Breakpoint for the debugger.

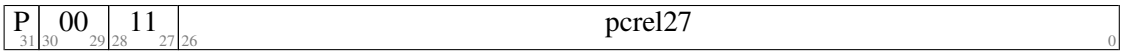
Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  new argument1 = _ZX_2(brknumber);
stage RR:
  break(argument1);
```


Kalray – all rights reserved

6.4.4 CALL — Call Subroutine PC-Relative

Encoding Format BCU_UB (Section 6.12.65), bcucode1 = 11



Syntax `call` *pcrel27*

Description The sequential next bundle **PC** is copied into the **RA** register. The next bundle **PC** is computed by adding to the current bundle **PC** the *pcrel27* after sign extension and scaling by 4.

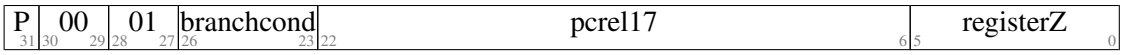
Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument1 = (_SX_27(pcrel27)<<2);
  new next = NPC;
  new address = PC + _SX_32(argument1);
  NPC = address;
  branch_info(1, address);
stage RR:
  RA = next;
```


Kalray – all rights reserved

6.4.5 CB Conditional Branch PC-Relative

Encoding Format BCU_CB (Section 6.12.47), bcucode1 = 01



Syntax `cb`*branchcond registerZ? pcrel17*

Description If *registerZ* satisfies condition *branchcond*, the next bundle **PC** is computed by adding to the current bundle **PC** the *pcrel17* after sign extension and scaling by 4. **SRHPC** register may be updated according to current privilege level and owners configuration.

Modifier(s)

- branchcond (cf. scalarcond in section 6.10.2 on page 1425)

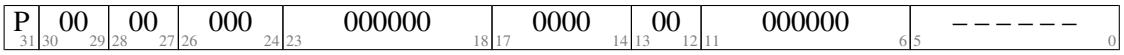
Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument1 = _ZX_4(branchcond);
  new argument2 = GPR[registerZ];
  new argument3 = (_SX_17(pcrel17)<<2);
stage RR:
  if (scalarcond(argument1, argument2)) {
    new address = PC + _SX_64(argument3);
    NPC = address;
    branch_info(1, address);
    srhpc_update();
  } else {
    branch_info(0, 0);
  }
```


Kalray – all rights reserved

6.4.6 ERROP — Error Opcode

Encoding Format BCU_NOB (Section 6.12.1), bcucode6 = 000000



Syntax `errop`

Description Invalid opcode, should always trap.

Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  _THROW(OPCODE);
```


Kalray – all rights reserved

6.4.7 GET Get System Register

Encoding Format BCU_GSR (Section 6.12.57), bcucode3 = 10001

P	00	01	1111	10001	– – –	systemS2	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `get` *registerZ* = *systemS2*

Description The *registerZ* is copied from the *systemS2* register.

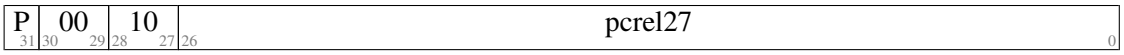
Execution Reservation table BCU_TINY_TINY_MAU_XNOP (Section 6.14.24)

```
stage RR:
  new argument2 = SFR[systemS2];
  if (get_check_access(systemS2, registerZ)) {
    new result1 = get(systemS2, _ZX_64(argument2));
  }
stage E1:
  GPR[registerZ] = result1;
```


Kalray – all rights reserved

6.4.8 GOTO — Branch Unconditional PC-Relative

Encoding Format BCU_UB (Section 6.12.65), bcucode1 = 10



Syntax `goto` *pcrel27*

Description The next bundle **PC** is computed by adding to the current bundle **PC** the *pcrel27* after sign extension and scaling by 4. **SRHPC** register may be updated according to current privilege level and owners configuration.

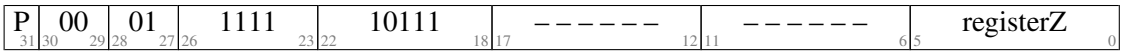
Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument1 = (_SX_27(pcrel27)<<2);
  new address = PC + _SX_32(argument1);
  NPC = address;
  branch_info(1, address);
stage RR:
  srhpc_update();
```


Kalray – all rights reserved

6.4.9 ICALL Indirect Call from Register

Encoding Format BCU_IBC (Section 6.12.62), bcucode3 = 10111



Syntax `icall` *registerZ*

Description The sequential next bundle **PC** is copied into the **RA** register. The next bundle **PC** is computed by reading the *registerZ*.

Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument1 = GPR[registerZ];
  new next = NPC;
stage RR:
  NPC = argument1;
  RA = next;
  branch_info(1, argument1);
```


Kalray – all rights reserved

6.4.10 IGET — Indirect Get System Register

Encoding Format BCU_IGSR (Section 6.12.60), bcucode3 = 10011

P	00	01	1111	10011	-------	-------	registerZ
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `iget` *registerZ*

Description The *registerZ* is copied from the system register whose index is in the *registerZ*.

Execution Reservation table BCU_TINY_TINY_MAU_XNOP (Section 6.14.24)

```
stage ID:
  new index = _ZX_9(GPR[registerZ]);
stage RR:
  if (get_check_access(index, registerZ)) {
    new result1 = get(index, SFR[index]);
  }
stage E1:
  GPR[registerZ] = result1;
```


Kalray – all rights reserved

6.4.11 IGOTO — Indirect Branch from Register

Encoding Format BCU_IBC (Section 6.12.62), bcucode3 = 10110

P	00	01	1111	10110	------	------	registerZ
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `igoto` *registerZ*

Description The next bundle **PC** is computed by reading the *registerZ*. **SRHPC** register may be updated according to current privilege level and owners configuration.

Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument1 = GPR[registerZ];
stage RR:
  NPC = argument1;
  branch_info(1, argument1);
  srhpc_update();
```


Kalray – all rights reserved

6.4.12 LOOPDO — Hardware Loop setup and Do execute

Encoding Format BCU_HLS (Section 6.12.48), bcucode2 = 1110

P	00	01	1110	pcrel17	registerZ
31	30 29	28 27	26 23	22 6	5 0

Syntax `loopdo` *registerZ*, *pcrel17*

Description If **PS.HLE** is cleared, this instruction throws an OPCODE trap. The **LS** register is set to the sequential next bundle **PC**. The **LE** register is set to the current bundle **PC** plus the *pcrel17* after sign extension and scaling by 4. The **LC** register is set with the *registerZ*. If the **PS.HLE** bit is set in the **PS** register, hardware loop execution proceeds by comparing the sequential next bundle **PC** to the value of the **LE** register:

- If different, the next bundle **PC** is computed as usual.
- If the sequential next bundle **PC** equals the value of the **LE** register:
  - If the value of the **LC** register -1 is non-zero, the value of the next bundle **PC** is read from the **LS** register, unless a branch is taken in the current bundle (in which case the next bundle **PC** is the branch target bundle **PC**).
  - If the value of the **LC** register -1 is zero, the value of the next bundle **PC** is the value of the **LE** register, unless a branch is taken in the current bundle (in which case the next bundle **PC** is the branch target bundle **PC**).
  - The **LC** register is decremented if its value is not already zero.

According to these rules, the LOOPDO instruction loops forever whenever the value of the *registerZ* is zero, as the the value of the **LC** register -1 is always non-zero, while the **LC** register is never decremented. This instruction is provided for counted loops which iterate at least once, and while loops by setting the counter to zero.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  new argument1 = GPR[registerZ];
  new argument2 = (_SX_17(pcrel17)<<2);
  if (!PS.HLE) {
    _THROW(OPCODE);
  }
  invalpfb();
  LC = argument1;
stage E2:
  LS = NPC;
  LE = PC + _SX_32(argument2);
```


Kalray – all rights reserved

6.4.13 RET — Return from Call

Encoding Format BCU_RTS (Section 6.12.61), bcucode3 = 10100

P	00	01	1111	10100	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `ret`

Description The next bundle **PC** is obtained by reading the contents of the **RA** register. **SRHPC** register may be updated according to current privilege level and owners configuration.

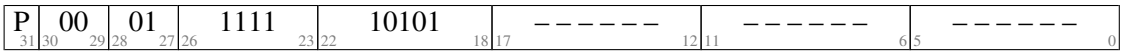
Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  NPC = RA;
stage RR:
  branch_info(1, RA);
  srhpc_update();
```


Kalray – all rights reserved

6.4.14 RFE — Return from Exception

Encoding Format BCU_RTS (Section 6.12.61), bcucode3 = 10101



Syntax `rfe`

Description The **PS** is restored from the **SPS**. The **SPS** is restored from the **SSPS**. The next bundle **PC** is obtained by reading the contents of the **SPC** register. The **SPC** is restored from the **SSPC**. **SRHPC** register may be updated according to current privilege level and owners configuration.

Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  if (rfe_owner()) {
    rfe();
    branch_info(1, SPC);
    srhpc_update();
  }
```


Kalray – all rights reserved

6.4.15 RSWAP General and System Register Swap

Encoding Format BCU_RSWAP (Section 6.12.58), bcucode3 = 10010

P	00	01	1111	10010	– – –	systemS4	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `rswap` *registerZ* = *systemS4*

Description The contents of the *registerZ* are swapped set with the contents of the *systemS4*.

Execution Reservation table BCU_TINY_TINY_MAU_XNOP (Section 6.14.24)

```
stage RR:
  new argument2 = SFR[systemS4];
  new argument1 = GPR[registerZ];
stage E1:
  if (get_check_access(systemS4, registerZ) && set_check_access(systemS4,
      _ZX_64(argument1), registerZ)) {
    new result2 = _ZX_64(argument1);
    new result1 = _ZX_64(argument2);
    GPR[registerZ] = result1;
    SFR[systemS4] = result2;
  }
```


Kalray – all rights reserved

Encoding Format BCU_RSWAPA (Section 6.12.59), bcucode3 = 10010

P	00	01	1111	10010	– – –	systemAlone	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `rswap` *registerZ* = *systemAlone*

Description The contents of the *registerZ* are swapped set with the contents of the *systemAlone*.

Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  new argument2 = SFR[systemAlone];
  new argument1 = GPR[registerZ];
stage E1:
  if (get_check_access(systemAlone, registerZ) && set_check_access(
      systemAlone, _ZX_64(argument1), registerZ)) {
    new result2 = _ZX_64(argument1);
    new result1 = _ZX_64(argument2);
    GPR[registerZ] = result1;
    SFR[systemAlone] = result2;
  }
```


Kalray – all rights reserved

6.4.16 SCALL System Call

Encoding Format BCU_SCR (Section 6.12.64), bcucode2 = 1111

P	00	01	1111	11001	-------	-------	registerZ
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `scall` *registerZ*

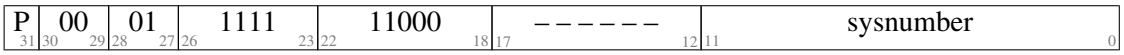
Description System call with number . Interrupts are disabled.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  new argument1 = GPR[registerZ];
  syscall(argument1);
  branch_info(1, argument1);
```


Kalray – all rights reserved

Encoding Format BCU_SCI (Section 6.12.63), bcucode2 = 1111



Syntax `scall` *sysnumber*

Description System call with number . Interrupts are disabled.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  new argument1 = _ZX_12(sysnumber);
  syscall(argument1);
  branch_info(1, argument1);
```


Kalray – all rights reserved

6.4.17 SET — Set System Register

Encoding Format BCU_SET (Section 6.12.54), bcucode3 = 10000

P	00	01	1111	10000	– – –	systemT3	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `set` *systemT3* = *registerZ*

Description The *systemT3* register is set with the *registerZ*.

Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument2 = GPR[registerZ];
stage E3:
  if (set_check_access(systemT3, _ZX_64(argument2), registerZ)) {
    new result1 = _ZX_64(argument2);
    SFR[systemT3] = result1;
  }
```


Kalray – all rights reserved

Encoding Format BCU_SETA (Section 6.12.55), bcucode3 = 10000

P	00	01	1111	10000	– – –	systemAlone	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `set` *systemAlone* = *registerZ*

Description The *systemAlone* register is set with the *registerZ*.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  new argument2 = GPR[registerZ];
stage E3:
  if (set_check_access(systemAlone, _ZX_64(argument2), registerZ)) {
    new result1 = _ZX_64(argument2);
    SFR[systemAlone] = result1;
  }
```


Kalray – all rights reserved

Encoding Format BCU_SETRA (Section 6.12.56), bcucode3 = 10000

P	00	01	1111	10000	– – –	systemRA	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `set` *systemRA* = *registerZ*

Description The *systemRA* register is set with the *registerZ*.

Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument2 = GPR[registerZ];
stage E3:
  if (set_check_access(systemRA, _ZX_64(argument2), registerZ)) {
    new result1 = _ZX_64(argument2);
    SFR[systemRA] = result1;
  }
```


Kalray – all rights reserved

6.4.18 SLEEP — Wait for an eligible interrupt.

Encoding Format BCU_IPC (Section 6.12.50), bcucode3 = 01001

P	00	01	1111	01001	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `sleep`

Description Wait until **WS.WU1** equals one (cf Idle Modes chapter). The main core clock is gated during the waiting period to reduce power consumption.

Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  idle(1);
```


Kalray – all rights reserved

6.4.19 STOP — Wait for power controller wake-up.

Encoding Format BCU_IPC (Section 6.12.50), bcucode3 = 01010

P	00	01	1111	01010	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `stop`

Description Wait until **WS.WU2** equals one (cf Idle Modes chapter). The main core clock is gated during the waiting period to reduce power consumption.

Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  if (stop_owner()) {
    idle(2);
  } else {
    _THROW(PRIVILEGE);
  }
```


Kalray – all rights reserved

6.4.20 SYNCGROUP — Synchronize Group of Cores

Encoding Format BCU_PGI (Section 6.12.51), bcucode3 = 01101

P	00	01	1111	01101	-------	-------	registerZ
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `syncgroup` *registerZ*

Description The *registerZ* is interpreted as four 16-bit fields: **waitclrF**, **waitclrB**, **notifyF**, **notifyB**. Upon executing the instruction, the core waits until all the events designated by bits in **waitclrF** and **waitclrB** in the 32 least-significant bits of **IPE** are set. When all the designated events are set, the core clears them, then notifies the event lines corresponding to the bit sets in fields **notifyF** and and **notifyB**. This notification of events is not registered in the core, it is a signal sent to set bits into the **IPE.FE** and **IPE.BE** fields of the destination cores. The successor core always receives the **notifyF** events in its **IPE.FE** field; likewise, the predecessor core always receives the **notifyB** events in its **IPE.BE** field. Furthermore, other PEs may receive these events, depending on the setting of the **IPE.FM** and **IPE.BM** fields in these neighbor PE cores.

Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument1 = GPR[registerZ];
stage RR:
  if (!syncgroup_owner()) {
    _THROW(PRIVILEGE);
  } else {
    new waitclrF = argument1 & 0xFFFF;
    new waitclrB = (argument1 >> 16) & 0xFFFF;
    new notifyF = (argument1 >> 32) & 0xFFFF;
    new notifyB = (argument1 >> 48) & 0xFFFF;
    syncgroup(waitclrF, waitclrB, notifyF, notifyB);
  }
```


Kalray – all rights reserved

6.4.21 TLBDINVAL Invalidate Data Micro-TLB.

Encoding Format BCU_TLB (Section 6.12.49), bcucode3 = 00011

P	00	01	1111	00011	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `tlbdinval`

Description Invalidate Data Micro-TLB.

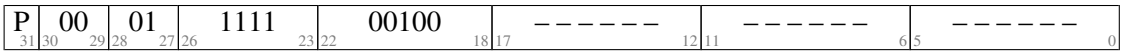
Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  invaldtlb();
```


Kalray – all rights reserved

6.4.22 TLBIINVAL — Invalidate Instruction Micro-TLB.

Encoding Format BCU_TLB (Section 6.12.49), bcucode3 = 00100



Syntax `tlbiinval`

Description Invalidate Instruction Micro-TLB.

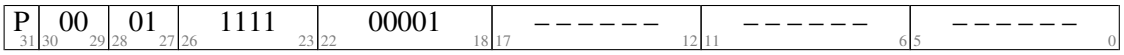
Execution Reservation table ALL (Section 6.14.1)

```
stage RR:
  invalitlb();
```


Kalray – all rights reserved

6.4.23 TLBPROBE Probe TLB

Encoding Format BCU_TLB (Section 6.12.49), bcucode3 = 00001



Syntax `tlbprobe`

Description Probe the TLB for virtual address match in the TEL:TEH register pair and write to MMC.IDX.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  if (!mmi_owner()) {
    _THROW(PRIVILEGE);
  } else {
    probetlb();
  }
```


Kalray – all rights reserved

6.4.24 TLBREAD — Read TLB Entry

Encoding Format BCU_TLB (Section 6.12.49), bcucode3 = 00000

P	00	01	1111	00000	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `tlbread`

Description Read the TLB entry into the TEL:TEH register pair using MMC.IDX.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  if (!mmi_owner()) {
    _THROW(PRIVILEGE);
  } else {
    readtlb();
  }
```


Kalray – all rights reserved

6.4.25 TLBWRITE — TLB Entry Write

Encoding Format BCU_TLB (Section 6.12.49), bcucode3 = 00010

P	00	01	1111	00010	-------	-------	-------
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `tlbwrite`

Description Write the TLB entry from the TEL:TEH register pair using MMC.IDX.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  if (!mmi_owner()) {
    _THROW(PRIVILEGE);
  } else {
    writetlb();
  }
```


Kalray – all rights reserved

6.4.26 WAITIT — Wait for Interrupt

Encoding Format BCU_PGI (Section 6.12.51), bcucode3 = 01100

P	00	01	1111	01100	------	------	registerZ
31	30 29	28 27	26 23	22 18	17 12	11 6	5 0

Syntax `waitit` *registerZ*

Description Execution waits for selected interrupts in the **ILR** register (masking non-owned interrupts, i.e. interrupts that belong to a more privileged PL. The *registerZ* is interpreted as an OR-mask in the least significant word and a AND-mask in the most significant word. The wait ends if all the bits in **ILR** match the non-zero AND-mask, or if any bit in **ILR** matches the non-zero OR-mask. When the wait is over, the value of **ILR** (masked in the same way) is written to the *registerZ*. When both the AND-mask and OR-mask are zero, no wait happens.

Execution Reservation table BCU_TINY_TINY_MAU_XNOP (Section 6.14.24)

```
stage ID:
  new argument1 = GPR[registerZ];
stage RR:
  new ormask = argument1 & 0xFFFFFFFF;
  new andmask = argument1.32[1];
  new result1 = waitit(ormask, andmask);
  GPR[registerZ] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.4.27 WFXL — Effects on Least Significant Word of System Register

Encoding Format BCU_WFX (Section 6.12.52), bcucode3 = 01110

P	00	01	1111	01110	– – –	systemT2	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `wfxl` *systemT2*, *registerZ*

Description The least significant word of the *systemT2* is operated, with the most significant word of the *registerZ* as a set mask and the least significant word of the *registerZ* as a clear mask.

Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument2 = GPR[registerZ];
stage E4:
  new argument1 = SFR[systemT2];
  if (wfxl_check_access(systemT2, registerZ)) {
    new result1 = wfxl(systemT2, _ZX_64(argument2));
  }
  SFR[systemT2] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_WFXA (Section 6.12.53), bcucode3 = 01110

P	00	01	1111	01110	– – –	systemAlone	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `wfxl` *systemAlone*, *registerZ*

Description The least significant word of the *systemAlone* is operated, with the most significant word of the *registerZ* as a set mask and the least significant word of the *registerZ* as a clear mask.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  new argument2 = GPR[registerZ];
stage E4:
  new argument1 = SFR[systemAlone];
  if (wfxl_check_access(systemAlone, registerZ)) {
    new result1 = wfxl(systemAlone, _ZX_64(argument2));
  }
  SFR[systemAlone] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.4.28 WFXM Word Effects on Most Significant Word of System Register

Encoding Format BCU_WFX (Section 6.12.52), bcucode3 = 01111

P	00	01	1111	01111	– – –	systemT2	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `wfxm` *systemT2*, *registerZ*

Description The most significant word of the *systemT2* is operated, with the most significant word of the *registerZ* as a set mask and the least significant word of the *registerZ* as a clear mask.

Execution Reservation table BCU (Section 6.14.21)

```
stage ID:
  new argument2 = GPR[registerZ];
stage E4:
  new argument1 = SFR[systemT2];
  if (wfxm_check_access(systemT2, registerZ)) {
    new result1 = wfxm(systemT2, _ZX_64(argument2));
  }
  SFR[systemT2] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_WFXA (Section 6.12.53), bcucode3 = 01111

P	00	01	1111	01111	– – –	systemAlone	registerZ
31	30 29	28 27	26 23	22 18	17 15	14 6	5 0

Syntax `wfxm` *systemAlone*, *registerZ*

Description The most significant word of the *systemAlone* is operated, with the most significant word of the *registerZ* as a set mask and the least significant word of the *registerZ* as a clear mask.

Execution Reservation table ALL (Section 6.14.1)

```
stage ID:
  new argument2 = GPR[registerZ];
stage E4:
  new argument1 = SFR[systemAlone];
  if (wfxm_check_access(systemAlone, registerZ)) {
    new result1 = wfxm(systemAlone, _ZX_64(argument2));
  }
  SFR[systemAlone] = result1;
```


Kalray – all rights reserved

6.4.29 XACCESSO Access from Extension into Octuple Word

Encoding Format BCU_XACCESSO2 (Section 6.12.8), log2size = 000

P	00	00	000	registerN	11	00	1	000	registerCg	0	registerZ
31	30 29	28 27	26 24	23 20	19 18	17 16	15	14 12	11 7	6	5 0

Syntax `xaccesso` *registerN* = *registerCg*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCg*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerN*.

Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new mask = 1;
  new buffer = registerCg << 1;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XACCESSO4 (Section 6.12.9), log2size = 000

P	00	00	000	registerN	11	00	1	000	registerCh	01	registerZ
31	30 29	28 27	26 24	23 20	19 18	17 16	15	14 12	11 8	7 6	5 0

Syntax `xaccesso` *registerN* = *registerCh*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCh*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerN*.

Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new mask = 3;
  new buffer = registerCh << 2;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XACCESSO8 (Section 6.12.10), log2size = 000

P	00	00	000	registerN	11	00	1	000	registerCi	011	registerZ
31	30 29	28 27	26 24	23 20	19 18	17 16	15	14 12	11 9	8 6	5 0

Syntax `xaccesso` *registerN* = *registerCi*, *registerZ*

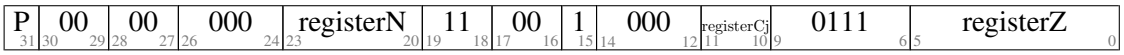
Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCi*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerN*.

Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new mask = 7;
  new buffer = registerCi << 3;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XACCESSO16 (Section 6.12.11), log2size = 000



Syntax `xaccesso` *registerN* = *registerCj*, *registerZ*

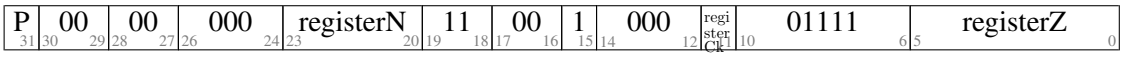
Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCj*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerN*.

Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new mask = 15;
  new buffer = registerCj << 4;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XACCESSO32 (Section 6.12.12), log2size = 000



Syntax `xaccesso` *registerN* = *registerCk*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCk*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerN*.

Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new mask = 31;
  new buffer = registerCk << 5;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XACCESSO64 (Section 6.12.13), log2size = 000

P	00	00	000	registerN	11	00	1	000	register Cl	11111	registerZ
31	30 29	28 27	26 24	23 20	19 18	17 16	15	14 12	11 10	6	5 0

Syntax `xaccesso` *registerN* = *registerCl*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCl*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerN*.

Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new mask = 63;
  new buffer = registerCl;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

6.4.30 XALIGNO Align Extension into Octuple Word

Encoding Format BCU_XALIGNO2 (Section 6.12.15), log2size = 000

P	00	00	001	registerA	00	1	000	registerCg	0	registerZ
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 7	6	5 0

Syntax `xaligno` *registerA* = *registerCg*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCg*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerA*.

Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

```
stage RR:
  new mask = 1;
  new buffer = registerCg << 1;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XALIGNO4 (Section 6.12.16), log2size = 000

P	00	00	001	registerA	00	1	000	registerCh	01	registerZ
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 8	7 6	5 0

Syntax `xaligno` *registerA* = *registerCh*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCh*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerA*.

Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

```
stage RR:
  new mask = 3;
  new buffer = registerCh << 2;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XALIGNO8 (Section 6.12.17), log2size = 000

P	00	00	001	registerA	00	1	000	registerCi	011	registerZ
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 9	8 6	5 0

Syntax `xaligno` *registerA* = *registerCi*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCi*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerA*.

Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

```
stage RR:
  new mask = 7;
  new buffer = registerCi << 3;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XALIGNO16 (Section 6.12.18), log2size = 000

P	00	00	001	registerA	00	1	000	registerCj	0111	registerZ
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 10	9 6	5 0

Syntax `xaligno` *registerA* = *registerCj*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCj*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerA*.

Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

```
stage RR:
  new mask = 15;
  new buffer = registerCj << 4;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XALIGNO32 (Section 6.12.19), log2size = 000

P	00	00	001	registerA	00	1	000	registerCk	01111	registerZ
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 10	6	5 0

Syntax `xaligno` *registerA* = *registerCk*, *registerZ*

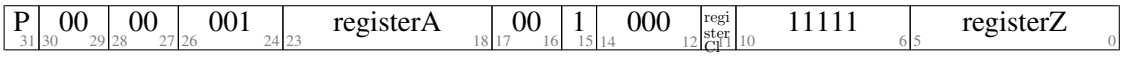
Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCk*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerA*.

Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

```
stage RR:
  new mask = 31;
  new buffer = registerCk << 5;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

Encoding Format BCU_XALIGNO64 (Section 6.12.20), log2size = 000



Syntax `xaligno` *registerA* = *registerCl*, *registerZ*

Description The 5 least significant bits of the *registerZ* are interpreted as a byte count. The most significant bits of the *registerZ* are interpreted as a vector register index into the buffer specified by the *registerCl*. The vector register indexed and its successor modulo the buffer size are concatenated, and shifted right bt the byte count. The resulting 32 bytes are written to the *registerA*.

Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

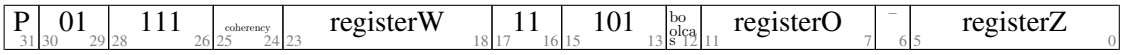
```
stage RR:
  new mask = 63;
  new buffer = registerCl;
  new address = GPR[registerZ];
  new shift = (address & 31) * 8;
  new index_0 = address >> 5;
  new index_1 = index_0 + 1;
  new where_0 = buffer + (index_0 & mask);
  new where_1 = buffer + (index_1 & mask);
  new result1_0 = XVR[where_0];
  new result1_1 = XVR[where_1];
  new result1 = result1_0;
  if (shift) {
    result1 = (_ZX_256(result1_0) >> shift) | (_ZX_256(result1_1) << (256 -
        shift));
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.5 LSU Instructions

6.5.1 ACSWAPD — Atomic Compare and Swap Double Word

Encoding Format LSU_ACSWAPSB (Section 6.12.182), lsucode5 = 101



Syntax `acswapd`*boolcascoherency registerW*, [*registerZ*] = *registerO*

Description The effective address is read from the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory double word (64 bits). The *registerO* is interpreted as the concatenation of `update` (low 64 bits) and `expected` (high 64 bits). The memory double word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory double word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerW*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be double word aligned (multiple of 8).

Modifier(s)

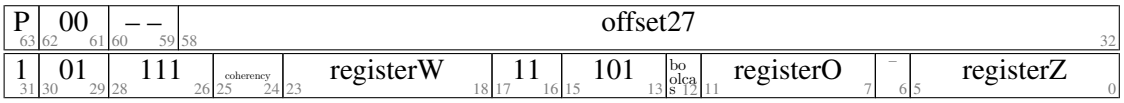
- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

Execution Reservation table LSU_AUXR_AUXW (Section 6.14.33)

```
stage ID:
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
stage E1:
  new arguments = PGR[registerO];
  new result1 = MEM.cas(address, 0xFF, boolcas | (coherency << 32), arguments
      .64[0], arguments.64[1], registerW);
stage SR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ACSWAPSB.O (Section 6.12.183), lsucode5 = 101



Syntax `acswapd`*boolcascoherency registerW*, *offset27*[*registerZ*] = *registerO*

Description The effective address is computed by adding *offset27* to the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory double word (64 bits). The *registerO* is interpreted as the concatenation of `update` (low 64 bits) and `expected` (high 64 bits). The memory double word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory double word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerW*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be double word aligned (multiple of 8).

Modifier(s)

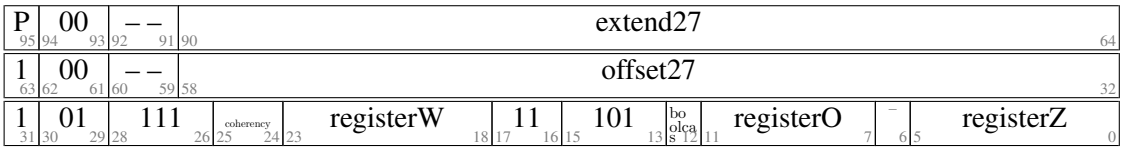
- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

Execution Reservation table LSU_AUXR_AUXW.X (Section 6.14.34)

```
stage ID:
  new offset = _SX_27(offset27);
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new arguments = PGR[registerO];
  new result1 = MEM.cas(address, 0xFF, boolcas | (coherency << 32), arguments
      .64[0], arguments.64[1], registerW);
stage SR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ACSWAPSB.D (Section 6.12.184), lsucode5 = 101



Syntax `acswapd`*boolcascoherency registerW*, *extend27_offset27*[*registerZ*] = *registerO*

Description The effective address is computed by adding *extend27_offset27* to the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory double word (64 bits). The *registerO* is interpreted as the concatenation of `update` (low 64 bits) and `expected` (high 64 bits). The memory double word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory double word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerW*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be double word aligned (multiple of 8).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

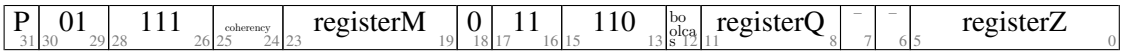
Execution Reservation table LSU_AUXR_AUXW.Y (Section 6.14.35)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new arguments = PGR[registerO];
  new result1 = MEM.cas(address, 0xFF, boolcas | (coherency << 32), arguments
      .64[0], arguments.64[1], registerW);
stage SR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.2 ACSWAPQ Atomic Compare and Swap Quadruple Word

Encoding Format LSU_ACSWAPQB (Section 6.12.185), lsucode5 = 110



Syntax `acswapq`*boolcascoherency registerM*, [*registerZ*] = *registerQ*

Description The effective address is read from the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory quadruple word (128 bits). The *registerQ* is interpreted as the concatenation of `update` (low 128 bits) and `expected` (high 128 bits). The memory double word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory double word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerM*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be quadruple word aligned (multiple of 16).

Modifier(s)

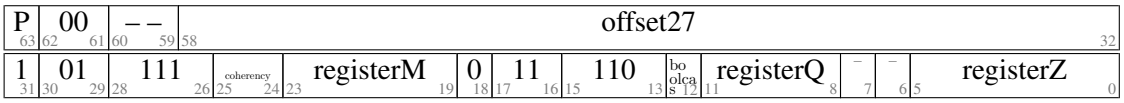
- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

Execution Reservation table LSU_AUXR_AUXW (Section 6.14.33)

```
stage ID:
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
stage E1:
  new arguments = QGR[registerQ];
  new result1 = MEM.cas(address, 0xFFFF, boolcas | (coherency << 32),
      arguments.128[0], arguments.128[1], registerM << 1);
stage SR:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ACSWAPQB.O (Section 6.12.186), lsucode5 = 110



Syntax `acswapq`*boolcascoherency registerM*, *offset27*[*registerZ*] = *registerQ*

Description The effective address is computed by adding *offset27* to the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory quadruple word (128 bits). The *registerQ* is interpreted as the concatenation of `update` (low 128 bits) and `expected` (high 128 bits). The memory double word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory double word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerM*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be quadruple word aligned (multiple of 16).

Modifier(s)

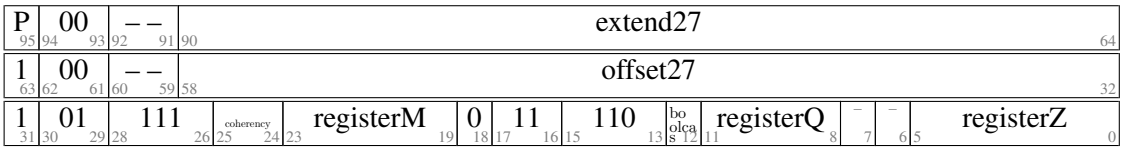
- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

Execution Reservation table LSU_AUXR_AUXW.X (Section 6.14.34)

```
stage ID:
  new offset = _SX_27(offset27);
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new arguments = QGR[registerQ];
  new result1 = MEM.cas(address, 0xFFFF, boolcas | (coherency << 32),
      arguments.128[0], arguments.128[1], registerM << 1);
stage SR:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ACSWAPQB.D (Section 6.12.187), lsucode5 = 110



Syntax `acswapq`*boolcascoherency registerM*, *extend27_offset27*[*registerZ*] = *registerQ*

Description The effective address is computed by adding *extend27_offset27* to the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory quadruple word (128 bits). The *registerQ* is interpreted as the concatenation of `update` (low 128 bits) and `expected` (high 128 bits). The memory double word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory double word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerM*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be quadruple word aligned (multiple of 16).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

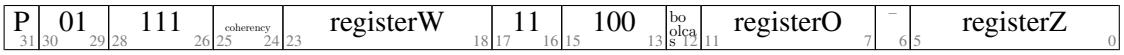
Execution Reservation table LSU_AUXR_AUXW.Y (Section 6.14.35)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new arguments = QGR[registerQ];
  new result1 = MEM.cas(address, 0xFFFF, boolcas | (coherency << 32),
      arguments.128[0], arguments.128[1], registerM << 1);
stage SR:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.5.3 ACSWAPW Atomic Compare and Swap Word

Encoding Format LSU_ACSWAPSB (Section 6.12.182), lsucode5 = 100



Syntax `acswapw`*boolcascoherency registerW*, [*registerZ*] = *registerO*

Description The effective address is read from the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory word (32 bits). The *registerO* is interpreted as the concatenation of `update` (low 64 bits) and `expected` (high 64 bits). The memory word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerW*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be word aligned (multiple of 4).

Modifier(s)

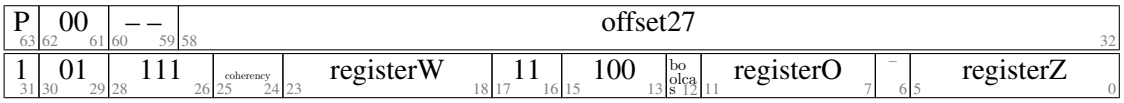
- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

Execution Reservation table LSU_AUXR_AUXW (Section 6.14.33)

```
stage ID:
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
stage E1:
  new arguments = PGR[registerO];
  new result1 = MEM.cas(address, 0xF, boolcas | (coherency << 32), arguments
      .32[0], arguments.32[2], registerW);
stage SR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ACSWAPSB.O (Section 6.12.183), lsucode5 = 100



Syntax `acswapw`*boolcascoherency registerW*, *offset27*[*registerZ*] = *registerO*

Description The effective address is computed by adding *offset27* to the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory word (32 bits). The *registerO* is interpreted as the concatenation of `update` (low 64 bits) and `expected` (high 64 bits). The memory word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerW*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be word aligned (multiple of 4).

Modifier(s)

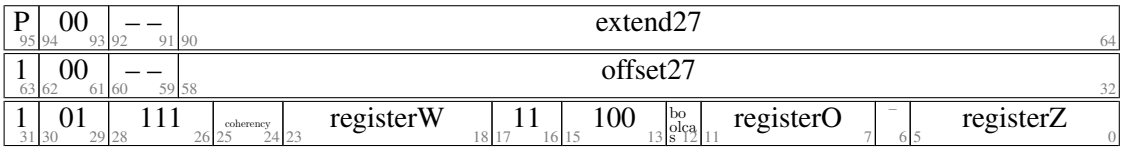
- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

Execution Reservation table LSU_AUXR_AUXW.X (Section 6.14.34)

```
stage ID:
  new offset = _SX_27(offset27);
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new arguments = PGR[registerO];
  new result1 = MEM.cas(address, 0xF, boolcas | (coherency << 32), arguments
      .32[0], arguments.32[2], registerW);
stage SR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ACSWAPSB.D (Section 6.12.184), lsucode5 = 100



Syntax `acswapw`*boolcascoherency registerW*, *extend27_offset27*[*registerZ*] = *registerO*

Description The effective address is computed by adding *extend27_offset27* to the *registerZ*. This instruction implements Compare-And-Swap (CAS) on memory word (32 bits). The *registerO* is interpreted as the concatenation of `update` (low 64 bits) and `expected` (high 64 bits). The memory word at the effective address is read into `current`. If `current` equals `expected`, the swap is successful and `update` is stored into the memory word at the effective address. Otherwise, the swap has failed and the memory is not updated. The instruction return value is stored into the *registerW*. It is true/false in case `boolcas` is true, or the `current` value in case `boolcas` is false. The effective address must be word aligned (multiple of 4).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)
- boolcas (cf. boolcas in section 6.10.2 on page 1413)

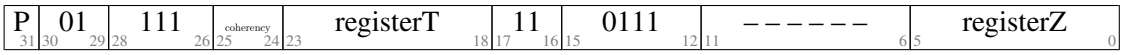
Execution Reservation table LSU_AUXR_AUXW.Y (Section 6.14.35)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new boolcas = _ZX_1(boolcas);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new arguments = PGR[registerO];
  new result1 = MEM.cas(address, 0xF, boolcas | (coherency << 32), arguments
      .32[0], arguments.32[2], registerW);
stage SR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.4 ALADDD — Atomic Load and Add Double Word

Encoding Format LSU_ALADDSB (Section 6.12.179), lsucode6 = 0111



Syntax `aladddcoherency` [*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. This instruction implements atomic load-add-store on double word. The double word at effective address is added with the *registerT*, and its previous value is returned into the *registerT*. The effective address must be double word aligned (multiple of 8).

Modifier(s)

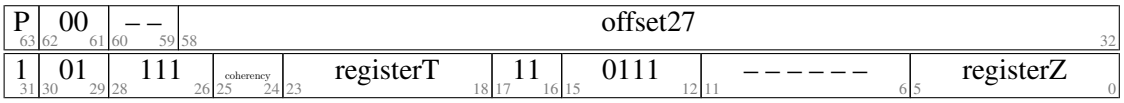
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR_AUXW (Section 6.14.33)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
stage E1:
  new argument2 = GPR[registerT];
  new result2 = MEM.add(address, 0xFF, coherency << 32, argument2.64[0],
      registerT);
stage SR:
  GPR[registerT] = result2;
```


Kalray – all rights reserved

Encoding Format LSU_ALADDSB.O (Section 6.12.180), lsucode6 = 0111



Syntax `aladdd`*coherency offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. This instruction implements atomic load-add-store on double word. The double word at effective address is added with the *registerT*, and its previous value is returned into the *registerT*. The effective address must be double word aligned (multiple of 8).

Modifier(s)

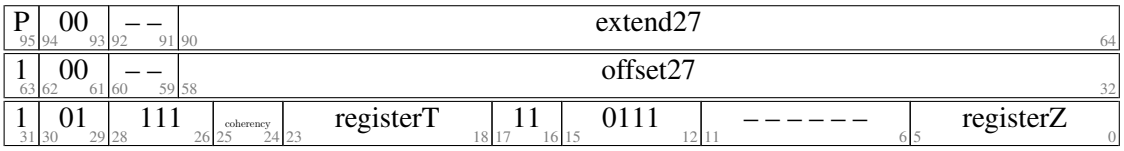
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR_AUXW.X (Section 6.14.34)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
  new result2 = MEM.add(address, 0xFF, coherency << 32, argument2.64[0],
      registerT);
stage SR:
  GPR[registerT] = result2;
```


Kalray – all rights reserved

Encoding Format LSU_ALADDSB.D (Section 6.12.181), lsucode6 = 0111



Syntax `aladdd`*coherency extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. This instruction implements atomic load-add-store on double word. The double word at effective address is added with the *registerT*, and its previous value is returned into the *registerT*. The effective address must be double word aligned (multiple of 8).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

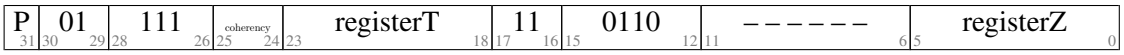
Execution Reservation table LSU_AUXR_AUXW.Y (Section 6.14.35)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
  new result2 = MEM.add(address, 0xFF, coherency << 32, argument2.64[0],
      registerT);
stage SR:
  GPR[registerT] = result2;
```


Kalray – all rights reserved

6.5.5 ALADDW Atomic Load and Add Word

Encoding Format LSU_ALADDSB (Section 6.12.179), lsucode6 = 0110



Syntax `aladdw`*coherency* [*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. This instruction implements atomic load-add-store on word. The word at effective address is added with the *registerT*, and its previous value is returned into the *registerT*. The effective address must be word aligned (multiple of 4).

Modifier(s)

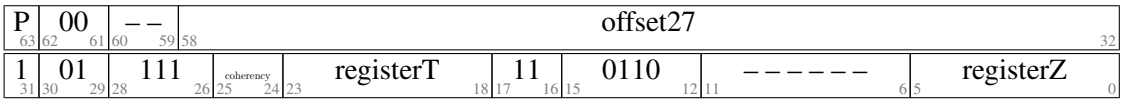
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR_AUXW (Section 6.14.33)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
stage E1:
  new argument2 = GPR[registerT];
  new result2 = _ZX_32(MEM.add(address, 0xF, coherency << 32, argument2
      .32[0], registerT));
stage SR:
  GPR[registerT] = result2;
```


Kalray – all rights reserved

Encoding Format LSU_ALADDSB.O (Section 6.12.180), lsucode6 = 0110



Syntax `aladdw`*coherency offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. This instruction implements atomic load-add-store on word. The word at effective address is added with the *registerT*, and its previous value is returned into the *registerT*. The effective address must be word aligned (multiple of 4).

Modifier(s)

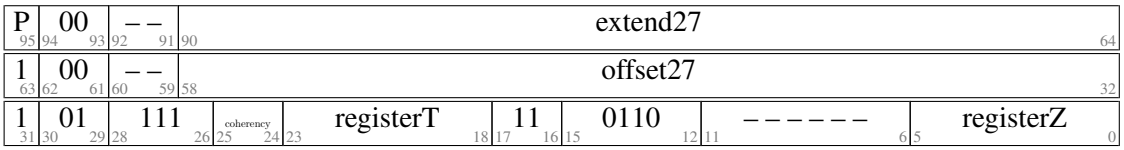
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR_AUXW.X (Section 6.14.34)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
  new result2 = _ZX_32(MEM.add(address, 0xF, coherency << 32, argument2
      .32[0], registerT));
stage SR:
  GPR[registerT] = result2;
```


Kalray – all rights reserved

Encoding Format LSU_ALADDSB.D (Section 6.12.181), lsucode6 = 0110



Syntax `aladdw`*coherency extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. This instruction implements atomic load-add-store on word. The word at effective address is added with the *registerT*, and its previous value is returned into the *registerT*. The effective address must be word aligned (multiple of 4).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR_AUXW.Y (Section 6.14.35)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
  new result2 = _ZX_32(MEM.add(address, 0xF, coherency << 32, argument2
      .32[0], registerT));
stage SR:
  GPR[registerT] = result2;
```


Kalray – all rights reserved

6.5.6 ALCLRD Atomic Load and Clear Double Word

Encoding Format LSU_ALSB (Section 6.12.173), lsucode6 = 0011

P	01	111	coherency	registerW	11	0011	------	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `alclrd`*coherency registerW* = [*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is loaded from the memory double word starting at the effective address. The value 0 is stored into the memory double word starting at the effective address. This atomic instruction implements Test-and-Clear in memory. A zero double word in memory means that the lock at the effective address is taken. Unlocking is achieved by storing a non-zero double word at the effective address.

Modifier(s)

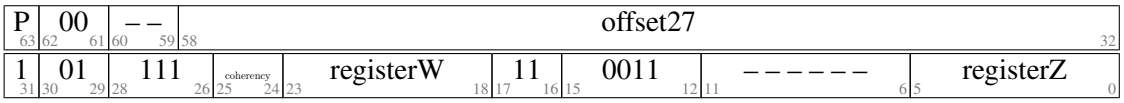
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
  new result1 = MEM.swap(address, 0xFF, coherency << 32, 0, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.O (Section 6.12.174), lsucode6 = 0011



Syntax `alclrd`*coherency registerW* = *offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is loaded from the memory double word starting at the effective address. The value 0 is stored into the memory double word starting at the effective address. This atomic instruction implements Test-and-Clear in memory. A zero double word in memory means that the lock at the effective address is taken. Unlocking is achieved by storing a non-zero double word at the effective address.

Modifier(s)

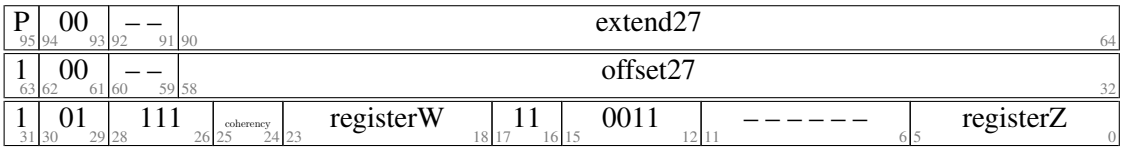
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.swap(address, 0xFF, coherency << 32, 0, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.D (Section 6.12.175), lsucode6 = 0011



Syntax `alclrd`*coherency registerW = extend27_offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is loaded from the memory double word starting at the effective address. The value 0 is stored into the memory double word starting at the effective address. This atomic instruction implements Test-and-Clear in memory. A zero double word in memory means that the lock at the effective address is taken. Unlocking is achieved by storing a non-zero double word at the effective address.

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.swap(address, 0xFF, coherency << 32, 0, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.7 ALCLRW Atomic Load and Clear Word

Encoding Format LSU_ALSB (Section 6.12.173), lsucode6 = 0010

P	01	111	coherency	registerW	11	0010	– – – – – –	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `alclrw`*coherency registerW* = [*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is loaded from the memory word starting at the effective address. The value 0 is stored into the memory word starting at the effective address. This atomic instruction implements Test-and-Clear in memory. A zero word in memory means that the lock at the effective address is taken. Unlocking is achieved by storing a non-zero word at the effective address.

Modifier(s)

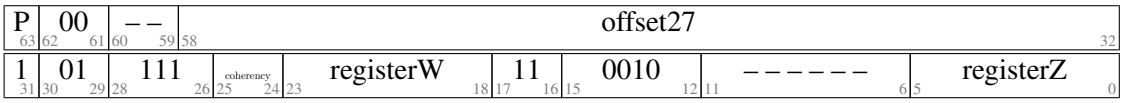
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
  new result1 = _ZX_32(MEM.swap(address, 0xF, coherency << 32, 0, registerW))
      ;
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.O (Section 6.12.174), lsucode6 = 0010



Syntax `alclrw`*coherency registerW* = *offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is loaded from the memory word starting at the effective address. The value 0 is stored into the memory word starting at the effective address. This atomic instruction implements Test-and-Clear in memory. A zero word in memory means that the lock at the effective address is taken. Unlocking is achieved by storing a non-zero word at the effective address.

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_32(MEM.swap(address, 0xF, coherency << 32, 0, registerW))
      ;
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.D (Section 6.12.175), lsucode6 = 0010

P	00	– –	extend27
95	94 93	92 91	90 64
1	00	– –	offset27
63	62 61	60 59	58 32

1	01	111	coherency	registerW	11	0010	– – – – – –	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `alclrw`*coherency registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is loaded from the memory word starting at the effective address. The value 0 is stored into the memory word starting at the effective address. This atomic instruction implements Test-and-Clear in memory. A zero word in memory means that the lock at the effective address is taken. Unlocking is achieved by storing a non-zero word at the effective address.

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

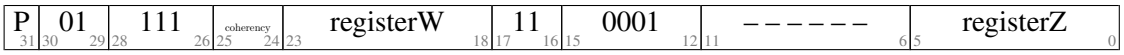
Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_32(MEM.swap(address, 0xF, coherency << 32, 0, registerW))
      ;
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.8 ALD — Atomic Load Double Word

Encoding Format LSU_ALSB (Section 6.12.173), lsucode6 = 0001



Syntax `ald`*coherency registerW* = [*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is atomically loaded from the memory double word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be double word aligned (multiple of 8).

Modifier(s)

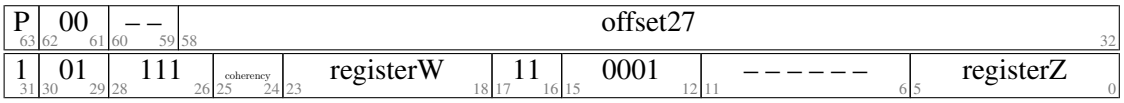
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
  new result1 = MEM.aload(address, 0xFF, coherency << 32, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.O (Section 6.12.174), lsucode6 = 0001



Syntax `ald`*coherency registerW* = *offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is atomically loaded from the memory double word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be double word aligned (multiple of 8).

Modifier(s)

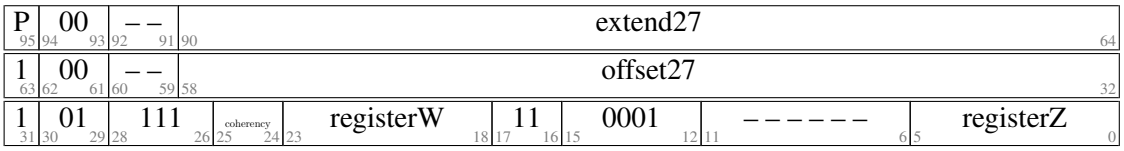
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.aload(address, 0xFF, coherency << 32, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.D (Section 6.12.175), lsucode6 = 0001



Syntax `ald`*coherency registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is atomically loaded from the memory double word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be double word aligned (multiple of 8).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.aload(address, 0xFF, coherency << 32, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.9 ALW Atomic Load Word

Encoding Format LSU_ALSB (Section 6.12.173), lsucode6 = 0000

P	01	111	coherency	registerW	11	0000	------	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `alw`*coherency registerW* = [*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is atomically loaded with zero extension from the memory word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be word aligned (multiple of 4).

Modifier(s)

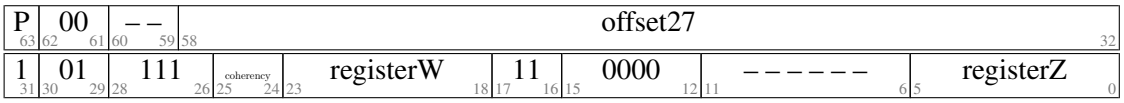
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
  new result1 = _ZX_32(MEM.aload(address, 0xF, coherency << 32, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.O (Section 6.12.174), lsucode6 = 0000



Syntax `alw`*coherency registerW = offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is atomically loaded with zero extension from the memory word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be word aligned (multiple of 4).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_32(MEM.aload(address, 0xF, coherency << 32, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_ALSB.D (Section 6.12.175), lsucode6 = 0000

P 95	00 94 93	– – 92 91	extend27 90 64												
1 63	00 62 61	– – 60 59	offset27 58 32												
1 31	01 30 29	111 28 26	coherency 25 24	registerW 23 18		11 17 16	0000 15 12	– – – – – – 11 6	registerZ 5 0						

Syntax `alw`*coherency registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is computed by adding the *coherency* to the *registerZ*. The *registerW* is atomically loaded with zero extension from the memory word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be word aligned (multiple of 4).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

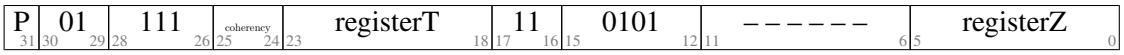
Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_32(MEM.aload(address, 0xF, coherency << 32, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.10 ASD — Atomic Store Double Word

Encoding Format LSU_ASSB (Section 6.12.176), lsucode6 = 0101



Syntax `asd`*coherency* [*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. The *registerT* is atomically stored to the memory double word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be double word aligned (multiple of 8).

Modifier(s)

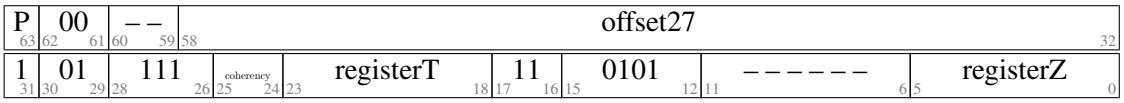
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
stage E1:
  new argument2 = GPR[registerT];
stage E3:
  MEM.astore(address, 0xFF, coherency << 32, argument2, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_ASSB.O (Section 6.12.177), lsucode6 = 0101



Syntax `asd`*coherency offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. The *registerT* is atomically stored to the memory double word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be double word aligned (multiple of 8).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
stage E3:
  MEM.astore(address, 0xFF, coherency << 32, argument2, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_ASSB.D (Section 6.12.178), lsucode6 = 0101

P 95	00 94 93	– – 92 91	extend27 90 64												
1 63	00 62 61	– – 60 59	offset27 58 32												
1 31	01 30 29	111 28 26	coherency 25 24	registerT 23 18		11 17 16	0101 15 12	– – – – – – 11 6	registerZ 5 0						

Syntax `asd`*coherency extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. The *registerT* is atomically stored to the memory double word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be double word aligned (multiple of 8).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
stage E3:
  MEM.astore(address, 0xFF, coherency << 32, argument2, registerT);
```


Kalray – all rights reserved

6.5.11 ASW Atomic Store Word

Encoding Format LSU_ASSB (Section 6.12.176), lsucode6 = 0100

P	01	111	coherency	registerT	11	0100	------	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `asw`*coherency* [*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. The *registerT* is atomically stored to the memory word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be word aligned (multiple of 4).

Modifier(s)

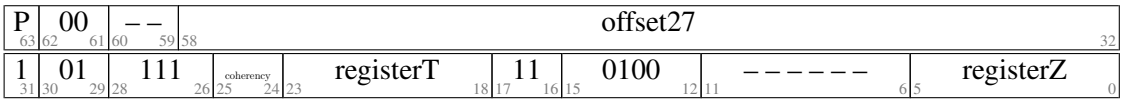
- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new coherency = _ZX_2(coherency);
stage RR:
  new address = GPR[registerZ];
stage E1:
  new argument2 = GPR[registerT];
stage E3:
  MEM.astore(address, 0xF, coherency << 32, argument2, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_ASSB.O (Section 6.12.177), lsucode6 = 0100



Syntax `aswcoherency` *offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. The *registerT* is atomically stored to the memory word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be word aligned (multiple of 4).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_27(offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
stage E3:
  MEM.astore(address, 0xF, coherency << 32, argument2, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_ASSB.D (Section 6.12.178), lsucode6 = 0100

P 95	00 94 93	– – 92 91	extend27 90 64										
1 63	00 62 61	– – 60 59	offset27 58 32										
1 31	01 30 29	111 28 26	coherency 25 24	registerT 23 18	11 17 16	0100 15 12	– – – – – – 11 6	registerZ 5 0					

Syntax `asw`*coherency extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerT* to the *registerZ*. The *registerT* is atomically stored to the memory word starting at the effective address. This instruction is provided to directly operate on the last level of the memory hierarchy, which is typically a processor DDR or a cluster TCM, irrespective of the MMU DCP (Data Cache Policy) settings. The effective address must be word aligned (multiple of 4).

Modifier(s)

- coherency (cf. coherency in section 6.10.2 on page 1416)

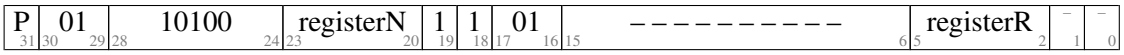
Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new coherency = _ZX_2(coherency);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument2 = GPR[registerT];
stage E3:
  MEM.astore(address, 0xF, coherency << 32, argument2, registerT);
```


Kalray – all rights reserved

6.5.12 COPYO — Copy Octuple Word

Encoding Format LSU_COPYO (Section 6.12.98), lsucode0 = 10100



Syntax `copyo` *registerN* = *registerR*

Description The *registerN* operand receives the *registerR* value.

Execution Reservation table LSU_AUXR_AUXW (Section 6.14.33)

```
stage RR:
  new argument2 = QGR[registerR];
  new result1 = argument2;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

6.5.13 D1INVAL Data Cache 1 Invalidate

Encoding Format LSU_MCC (Section 6.12.78), lsucode2 = 1000

P	01	111	00	1000	1	1	00	----------
31	30 29	28 26	25 24	23 20	19	18	17 16	15 0

Syntax `d1inval`

Description Request to invalidate the data cache. Cache contents is invalidated even if it is modified.

Execution Reservation table LSU (Section 6.14.27)

```
stage E3:
  MEM.d1inval();
```


Kalray – all rights reserved

6.5.14 DFLUSHL — Data Cache Flush Line

Encoding Format LSU_FXBO (Section 6.12.75), lsucode2 = 0011

P	01	111	00	0011	1	1	00	signed10	registerZ
31	30 29	28 26	25 24	23 20	19	18	17 16	15 6	5 0

Syntax `dflushl` *signed10*[*registerZ*]

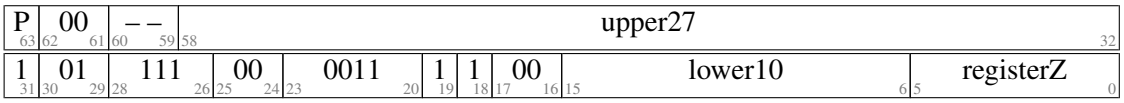
Description The effective address is computed by adding the *signed10* to the *registerZ*. The effective address is sent to the data cache, with request to evict the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dflushl(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.X (Section 6.12.76), lsucode2 = 0011



Syntax `dflushl` *upper27_lower10*[*registerZ*]

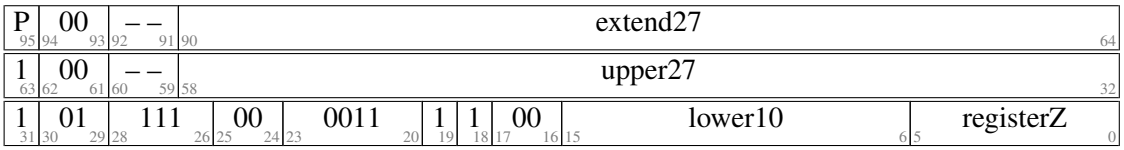
Description The effective address is computed by adding the *upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to evict the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dflushl(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.Y (Section 6.12.77), lsucode2 = 0011



Syntax `dflushl` *extend27_upper27_lower10*[*registerZ*]

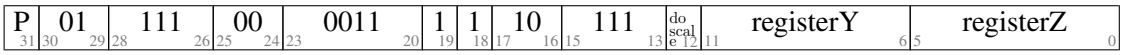
Description The effective address is computed by adding the *extend27_upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to evict the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dflushl(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBI (Section 6.12.123), lsucode2 = 0011



Syntax `dflushl`*doscale registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY*. The effective address is sent to the data cache, with request to evict the corresponding cache line. Has no effects if the line is not cached.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

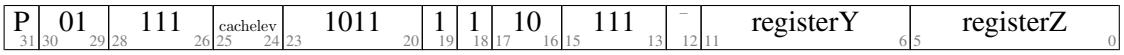
Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 64 : 1;
  new address = base + (index * scaling);
stage E3:
  MEM.dflushl(address);
```


Kalray – all rights reserved

6.5.15 DFLUSHSW — Data Cache Flush Line by Set and Way

Encoding Format LSU_FXSW (Section 6.12.124), lsucode2 = 1011



Syntax `dflushsw`*cachelev registerY*, *registerZ*

Description The set and way are sent to the data cache, with request to evict the corresponding cache line. Has no effects if the line is not cached.

Modifier(s)

- cachelev (cf. cachelev in section 6.10.2 on page 1414)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new cachelev = _ZX_2(cachelev);
stage RR:
  new way = GPR[registerZ];
  new set = GPR[registerY];
stage E3:
  MEM.dflushlsw(set, way, cachelev);
```


Kalray – all rights reserved

6.5.16 DINVALL Data Cache Invalidate Line

Encoding Format LSU_FXBO (Section 6.12.75), lsucode2 = 0001

P	01	111	00	0001	1	1	00	signed10	registerZ
31	30 29	28 26	25 24	23 20	19	18	17 16	15 6	5 0

Syntax `dinvall` *signed10*[*registerZ*]

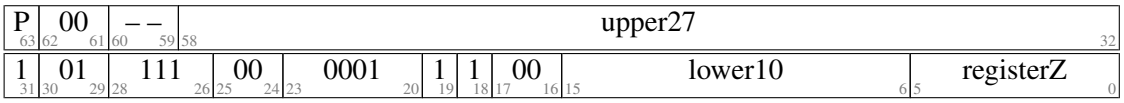
Description The effective address is computed by adding the *signed10* to the *registerZ*. The effective address is sent to the data cache, with request to invalidate the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dinvall(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.X (Section 6.12.76), lsucode2 = 0001



Syntax `dinvall` *upper27_lower10*[*registerZ*]

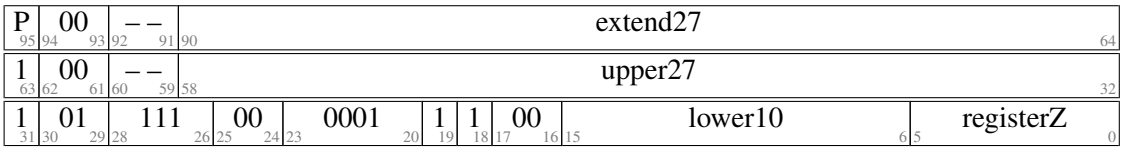
Description The effective address is computed by adding the *upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to invalidate the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dinvall(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.Y (Section 6.12.77), lsucode2 = 0001



Syntax `dinvall` *extend27_upper27_lower10*[*registerZ*]

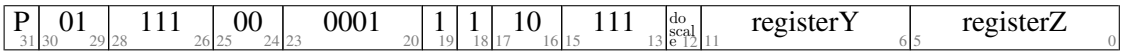
Description The effective address is computed by adding the *extend27_upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to invalidate the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dinvall(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBI (Section 6.12.123), lsucode2 = 0001



Syntax `dinvall`*doscale registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY*. The effective address is sent to the data cache, with request to invalidate the corresponding cache line. Has no effects if the line is not cached.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

Execution Reservation table LSU (Section 6.14.27)

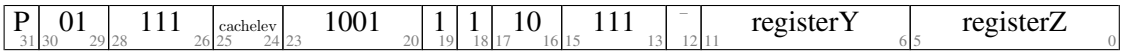
```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 64 : 1;
  new address = base + (index * scaling);
stage E3:
  MEM.dinvall(address);
```


Kalray – all rights reserved

©2025 Kalray

6.5.17 DINVALSW — Data Cache Invalidate Line by Set and Way

Encoding Format LSU_FXSW (Section 6.12.124), lsucode2 = 1001



Syntax `dinvalsw`*cachelev registerY*, *registerZ*

Description The set and way are sent to the data cache, with request to invalidate the corresponding cache line. Has no effects if the line is not cached.

Modifier(s)

- cachelev (cf. cachelev in section 6.10.2 on page 1414)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new cachelev = _ZX_2(cachelev);
stage RR:
  new way = GPR[registerZ];
  new set = GPR[registerY];
stage E3:
  if (!dinvallsw_owner()) {
    _THROW(PRIVILEGE);
  } else {
    MEM.dinvallsw(set, way, cachelev);
  }
```


Kalray – all rights reserved

6.5.18 DPURGEL — Data Cache Purge Line

Encoding Format LSU_FXBO (Section 6.12.75), lsucode2 = 0010

P	01	111	00	0010	1	1	00	signed10	registerZ
31	30 29	28 26	25 24	23 20	19	18	17 16	15 6	5 0

Syntax `dpurgel` *signed10*[*registerZ*]

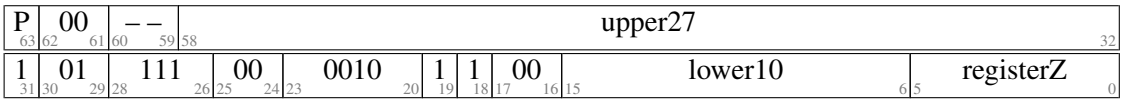
Description The effective address is computed by adding the *signed10* to the *registerZ*. The effective address is sent to the data cache, with request to purge the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dpurgel(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.X (Section 6.12.76), lsucode2 = 0010



Syntax `dpurgel` *upper27_lower10*[*registerZ*]

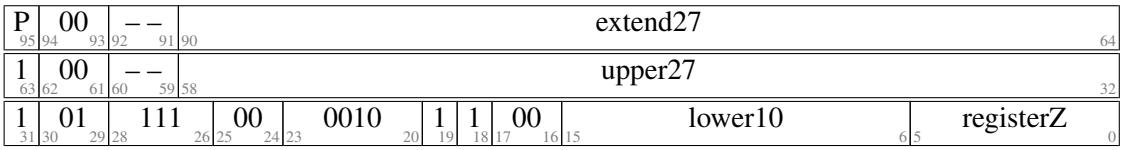
Description The effective address is computed by adding the *upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to purge the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dpurgel(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.Y (Section 6.12.77), lsucode2 = 0010



Syntax `dpurgel` *extend27_upper27_lower10*[*registerZ*]

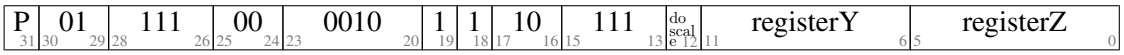
Description The effective address is computed by adding the *extend27_upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to purge the corresponding cache line. Has no effects if the line is not cached.

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dpurgel(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBI (Section 6.12.123), lsucode2 = 0010



Syntax `dpurgel`*doscale registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY*. The effective address is sent to the data cache, with request to purge the corresponding cache line. Has no effects if the line is not cached.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

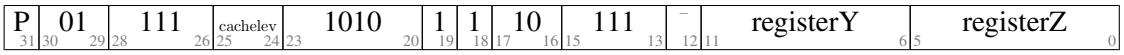
Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 64 : 1;
  new address = base + (index * scaling);
stage E3:
  MEM.dpurgel(address);
```


Kalray – all rights reserved

6.5.19 DPURGESW Data Cache Purge Line by Set and Way

Encoding Format LSU_FXSW (Section 6.12.124), lsucode2 = 1010



Syntax `dpurgesw`*cachelev registerY*, *registerZ*

Description The set and way are sent to the data cache, with request to purge the corresponding cache line. Has no effects if the line is not cached.

Modifier(s)

- cachelev (cf. cachelev in section 6.10.2 on page 1414)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new cachelev = _ZX_2(cachelev);
stage RR:
  new way = GPR[registerZ];
  new set = GPR[registerY];
stage E3:
  MEM.dpurgelsw(set, way, cachelev);
```


Kalray – all rights reserved

6.5.20 DTOUCHL Data Cache Touch Line

Encoding Format LSU_FXBO (Section 6.12.75), lsucode2 = 0000

P	01	111	00	0000	1	1	00	signed10	registerZ
31	30 29	28 26	25 24	23 20	19	18	17 16	15 6	5 0

Syntax `dtouchl` *signed10*[*registerZ*]

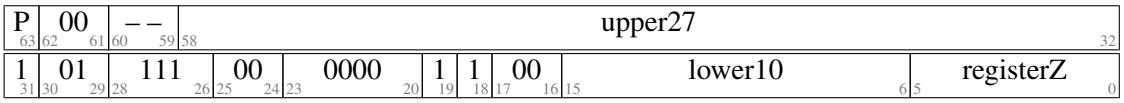
Description The effective address is computed by adding the *signed10* to the *registerZ*. The effective address is sent to the data cache, with request to prefetch the corresponding cache line.

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dtouchl(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.X (Section 6.12.76), lsucode2 = 0000



Syntax `dtouchl` *upper27_lower10*[*registerZ*]

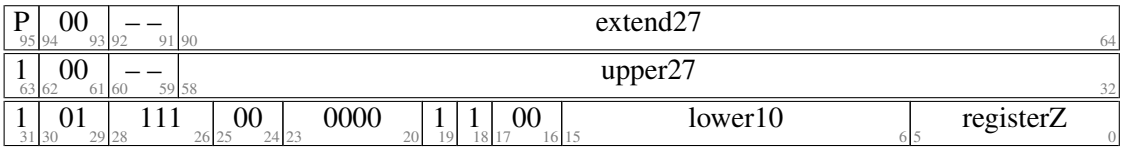
Description The effective address is computed by adding the *upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to prefetch the corresponding cache line.

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dtouchl(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.Y (Section 6.12.77), lsucode2 = 0000



Syntax `dtouchl` *extend27_upper27_lower10*[*registerZ*]

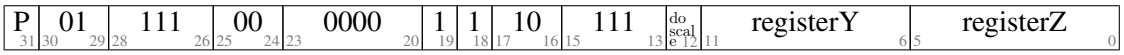
Description The effective address is computed by adding the *extend27_upper27_lower10* to the *registerZ*. The effective address is sent to the data cache, with request to prefetch the corresponding cache line.

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.dtouchl(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBI (Section 6.12.123), lsucode2 = 0000



Syntax `dtouchl`*doscale registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY*. The effective address is sent to the data cache, with request to prefetch the corresponding cache line.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

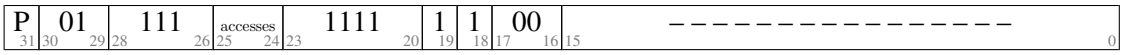
Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 64 : 1;
  new address = base + (index * scaling);
stage E3:
  MEM.dtouchl(address);
```


Kalray – all rights reserved

6.5.21 FENCE — Memory Fence

Encoding Format LSU_FENCE (Section 6.12.79), lsucode2 = 1111



Syntax `fence`*accesses*

Description Ensures that all issued memory accesses are committed to memory. The *accesses* specifies for what types of access. FENCE.R instructions cannot pass earlier load instructions. FENCE.W instructions cannot pass earlier store, atomic and cache instructions. FENCE instructions produces the effects of FENCE.R, FENCE.W and in addition wait for the completion of outstanding prefetch instructions.

Modifier(s)

- accesses (cf. accesses in section 6.10.2 on page 1412)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new accesses = _ZX_2(accesses);
stage E3:
  MEM.fence(accesses);
```


Kalray – all rights reserved

6.5.22 I1INVAL — Instruction Cache 1 Invalidate

Encoding Format LSU_MCC (Section 6.12.78), lsucode2 = 1100

P	01	111	00	1100	1	1	00	– – – – – – – – – – – – – – – –
31	30 29	28 26	25 24	23 20	19	18	17 16	15 0

Syntax `i1inval`

Description Request to invalidate the instruction cache.

Execution Reservation table LSU (Section 6.14.27)

```
stage E3:
  MEM.i1inval();
```


Kalray – all rights reserved

6.5.23 I1INVALS Instruction Cache 1 Invalidate Set

Encoding Format LSU_FXBO (Section 6.12.75), lsucode2 = 0101

P	01	111	00	0101	1	1	00	signed10	registerZ
31	30 29	28 26	25 24	23 20	19	18	17 16	15 6	5 0

Syntax `i1invals` *signed10*[*registerZ*]

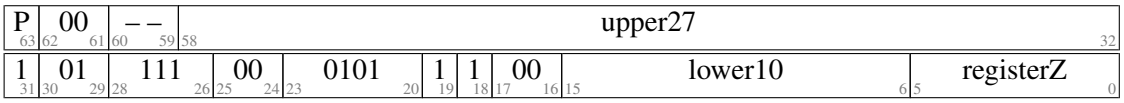
Description The effective address is computed by adding the *signed10* to the *registerZ*. The effective address is sent to the instruction cache, with request to invalidate the corresponding L1 cache set. (All the cache lines with the same index.)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.i1invals(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.X (Section 6.12.76), lsucode2 = 0101



Syntax `i1invals` *upper27_lower10*[*registerZ*]

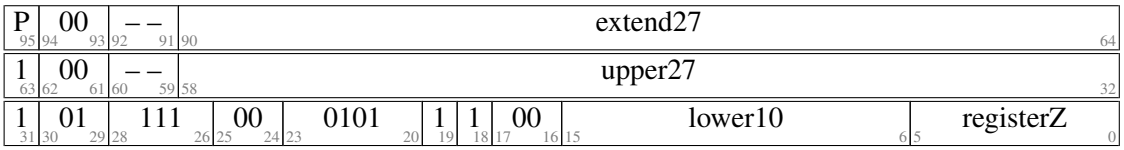
Description The effective address is computed by adding the *upper27_lower10* to the *registerZ*. The effective address is sent to the instruction cache, with request to invalidate the corresponding L1 cache set. (All the cache lines with the same index.)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.i1invals(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBO.Y (Section 6.12.77), lsucode2 = 0101



Syntax `i1invals` *extend27_upper27_lower10*[*registerZ*]

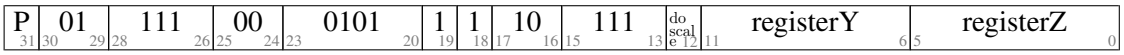
Description The effective address is computed by adding the *extend27_upper27_lower10* to the *registerZ*. The effective address is sent to the instruction cache, with request to invalidate the corresponding L1 cache set. (All the cache lines with the same index.)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E3:
  MEM.i1invals(address);
```


Kalray – all rights reserved

Encoding Format LSU_FXBI (Section 6.12.123), lsucode2 = 0101



Syntax `i1invals`*doscale registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY*. The effective address is sent to the instruction cache, with request to invalidate the corresponding L1 cache set. (All the cache lines with the same index.)

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

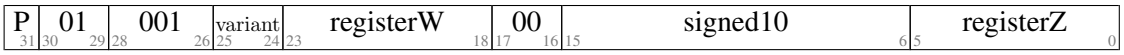
Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 64 : 1;
  new address = base + (index * scaling);
stage E3:
  MEM.i1invals(address);
```


Kalray – all rights reserved

6.5.24 LBS Load Byte Sign Extended

Encoding Format LSU_LSBO (Section 6.12.66), loadcode = 001



Syntax `lbs`*variant registerW* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerW* is loaded from the memory byte at the effective address after sign extension.

Modifier(s)

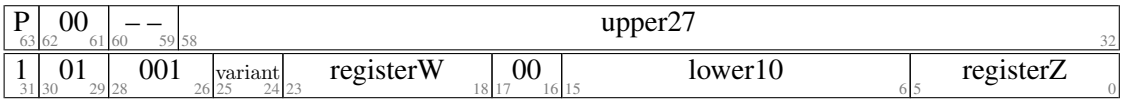
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.X (Section 6.12.67), loadcode = 001



Syntax `lbs`*variant registerW = upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerW* is loaded from the memory byte at the effective address after sign extension.

Modifier(s)

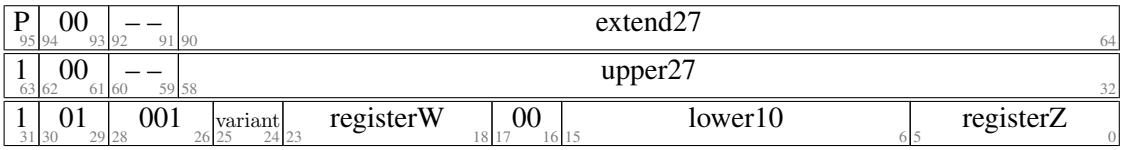
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.Y (Section 6.12.68), loadcode = 001



Syntax `lbs`*variant registerW = extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerW* is loaded from the memory byte at the effective address after sign extension.

Modifier(s)

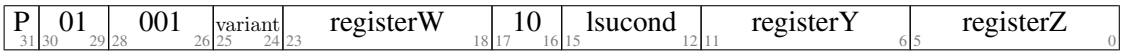
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSPB (Section 6.12.108), loadcode = 001



Syntax `lbs`*variantlsucond registerY*? *registerW* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory byte at the effective address after sign extension.

Modifier(s)

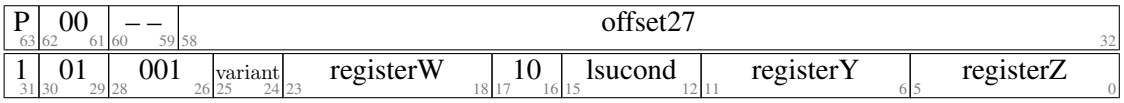
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_8(MEM.load(address, 0x1, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.O (Section 6.12.109), loadcode = 001



Syntax `lbs`*variantlsucond registerY*? *registerW* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory byte at the effective address after sign extension.

Modifier(s)

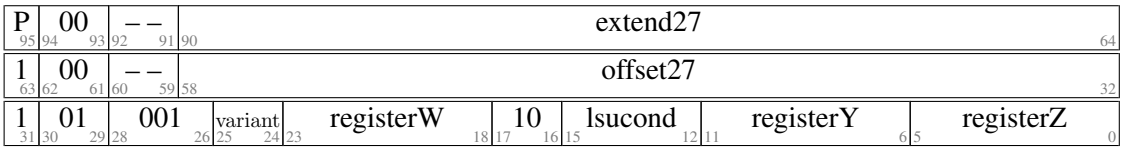
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_8(MEM.load(address, 0x1, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.D (Section 6.12.110), loadcode = 001



Syntax `lbs`*variantlsucond registerY*? *registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory byte at the effective address after sign extension.

Modifier(s)

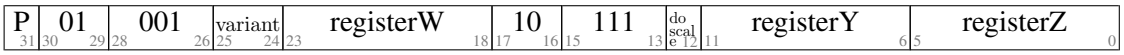
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_8(MEM.load(address, 0x1, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSBI (Section 6.12.120), loadcode = 001



Syntax `lbs`*variantdoscale registerW* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerW* is loaded from the memory byte at the effective address after sign extension.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

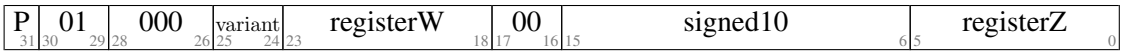
Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 1 : 1;
  new address = base + (index * scaling);
  new result1 = _SX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.25 LBZ — Load Byte Zero Extended

Encoding Format LSU_LSBO (Section 6.12.66), loadcode = 000



Syntax `lbz`*variant registerW* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerW* is loaded from the memory byte at the effective address after zero extension.

Modifier(s)

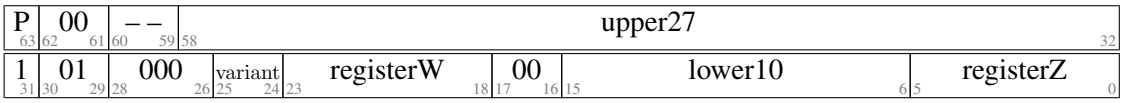
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.X (Section 6.12.67), loadcode = 000



Syntax `lbz`*variant registerW = upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerW* is loaded from the memory byte at the effective address after zero extension.

Modifier(s)

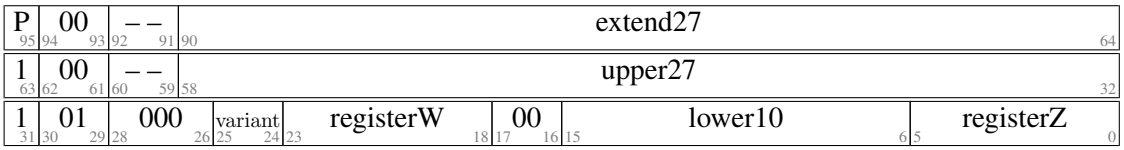
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.Y (Section 6.12.68), loadcode = 000



Syntax `lbz`*variant registerW = extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerW* is loaded from the memory byte at the effective address after zero extension.

Modifier(s)

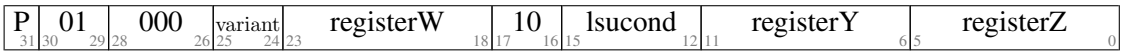
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSPB (Section 6.12.108), loadcode = 000



Syntax `lbz`*variantlsucond registerY*? *registerW* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory byte at the effective address after zero extension.

Modifier(s)

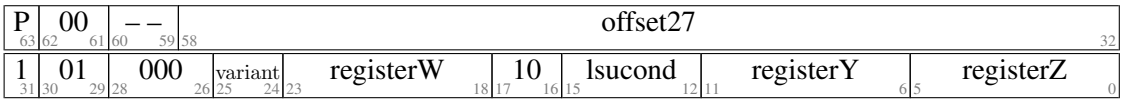
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_8(MEM.load(address, 0x1, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.O (Section 6.12.109), loadcode = 000



Syntax `lbz`*variantlsucond registerY*? *registerW* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory byte at the effective address after zero extension.

Modifier(s)

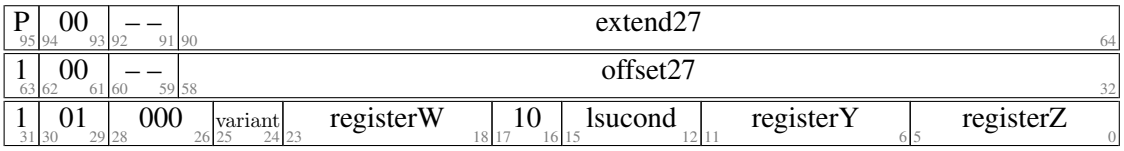
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_8(MEM.load(address, 0x1, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.D (Section 6.12.110), loadcode = 000



Syntax `lbz`*variantlsucond registerY*? *registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory byte at the effective address after zero extension.

Modifier(s)

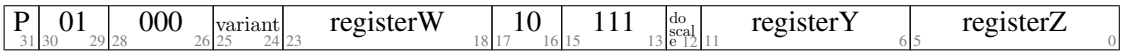
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_8(MEM.load(address, 0x1, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSBI (Section 6.12.120), loadcode = 000



Syntax `lbz`*variantdoscale registerW* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerW* is loaded from the memory byte at the effective address after zero extension.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 1 : 1;
  new address = base + (index * scaling);
  new result1 = _ZX_8(MEM.load(address, 0x1, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.26 LD — Load Double Word

Encoding Format LSU_LSBO (Section 6.12.66), loadcode = 110

P	01	110	variant	registerW	00	signed10	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 6	5 0

Syntax `ld`*variant registerW* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerW* is loaded from the memory double word at the effective address.

Modifier(s)

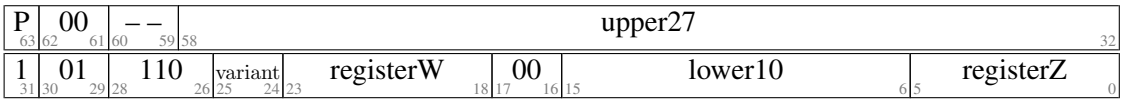
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFF, variant, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.X (Section 6.12.67), loadcode = 110



Syntax `ld`*variant registerW* = *upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerW* is loaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFF, variant, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.Y (Section 6.12.68), loadcode = 110

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	01 30 29	110 28 26	variant 25 24	registerW 23 18	00 17 16	lower10 15 6	registerZ 5 0

Syntax `ld`*variant registerW* = *extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerW* is loaded from the memory double word at the effective address.

Modifier(s)

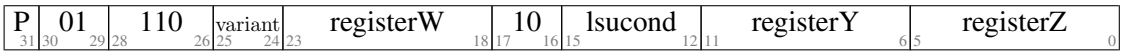
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFF, variant, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSPB (Section 6.12.108), loadcode = 110



Syntax `ld`*variantlsucond registerY*? *registerW* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory double word at the effective address.

Modifier(s)

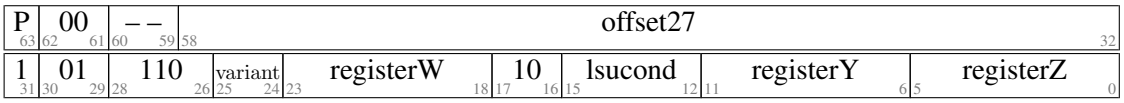
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFF, variant, registerW);
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.O (Section 6.12.109), loadcode = 110



Syntax `ld`*variantlsucond registerY*? *registerW* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory double word at the effective address.

Modifier(s)

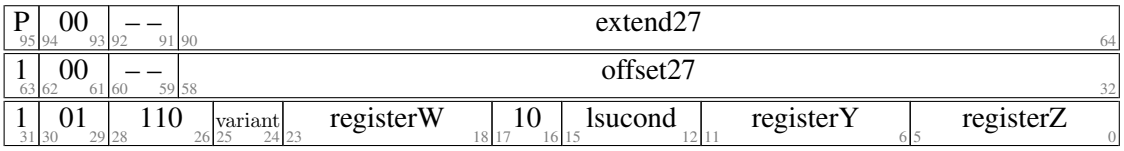
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFF, variant, registerW);
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.D (Section 6.12.110), loadcode = 110



Syntax `ld`*variantlsucond registerY*? *registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory double word at the effective address.

Modifier(s)

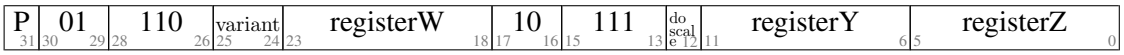
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFF, variant, registerW);
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSBI (Section 6.12.120), loadcode = 110



Syntax `ld`*variantdoscale registerW* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerW* is loaded from the memory double word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 8 : 1;
  new address = base + (index * scaling);
  new result1 = MEM.load(address, 0xFF, variant, registerW);
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.27 LHS — Load Half Word Sign Extended

Encoding Format LSU_LSBO (Section 6.12.66), loadcode = 011

P	01	011	variant	registerW	00	signed10	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 6	5 0

Syntax `lhs`*variant registerW* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerW* is loaded from the memory half word at the effective address after sign extension.

Modifier(s)

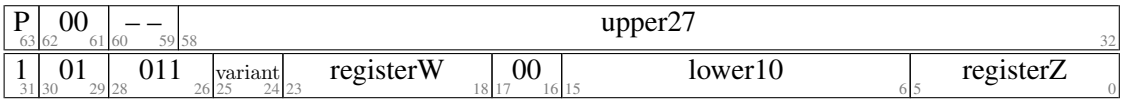
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.X (Section 6.12.67), loadcode = 011



Syntax `lhs`*variant registerW* = *upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerW* is loaded from the memory half word at the effective address after sign extension.

Modifier(s)

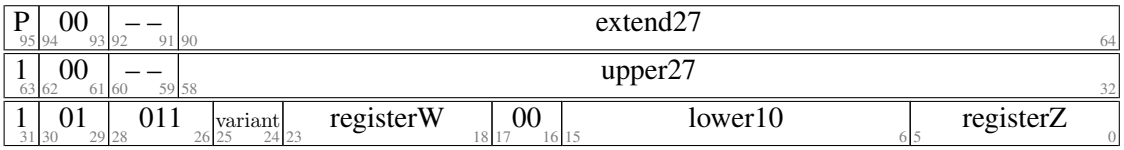
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.Y (Section 6.12.68), loadcode = 011



Syntax `lhs`*variant registerW = extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerW* is loaded from the memory half word at the effective address after sign extension.

Modifier(s)

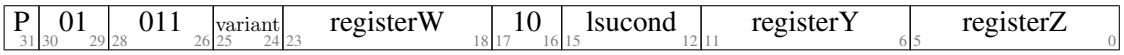
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSPB (Section 6.12.108), loadcode = 011



Syntax `lhs`*variantlsucond registerY*? *registerW* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory half word at the effective address after sign extension.

Modifier(s)

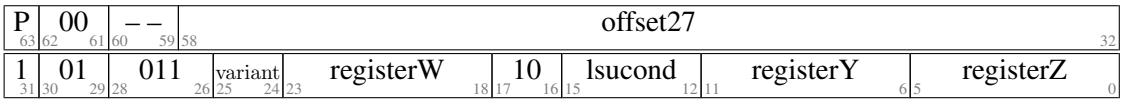
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_16(MEM.load(address, 0x3, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.O (Section 6.12.109), loadcode = 011



Syntax `lhs`*variantlsucond registerY*? *registerW* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory half word at the effective address after sign extension.

Modifier(s)

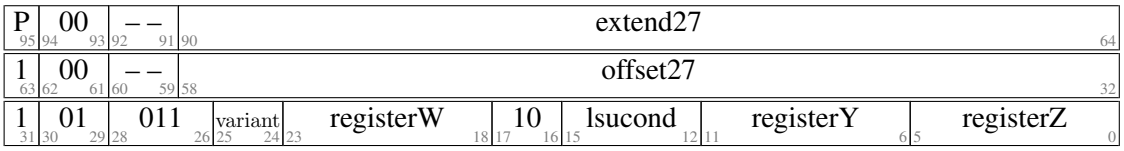
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_16(MEM.load(address, 0x3, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.D (Section 6.12.110), loadcode = 011



Syntax `lhs`*variantlsucond registerY*? *registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory half word at the effective address after sign extension.

Modifier(s)

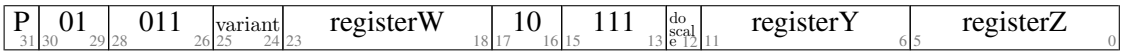
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_16(MEM.load(address, 0x3, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSBI (Section 6.12.120), loadcode = 011



Syntax `lhs`*variantdoscale registerW* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerW* is loaded from the memory half word at the effective address after sign extension.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 2 : 1;
  new address = base + (index * scaling);
  new result1 = _SX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.28 LHZ Load Half Word Zero Extended

Encoding Format LSU_LSBO (Section 6.12.66), loadcode = 010

P	01	010	variant	registerW	00	signed10	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 6	5 0

Syntax `lhz`*variant registerW* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerW* is loaded from the memory half word at the effective address after zero extension.

Modifier(s)

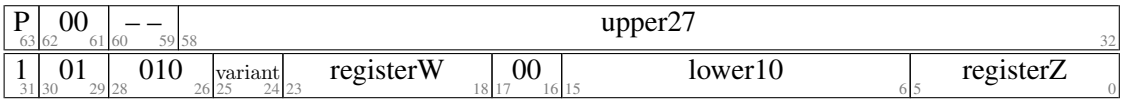
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.X (Section 6.12.67), loadcode = 010



Syntax `lhz`*variant registerW = upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerW* is loaded from the memory half word at the effective address after zero extension.

Modifier(s)

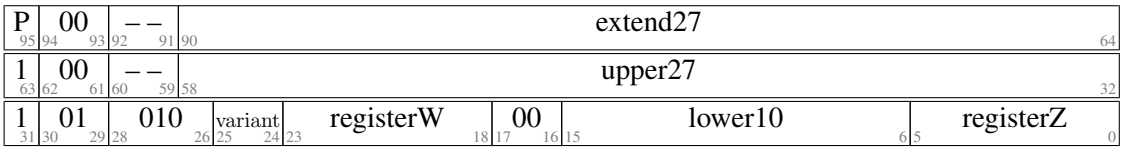
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.Y (Section 6.12.68), loadcode = 010



Syntax `lhz`*variant registerW = extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerW* is loaded from the memory half word at the effective address after zero extension.

Modifier(s)

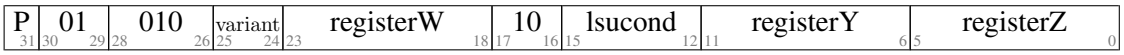
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSPB (Section 6.12.108), loadcode = 010



Syntax `lhz`*variantlsucond registerY*? *registerW* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory half word at the effective address after zero extension.

Modifier(s)

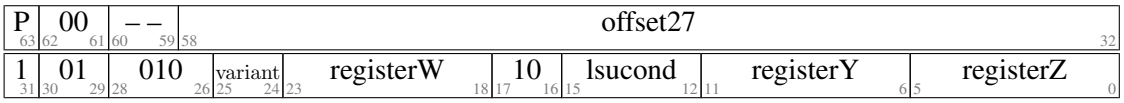
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_16(MEM.load(address, 0x3, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.O (Section 6.12.109), loadcode = 010



Syntax `lhz`*variantlsucond registerY*? *registerW* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory half word at the effective address after zero extension.

Modifier(s)

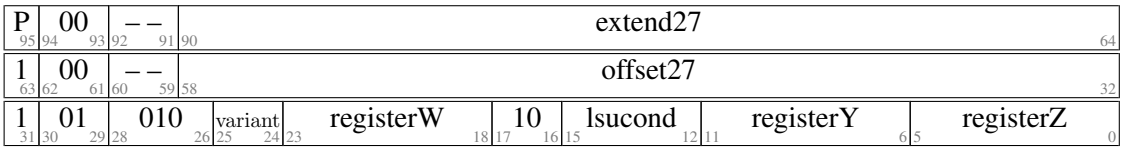
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_16(MEM.load(address, 0x3, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.D (Section 6.12.110), loadcode = 010



Syntax `lhz`*variantlsucond registerY*? *registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory half word at the effective address after zero extension.

Modifier(s)

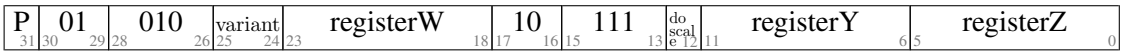
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_16(MEM.load(address, 0x3, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSBI (Section 6.12.120), loadcode = 010



Syntax `lhz`*variantdoscale registerW* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerW* is loaded from the memory half word at the effective address after zero extension.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

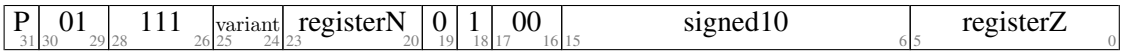
Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 2 : 1;
  new address = base + (index * scaling);
  new result1 = _ZX_16(MEM.load(address, 0x3, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.29 LO — Load Octuple Word

Encoding Format LSU_LOBO (Section 6.12.72), loadcode = 111



Syntax `lo`*variant registerN* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

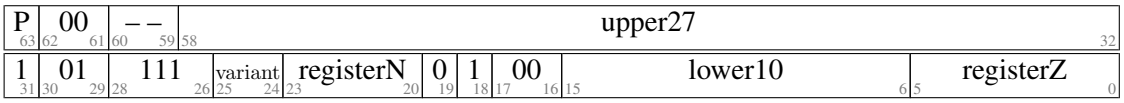
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
      << 2);
stage CR:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LOBO.X (Section 6.12.73), loadcode = 111



Syntax `lo`*variant registerN = upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

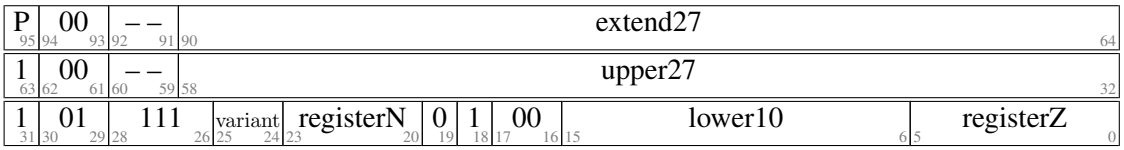
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
      << 2);
stage CR:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LOBO.Y (Section 6.12.74), loadcode = 111



Syntax `lo`*variant registerN = extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

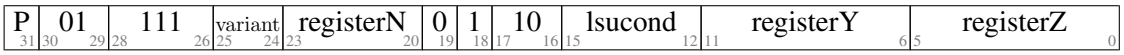
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
      << 2);
stage CR:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LOPB (Section 6.12.114), loadcode = 111



Syntax `lo`*variantlsucond registerY*? *registerN* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

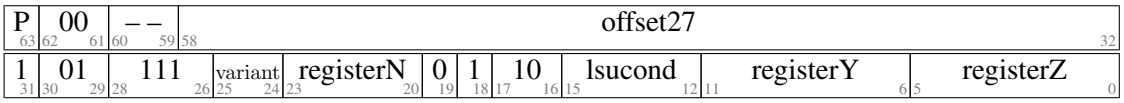
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new address = GPR[registerZ];
  new argument2 = GPR[registerY];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
        << 2);
    QGR[registerN] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LOPB.O (Section 6.12.115), loadcode = 111



Syntax `lo`*variant**lsucond** registerY*? *registerN* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

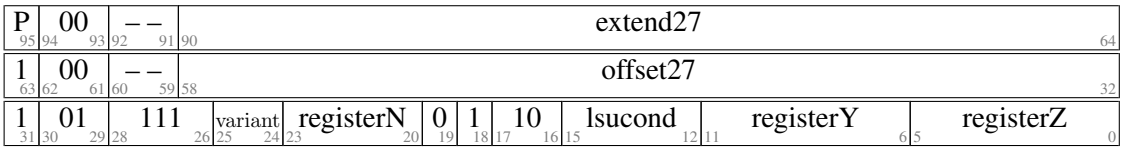
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
        << 2);
    QGR[registerN] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LOPB.D (Section 6.12.116), loadcode = 111



Syntax `lo`*variantlsucond registerY*? *registerN* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

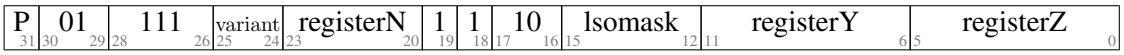
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
        << 2);
    QGR[registerN] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LOMB (Section 6.12.117), loadcode = 111



Syntax `lo`*variantlsomask registerY*? *registerN* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsomask*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

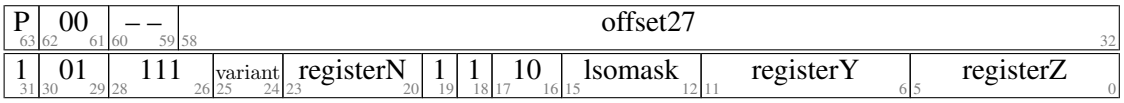
- lsomask (cf. lsomask in section 6.10.2 on page 1421)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsomask);
stage RR:
  new address = GPR[registerZ];
  new argument2 = GPR[registerY];
stage E1:
  new argument1 = QGR[registerN];
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
      << 2);
stage CR:
  QGR[registerN] = (result1 & passbytes) | (argument1 & clearbytes);
```


Kalray – all rights reserved

Encoding Format LSU_LOMB.O (Section 6.12.118), loadcode = 111



Syntax `lo`*variantlsomask registerY*? *registerN* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsomask*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

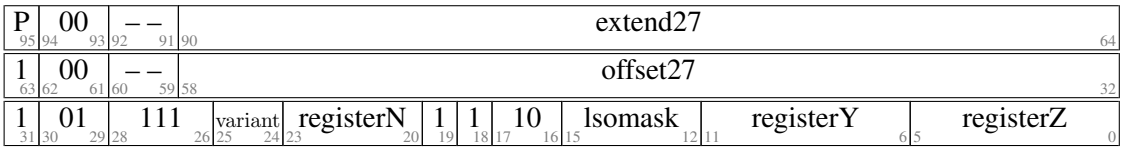
- lsomask (cf. lsomask in section 6.10.2 on page 1421)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsomask);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
stage E1:
  new argument1 = QGR[registerN];
  new address = base + offset;
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
      << 2);
stage CR:
  QGR[registerN] = (result1 & passbytes) | (argument1 & clearbytes);
```


Kalray – all rights reserved

Encoding Format LSU_LOMB.D (Section 6.12.119), loadcode = 111



Syntax `lo`*variantlsomask registerY*? *registerN* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsomask*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

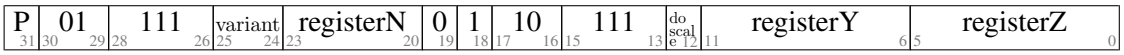
- lsomask (cf. lsomask in section 6.10.2 on page 1421)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsomask);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
stage E1:
  new argument1 = QGR[registerN];
  new address = base + offset;
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
      << 2);
stage CR:
  QGR[registerN] = (result1 & passbytes) | (argument1 & clearbytes);
```


Kalray – all rights reserved

Encoding Format LSU_LOBI (Section 6.12.122), loadcode = 111



Syntax `lo`*variantdoscale registerN* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerN* is loaded from the memory octuple word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

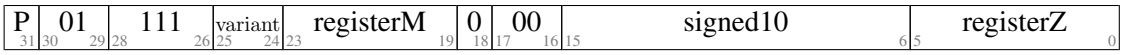
Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 32 : 1;
  new address = base + (index * scaling);
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, registerN
      << 2);
stage CR:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

6.5.30 LQ — Load Quadruple Word

Encoding Format LSU_LQBO (Section 6.12.69), loadcode = 111



Syntax `lq`*variant registerM* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerM* is loaded from the memory quadruple word at the effective address.

Modifier(s)

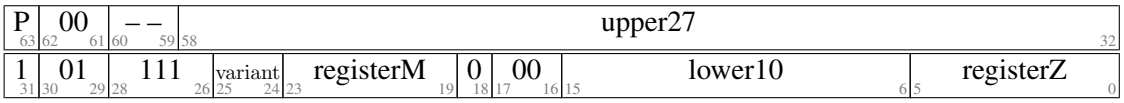
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFFFF, variant, registerM << 1);
stage CR:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LQBO.X (Section 6.12.70), loadcode = 111



Syntax `lq`*variant registerM* = *upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerM* is loaded from the memory quadruple word at the effective address.

Modifier(s)

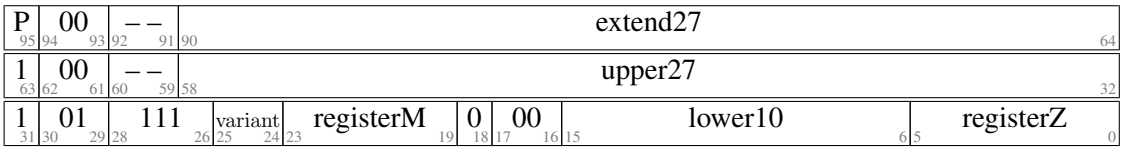
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFFFF, variant, registerM << 1);
stage CR:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LQBO.Y (Section 6.12.71), loadcode = 111



Syntax `lq`*variant registerM* = *extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerM* is loaded from the memory quadruple word at the effective address.

Modifier(s)

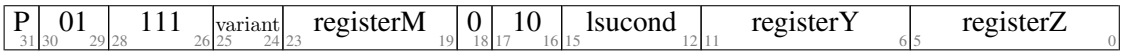
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = MEM.load(address, 0xFFFF, variant, registerM << 1);
stage CR:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LQPB (Section 6.12.111), loadcode = 111



Syntax `lq`*variantlsucond registerY*? *registerM* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerM* is loaded from the memory quadruple word at the effective address.

Modifier(s)

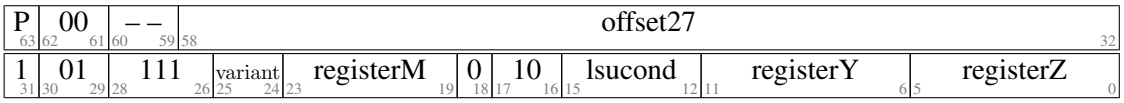
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFFFF, variant, registerM << 1);
    PGR[registerM] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LQPB.O (Section 6.12.112), loadcode = 111



Syntax `lq`*variantlsucond registerY*? *registerM* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerM* is loaded from the memory quadruple word at the effective address.

Modifier(s)

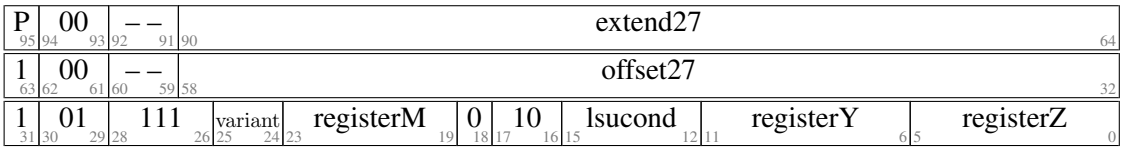
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFFFF, variant, registerM << 1);
    PGR[registerM] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LQPB.D (Section 6.12.113), loadcode = 111



Syntax `lq`*variantlsucond registerY*? *registerM* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerM* is loaded from the memory quadruple word at the effective address.

Modifier(s)

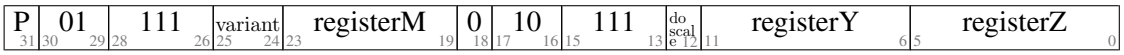
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = MEM.load(address, 0xFFFF, variant, registerM << 1);
    PGR[registerM] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LQBI (Section 6.12.121), loadcode = 111



Syntax `lq`*variantdoscale registerM* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerM* is loaded from the memory quadruple word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

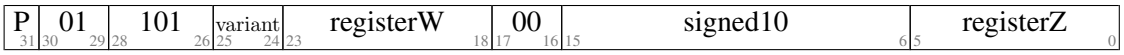
Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 16 : 1;
  new address = base + (index * scaling);
  new result1 = MEM.load(address, 0xFFFF, variant, registerM << 1);
stage CR:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.5.31 LWS Load Word Sign Extended

Encoding Format LSU_LSBO (Section 6.12.66), loadcode = 101



Syntax `lws`*variant registerW* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerW* is loaded from the memory word at the effective address after sign extension.

Modifier(s)

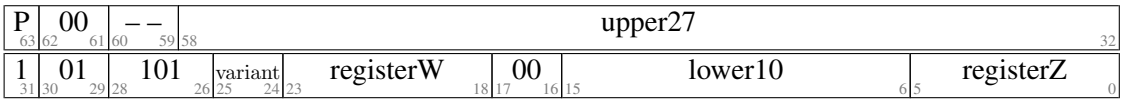
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.X (Section 6.12.67), loadcode = 101



Syntax `lws`*variant registerW* = *upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerW* is loaded from the memory word at the effective address after sign extension.

Modifier(s)

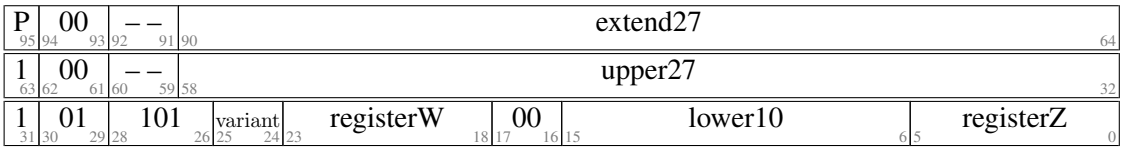
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.Y (Section 6.12.68), loadcode = 101



Syntax `lws`*variant registerW* = *extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerW* is loaded from the memory word at the effective address after sign extension.

Modifier(s)

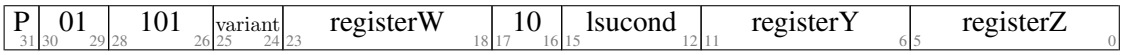
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _SX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSPB (Section 6.12.108), loadcode = 101



Syntax `lws`*variantlsucond registerY*? *registerW* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory word at the effective address after sign extension.

Modifier(s)

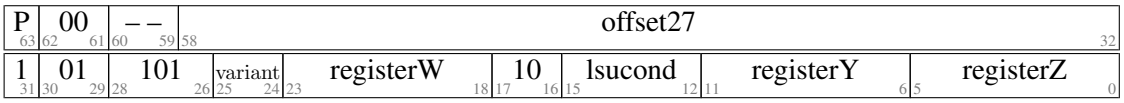
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_32(MEM.load(address, 0xF, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.O (Section 6.12.109), loadcode = 101



Syntax `lws`*variantlsucond registerY*? *registerW* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory word at the effective address after sign extension.

Modifier(s)

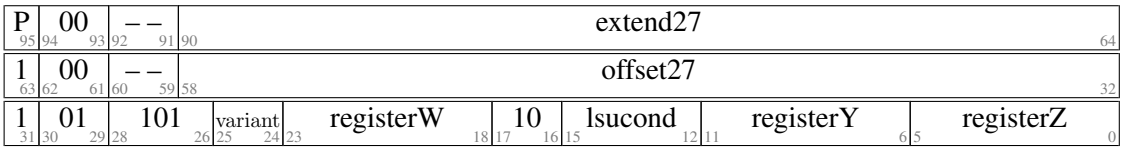
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_32(MEM.load(address, 0xF, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.D (Section 6.12.110), loadcode = 101



Syntax `lws`*variantlsucond registerY*? *registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory word at the effective address after sign extension.

Modifier(s)

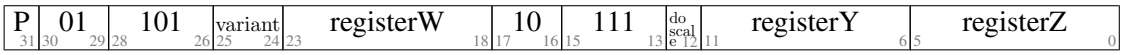
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _SX_32(MEM.load(address, 0xF, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSBI (Section 6.12.120), loadcode = 101



Syntax `lws`*variantdoscale registerW* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerW* is loaded from the memory word at the effective address after sign extension.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

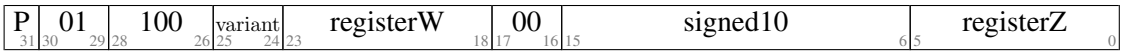
Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 4 : 1;
  new address = base + (index * scaling);
  new result1 = _SX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.32 LWZ — Load Word Zero Extended

Encoding Format LSU_LSBO (Section 6.12.66), loadcode = 100



Syntax `lwz`*variant registerW* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerW* is loaded from the memory word at the effective address after zero extension.

Modifier(s)

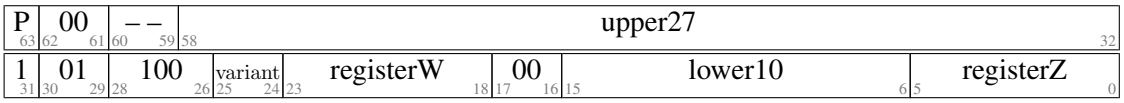
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.X (Section 6.12.67), loadcode = 100



Syntax `lwz`*variant registerW* = *upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerW* is loaded from the memory word at the effective address after zero extension.

Modifier(s)

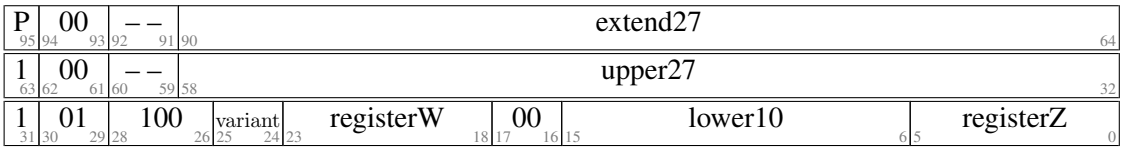
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSBO.Y (Section 6.12.68), loadcode = 100



Syntax `lwz`*variant registerW = extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerW* is loaded from the memory word at the effective address after zero extension.

Modifier(s)

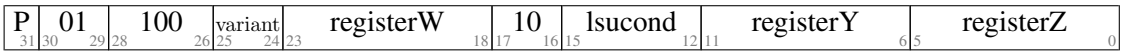
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
  new result1 = _ZX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_LSPB (Section 6.12.108), loadcode = 100



Syntax `lwz`*variantlsucond registerY*? *registerW* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory word at the effective address after zero extension.

Modifier(s)

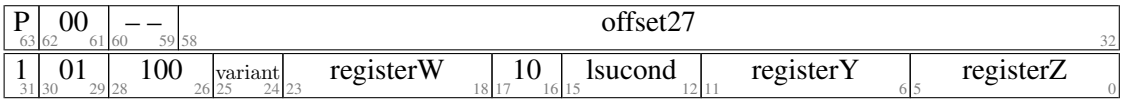
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument2 = GPR[registerY];
  new address = GPR[registerZ];
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_32(MEM.load(address, 0xF, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.O (Section 6.12.109), loadcode = 100



Syntax `lwz`*variantlsucond registerY*? *registerW* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory word at the effective address after zero extension.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.X (Section 6.14.37)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_32(MEM.load(address, 0xF, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSPB.D (Section 6.12.110), loadcode = 100

P 95	00 94 93	– – 92 91	extend27 90 64						
1 63	00 62 61	– – 60 59	offset27 58 32						
1 31	01 30 29	100 28 26	variant 25 24	registerW 23 18	10 17 16	lsucond 15 12	registerY 11 6	registerZ 5 0	

Syntax `lwz`*variantlsucond registerY*? *registerW* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsucond*. The *registerW* is loaded from the memory word at the effective address after zero extension.

Modifier(s)

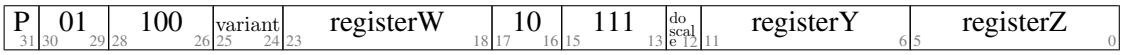
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW.Y (Section 6.14.38)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new argument4 = _ZX_4(lsucond);
stage RR:
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage CR:
  if (scalarcond(argument4, argument2)) {
    new result1 = _ZX_32(MEM.load(address, 0xF, variant, registerW));
    GPR[registerW] = result1;
  }
```


Kalray – all rights reserved

Encoding Format LSU_LSBI (Section 6.12.120), loadcode = 100



Syntax `lwz`*variantdoscale registerW* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerW* is loaded from the memory word at the effective address after zero extension.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU_AUXW (Section 6.14.36)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 4 : 1;
  new address = base + (index * scaling);
  new result1 = _ZX_32(MEM.load(address, 0xF, variant, registerW));
stage CR:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.5.33 SB Store Byte

Encoding Format LSU_SSBO (Section 6.12.89), lsucode0 = 10000

P	01	10000	registerT	01	signed10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

Syntax `sb` *signed10*[*registerZ*] = *registerT*

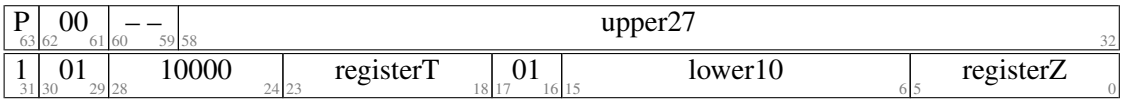
Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerT* is stored into the memory byte at the effective address.

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x1, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.X (Section 6.12.90), lsucode0 = 10000



Syntax `sb` *upper27_lower10*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerT* is stored into the memory byte at the effective address.

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x1, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.Y (Section 6.12.91), lsucode0 = 10000

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	01 30 29	10000 28 24	registerT 23 18	01 17 16	lower10 15 6	registerZ 5 0	

Syntax `sb` *extend27_upper27_lower10*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerT* is stored into the memory byte at the effective address.

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x1, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBI (Section 6.12.191), lsucode0 = 10000

Syntax `sb`*doscale registerY*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerT* is stored into the memory byte at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new index = GPR[registerY];
  new base = GPR[registerZ];
  new scaling = doscale ? 1 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x1, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSPB (Section 6.12.152), lsucode0 = 10000

P	01	10000	registerT	11	lsucond	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

Syntax `sb`*lsucond registerY*? [*registerZ*] = *registerT*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory byte at the effective address.

Modifier(s)

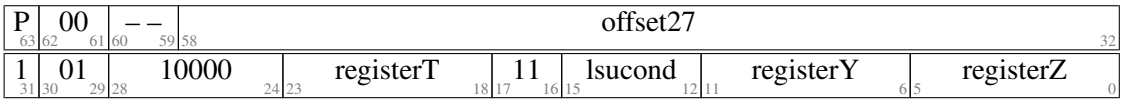
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument1 = GPR[registerY];
  new address = GPR[registerZ];
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0x1, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.O (Section 6.12.153), lsucode0 = 10000



Syntax `sb`*lsucond registerY*? *offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory byte at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_27(offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0x1, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.D (Section 6.12.154), lsucode0 = 10000

P 95	00 94 93	– – 92 91	extend27 90 64												
1 63	00 62 61	– – 60 59	offset27 58 32												
1 31	01 30 29	10000 28 24			registerT 23 18		11 17 16	lsucond 15 12		registerY 11 6		registerZ 5 0			

Syntax `sb`*lsucond registerY*? *extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *extend27_offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory byte at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_54(extend27_offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0x1, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

6.5.34 SD — Store Double Word

Encoding Format LSU_SSBO (Section 6.12.89), lsucode0 = 10011

P	01	10011	registerT	01	signed10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

Syntax `sd` *signed10*[*registerZ*] = *registerT*

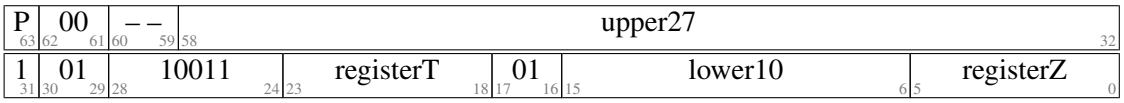
Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerT* is stored into the memory double word at the effective address.

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xFF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.X (Section 6.12.90), lsucode0 = 10011



Syntax `sd` *upper27_lower10*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerT* is stored into the memory double word at the effective address.

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xFF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.Y (Section 6.12.91), lsucode0 = 10011

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	01 30 29	10011 28 24	registerT 23 18	01 17 16	lower10 15 6	registerZ 5 0	

Syntax `sd` *extend27_upper27_lower10*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerT* is stored into the memory double word at the effective address.

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xFF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBI (Section 6.12.191), lsucode0 = 10011

P	01	10011	registerT	11	111	doscale	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 13	12	11 6	5 0

Syntax `sd`*doscale registerY*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerT* is stored into the memory double word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new index = GPR[registerY];
  new base = GPR[registerZ];
  new scaling = doscale ? 8 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xFF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSPB (Section 6.12.152), lsucode0 = 10011

P	01	10011	registerT	11	lsucond	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

Syntax `sd`*lsucond registerY*? [*registerZ*] = *registerT*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory double word at the effective address.

Modifier(s)

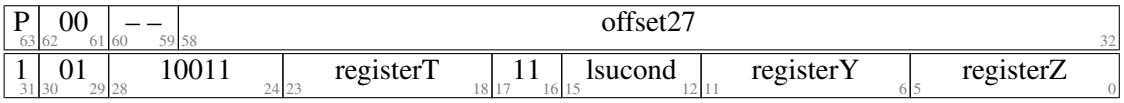
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument1 = GPR[registerY];
  new address = GPR[registerZ];
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xFF, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.O (Section 6.12.153), lsucode0 = 10011



Syntax `sd`*lsucond registerY*? *offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory double word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_27(offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xFF, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.D (Section 6.12.154), lsucode0 = 10011

P	00	– –	extend27												
95	94 93	92 91	90 64												
1	00	– –	offset27												
63	62 61	60 59	58 32												
1	01	10011			registerT			11	lsucond		registerY			registerZ	
31	30 29	28 24	23 18					17 16	15 12		11 6			5 0	

Syntax `sd`*lsucond registerY*? *extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *extend27_offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory double word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_54(extend27_offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xFF, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

6.5.35 SH Store Half Word

Encoding Format LSU_SSBO (Section 6.12.89), lsucode0 = 10001

P	01	10001	registerT	01	signed10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

Syntax `sh` *signed10*[*registerZ*] = *registerT*

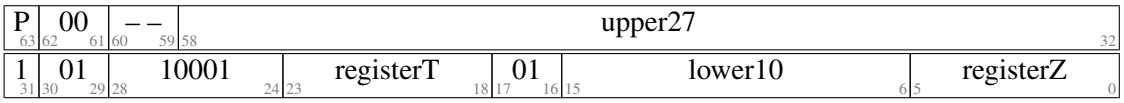
Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerT* is stored into the memory half word at the effective address.

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x3, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.X (Section 6.12.90), lsucode0 = 10001



Syntax `sh` *upper27_lower10*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerT* is stored into the memory half word at the effective address.

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x3, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.Y (Section 6.12.91), lsucode0 = 10001

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	01 30 29	10001 28 24	registerT 23 18	01 17 16	lower10 15 6	registerZ 5 0	

Syntax `sh` *extend27_upper27_lower10*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerT* is stored into the memory half word at the effective address.

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x3, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBI (Section 6.12.191), lsucode0 = 10001

Syntax `sh`*doscale registerY*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerT* is stored into the memory half word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new index = GPR[registerY];
  new base = GPR[registerZ];
  new scaling = doscale ? 2 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0x3, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSPB (Section 6.12.152), lsucode0 = 10001

P	01	10001	registerT	11	lsucond	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

Syntax `sh`*lsucond registerY*? [*registerZ*] = *registerT*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory half word at the effective address.

Modifier(s)

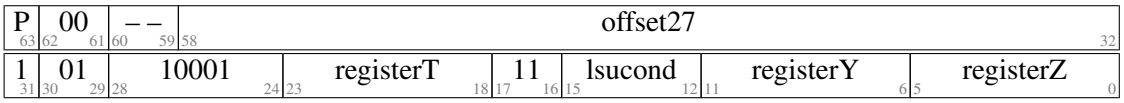
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument1 = GPR[registerY];
  new address = GPR[registerZ];
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0x3, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.O (Section 6.12.153), lsucode0 = 10001



Syntax `sh`*lsucond registerY*? *offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory half word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_27(offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0x3, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.D (Section 6.12.154), lsucode0 = 10001

P	00	– –	extend27												
95	94 93	92 91	90 64												
1	00	– –	offset27												
63	62 61	60 59	58 32												
1	01	10001			registerT			11	lsucond		registerY			registerZ	
31	30 29	28 24	23 18					17 16	15 12		11 6			5 0	

Syntax sh*lsucond registerY*? *extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *extend27_offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory half word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

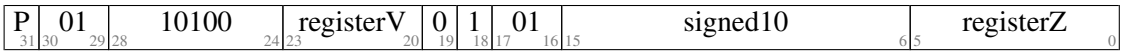
Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_54(extend27_offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0x3, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

6.5.36 SO — Store Octuple Word

Encoding Format LSU_SOBO (Section 6.12.95), lsucode0 = 10100



Syntax `so` *signed10*[*registerZ*] = *registerV*

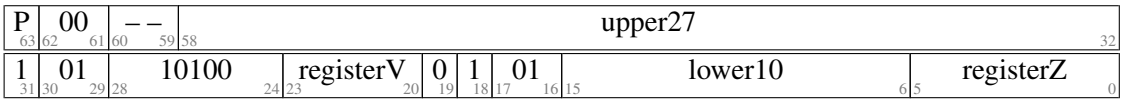
Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerV* is stored into the memory octuple word at the effective address.

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = QGR[registerV];
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
```


Kalray – all rights reserved

Encoding Format LSU_SOBO.X (Section 6.12.96), lsucode0 = 10100



Syntax `so` *upper27_lower10*[*registerZ*] = *registerV*

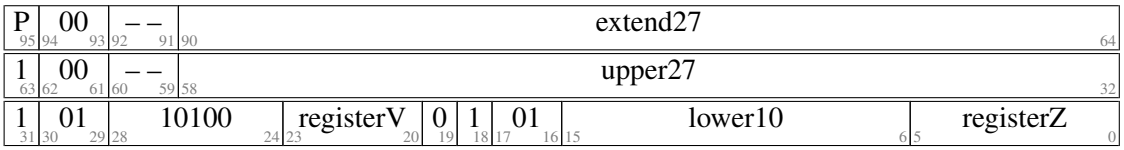
Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerV* is stored into the memory octuple word at the effective address.

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = QGR[registerV];
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
```


Kalray – all rights reserved

Encoding Format LSU_SOBO.Y (Section 6.12.97), lsucode0 = 10100



Syntax `so` *extend27_upper27_lower10*[*registerZ*] = *registerV*

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerV* is stored into the memory octuple word at the effective address.

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = QGR[registerV];
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
```


Kalray – all rights reserved

Encoding Format LSU_SOPB (Section 6.12.158), lsucode0 = 10100

P	01	10100	registerV	0	1	11	lsucond	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

Syntax `so`*lsucond registerY*? [*registerZ*] = *registerV*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerV* is stored into the memory octuple word at the effective address.

Modifier(s)

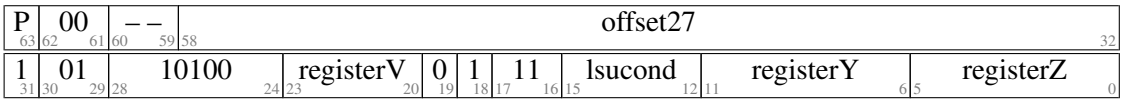
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new argument1 = GPR[registerY];
  new address = GPR[registerZ];
stage E1:
  new argument3 = QGR[registerV];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SOPB.O (Section 6.12.159), lsucode0 = 10100



Syntax `so`*lsucond registerY*? *offset27*[*registerZ*] = *registerV*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerV* is stored into the memory octuple word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_27(offset27);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = QGR[registerV];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SOPB.D (Section 6.12.160), lsucode0 = 10100

| | | |

Encoding Format LSU_SOMB (Section 6.12.161), lsucode0 = 10100

P	01	10100	registerV	1	1	11	lsomask	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

Syntax `so`*lsomask registerY*? [*registerZ*] = *registerV*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsomask*. The *registerV* is stored into the memory octuple word at the effective address.

Modifier(s)

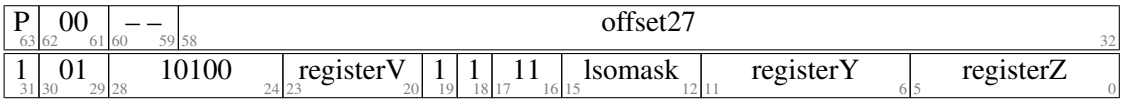
- lsomask (cf. lsomask in section 6.10.2 on page 1421)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new lsumask = _ZX_4(lsomask);
stage RR:
  new address = GPR[registerZ];
  new argument1 = GPR[registerY];
stage E1:
  new argument3 = QGR[registerV];
  new bytemask = lsumask_bytemask(lsumask, argument1);
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
```


Kalray – all rights reserved

Encoding Format LSU_SOMB.O (Section 6.12.162), lsucode0 = 10100



Syntax `so`*lsomask registerY*? *offset27*[*registerZ*] = *registerV*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsomask*. The *registerV* is stored into the memory octuple word at the effective address.

Modifier(s)

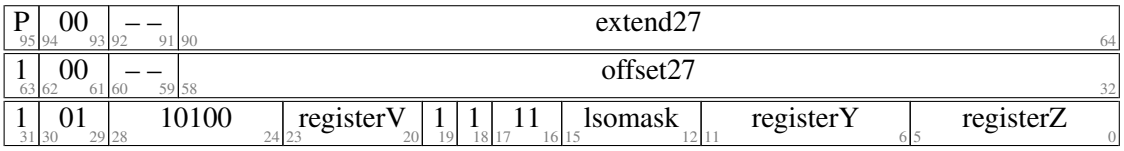
- lsomask (cf. lsomask in section 6.10.2 on page 1421)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_27(offset27);
  new lsumask = _ZX_4(lsomask);
stage RR:
  new base = GPR[registerZ];
  new argument1 = GPR[registerY];
  new address = base + offset;
stage E1:
  new argument3 = QGR[registerV];
  new bytemask = lsumask_bytemask(lsumask, argument1);
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
```


Kalray – all rights reserved

Encoding Format LSU_SOMB.D (Section 6.12.163), lsucode0 = 10100



Syntax so*lsomask registerY*? *extend27_offset27*[*registerZ*] = *registerV*

Description The effective address is given by adding the *extend27_offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsomask*. The *registerV* is stored into the memory octuple word at the effective address.

Modifier(s)

- lsomask (cf. lsomask in section 6.10.2 on page 1421)

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new lsumask = _ZX_4(lsomask);
stage RR:
  new base = GPR[registerZ];
  new argument1 = GPR[registerY];
  new address = base + offset;
stage E1:
  new argument3 = QGR[registerV];
  new bytemask = lsumask_bytemask(lsumask, argument1);
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
```


Kalray – all rights reserved

Encoding Format LSU_SOBI (Section 6.12.193), lsucode0 = 10100

Syntax `so`*doscale registerY*[*registerZ*] = *registerV*

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerV* is stored into the memory octuple word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

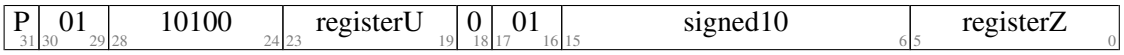
Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new index = GPR[registerY];
  new base = GPR[registerZ];
  new scaling = doscale ? 32 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument3 = QGR[registerV];
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, registerV << 2);
```


Kalray – all rights reserved

6.5.37 SQ — Store Quadruple Word

Encoding Format LSU_SQBO (Section 6.12.92), lsucode0 = 10100



Syntax `sq` *signed10*[*registerZ*] = *registerU*

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerU* is stored into the memory quadruple word at the effective address.

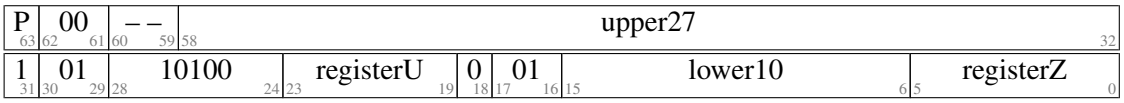
Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = PGR[registerU];
stage E3:
  MEM.store(address, 0xFFFF, 0, argument3, registerU << 1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format LSU_SQBO.X (Section 6.12.93), lsucode0 = 10100



Syntax `sq` *upper27_lower10*[*registerZ*] = *registerU*

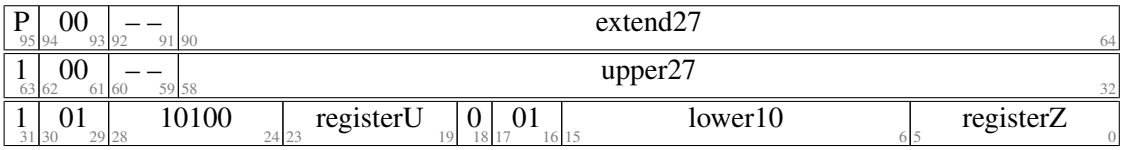
Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerU* is stored into the memory quadruple word at the effective address.

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = PGR[registerU];
stage E3:
  MEM.store(address, 0xFFFF, 0, argument3, registerU << 1);
```


Kalray – all rights reserved

Encoding Format LSU_SQBO.Y (Section 6.12.94), lsucode0 = 10100



Syntax `sq` *extend27_upper27_lower10*[*registerZ*] = *registerU*

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerU* is stored into the memory quadruple word at the effective address.

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = PGR[registerU];
stage E3:
  MEM.store(address, 0xFFFF, 0, argument3, registerU << 1);
```


Kalray – all rights reserved

Encoding Format LSU_SQBI (Section 6.12.192), lsucode0 = 10100

Syntax `sq`*doscale registerY*[*registerZ*] = *registerU*

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerU* is stored into the memory quadruple word at the effective address.

Modifier(s)

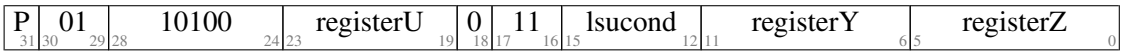
- doscale (cf. doscale in section 6.10.2 on page 1419)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new index = GPR[registerY];
  new base = GPR[registerZ];
  new scaling = doscale ? 16 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument3 = PGR[registerU];
stage E3:
  MEM.store(address, 0xFFFF, 0, argument3, registerU << 1);
```


Kalray – all rights reserved

Encoding Format LSU_SQPB (Section 6.12.155), lsucode0 = 10100



Syntax sq*lsucond registerY*? [*registerZ*] = *registerU*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerU* is stored into the memory quadruple word at the effective address.

Modifier(s)

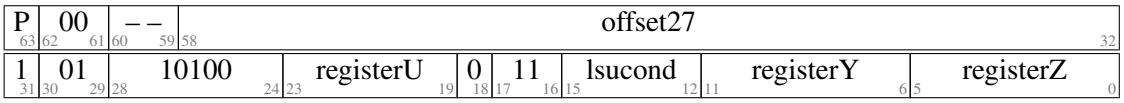
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument1 = GPR[registerY];
  new address = GPR[registerZ];
stage E1:
  new argument3 = PGR[registerU];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xFFFF, 0, argument3, registerU << 1);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SQPB.O (Section 6.12.156), lsucode0 = 10100



Syntax `sq`*lsucond registerY*? *offset27*[*registerZ*] = *registerU*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerU* is stored into the memory quadruple word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_27(offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = PGR[registerU];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xFFFF, 0, argument3, registerU << 1);
  }
```

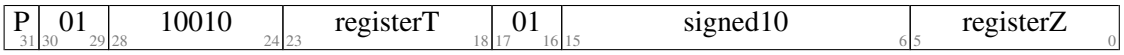

Kalray – all rights reserved

Encoding Format LSU_SQPB.D (Section 6.12.157), lsucode0 = 10100

| | | |

6.5.38 SW Store Word

Encoding Format LSU_SSBO (Section 6.12.89), lsucode0 = 10010



Syntax `sw` *signed10*[*registerZ*] = *registerT*

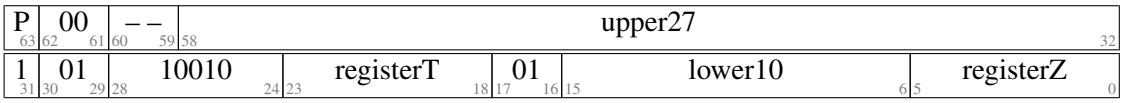
Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerT* is stored into the memory word at the effective address.

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.X (Section 6.12.90), lsucode0 = 10010



Syntax `sw` *upper27_lower10*[*registerZ*] = *registerT*

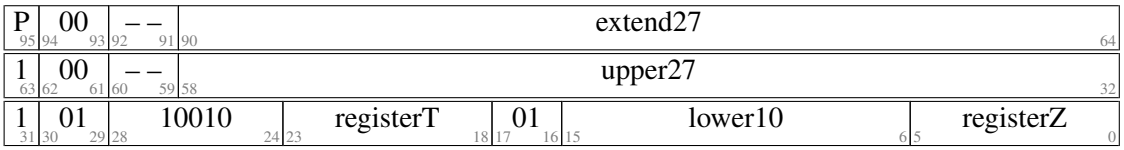
Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerT* is stored into the memory word at the effective address.

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBO.Y (Section 6.12.91), lsucode0 = 10010



Syntax `sw` *extend27_upper27_lower10*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerT* is stored into the memory word at the effective address.

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSBI (Section 6.12.191), lsucode0 = 10010

P	01	10010	registerT	11	111	doscale	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 13	12	11 6	5 0

Syntax `sw`*doscale registerY*[*registerZ*] = *registerT*

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerT* is stored into the memory word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new index = GPR[registerY];
  new base = GPR[registerZ];
  new scaling = doscale ? 4 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  MEM.store(address, 0xF, 0, argument3, registerT);
```


Kalray – all rights reserved

Encoding Format LSU_SSPB (Section 6.12.152), lsucode0 = 10010

P	01	10010	registerT	11	lsucond	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

Syntax `sw`*lsucond registerY*? [*registerZ*] = *registerT*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory word at the effective address.

Modifier(s)

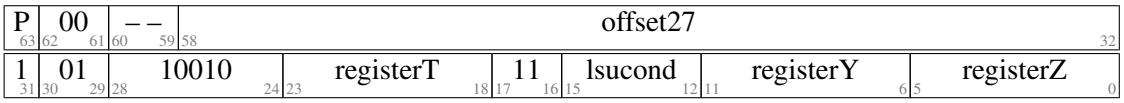
- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR (Section 6.14.30)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
stage RR:
  new argument1 = GPR[registerY];
  new address = GPR[registerZ];
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xF, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.O (Section 6.12.153), lsucode0 = 10010



Syntax `sw`*lsucond registerY*? *offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.X (Section 6.14.31)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_27(offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xF, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

Encoding Format LSU_SSPB.D (Section 6.12.154), lsucode0 = 10010

P 95	00 94 93	– – 92 91	extend27 90 64												
1 63	00 62 61	– – 60 59	offset27 58 32												
1 31	01 30 29	10010 28 24		registerT 23 18		11 17 16	lsucond 15 12		registerY 11 6		registerZ 5 0				

Syntax `sw`*lsucond registerY*? *extend27_offset27*[*registerZ*] = *registerT*

Description The effective address is given by adding the *extend27_offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsucond*. The *registerT* is stored into the memory word at the effective address.

Modifier(s)

- lsucond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table LSU_AUXR.Y (Section 6.14.32)

```
stage ID:
  new argument4 = _ZX_4(lsucond);
  new offset = _SX_54(extend27_offset27);
stage RR:
  new argument1 = GPR[registerY];
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = GPR[registerT];
stage E3:
  if (scalarcond(argument4, argument1)) {
    MEM.store(address, 0xF, 0, argument3, registerT);
  }
```


Kalray – all rights reserved

6.5.39 XLO Extension Load Octuple Word

Encoding Format LSU_XLOBO (Section 6.12.80), steering = 01

P	01	000	variant	registerG	01	signed10	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 6	5 0

Syntax `xlo`*variant registerG* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerG* is loaded from the memory octuple word at the effective address.

Modifier(s)

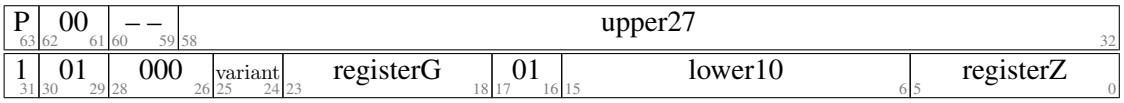
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_10(signed10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerG;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
stage CR:
  XVR[registerG] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XLOBO.X (Section 6.12.81), steering = 01



Syntax `xlo`*variant registerG* = *upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerG* is loaded from the memory octuple word at the effective address.

Modifier(s)

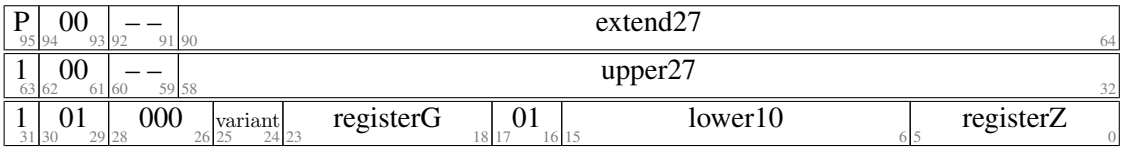
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerG;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
stage CR:
  XVR[registerG] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XLOBO.Y (Section 6.12.82), steering = 01



Syntax `xlo`*variant registerG* = *extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerG* is loaded from the memory octuple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new variant = _ZX_2(variant);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerG;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
stage CR:
  XVR[registerG] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XLOC4BO (Section 6.12.86), steering = 01

Syntax `xlo`*variantqindex registerGq* = *signed10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerGq* is loaded from the memory octuple word at the effective address.

Modifier(s)

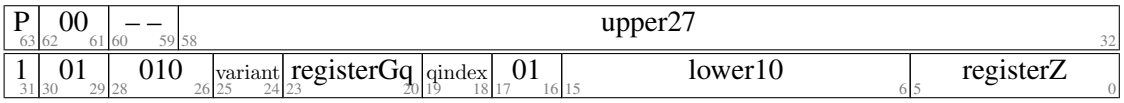
- qindex (cf. qindex in section 6.10.2 on page 1423)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
  new qindex = _ZX_2(qindex);
  new offset = _SX_10(signed10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerGq << 2;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument1_0 = XMR[registerGq]:0;
  new argument1_1 = XMR[registerGq]:1;
  new argument1_2 = XMR[registerGq]:2;
  new argument1_3 = XMR[registerGq]:3;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  new result1_0 = _ZX_64(result1);
  new result1_1 = result1 >> 64;
  new result1_2 = result1 >> 128;
  new result1_3 = result1 >> 192;
stage E3:
  XMR[registerGq]:0 = insert_64(argument1_0, result1_0, qindex);
  XMR[registerGq]:1 = insert_64(argument1_1, result1_1, qindex);
  XMR[registerGq]:2 = insert_64(argument1_2, result1_2, qindex);
  XMR[registerGq]:3 = insert_64(argument1_3, result1_3, qindex);
```


Kalray – all rights reserved

Encoding Format LSU_XLOC4BO.X (Section 6.12.87), steering = 01



Syntax `xlo`*variantqindex registerGq* = *upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerGq* is loaded from the memory octuple word at the effective address.

Modifier(s)

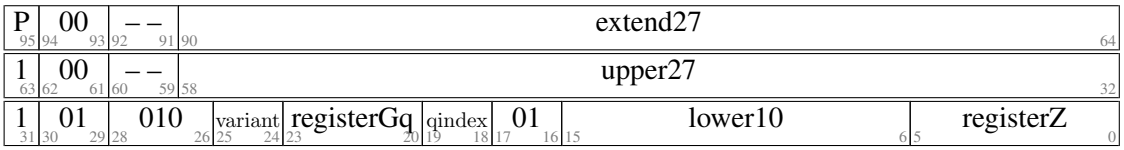
- qindex (cf. qindex in section 6.10.2 on page 1423)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new variant = _ZX_2(variant);
  new qindex = _ZX_2(qindex);
  new offset = _SX_37(upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerGq << 2;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument1_0 = XMR[registerGq]:0;
  new argument1_1 = XMR[registerGq]:1;
  new argument1_2 = XMR[registerGq]:2;
  new argument1_3 = XMR[registerGq]:3;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  new result1_0 = _ZX_64(result1);
  new result1_1 = result1 >> 64;
  new result1_2 = result1 >> 128;
  new result1_3 = result1 >> 192;
stage E3:
  XMR[registerGq]:0 = insert_64(argument1_0, result1_0, qindex);
  XMR[registerGq]:1 = insert_64(argument1_1, result1_1, qindex);
  XMR[registerGq]:2 = insert_64(argument1_2, result1_2, qindex);
  XMR[registerGq]:3 = insert_64(argument1_3, result1_3, qindex);
```


Kalray – all rights reserved

Encoding Format LSU_XLOC4BO.Y (Section 6.12.88), steering = 01



Syntax `xlo`*variantqindex registerGq* = *extend27_upper27_lower10*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerGq* is loaded from the memory octuple word at the effective address.

Modifier(s)

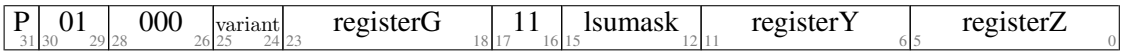
- qindex (cf. qindex in section 6.10.2 on page 1423)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new variant = _ZX_2(variant);
  new qindex = _ZX_2(qindex);
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerGq << 2;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument1_0 = XMR[registerGq]:0;
  new argument1_1 = XMR[registerGq]:1;
  new argument1_2 = XMR[registerGq]:2;
  new argument1_3 = XMR[registerGq]:3;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  new result1_0 = _ZX_64(result1);
  new result1_1 = result1 >> 64;
  new result1_2 = result1 >> 128;
  new result1_3 = result1 >> 192;
stage E3:
  XMR[registerGq]:0 = insert_64(argument1_0, result1_0, qindex);
  XMR[registerGq]:1 = insert_64(argument1_1, result1_1, qindex);
  XMR[registerGq]:2 = insert_64(argument1_2, result1_2, qindex);
  XMR[registerGq]:3 = insert_64(argument1_3, result1_3, qindex);
```


Kalray – all rights reserved

Encoding Format LSU_XLOMB (Section 6.12.125), steering = 01



Syntax `xlo`*variantlsumask registerY*? *registerG* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *lsumask*. The *registerG* is loaded from the memory octuple word at the effective address.

Modifier(s)

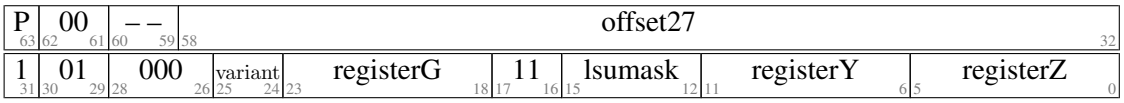
- lsumask (cf. lsumask in section 6.10.2 on page 1422)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsumask);
stage RR:
  new dri = registerG;
  new address = GPR[registerZ];
  new argument2 = GPR[registerY];
stage E1:
  new argument1 = XVR[registerG];
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
stage CR:
  XVR[registerG] = (result1 & passbytes) | (argument1 & clearbytes);
```


Kalray – all rights reserved

Encoding Format LSU_XLOMB.O (Section 6.12.126), steering = 01



Syntax `xlo`*variantlsumask registerY*? *registerG* = *offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsumask*. The *registerG* is loaded from the memory octuple word at the effective address.

Modifier(s)

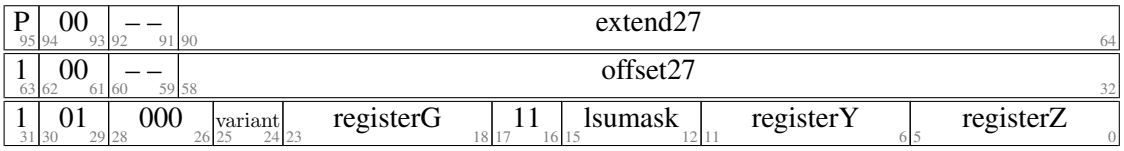
- lsumask (cf. lsumask in section 6.10.2 on page 1422)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsumask);
stage RR:
  new dri = registerG;
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
stage E1:
  new argument1 = XVR[registerG];
  new address = base + offset;
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
stage CR:
  XVR[registerG] = (result1 & passbytes) | (argument1 & clearbytes);
```


Kalray – all rights reserved

Encoding Format LSU_XLOMB.D (Section 6.12.127), steering = 01



Syntax `xlo`*variantlsumask registerY*? *registerG* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *variant* to the *registerZ*. The predicate is computed on the *registerY* with the *lsumask*. The *registerG* is loaded from the memory octuple word at the effective address.

Modifier(s)

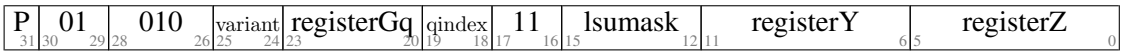
- lsumask (cf. lsumask in section 6.10.2 on page 1422)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsumask);
stage RR:
  new dri = registerG;
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
stage E1:
  new argument1 = XVR[registerG];
  new address = base + offset;
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
stage CR:
  XVR[registerG] = (result1 & passbytes) | (argument1 & clearbytes);
```


Kalray – all rights reserved

Encoding Format LSU_XLOC4MB (Section 6.12.131), steering = 01



Syntax `xlo`*variantlsumaskqindex registerY*? *registerGq* = [*registerZ*]

Description The effective address is given by the *registerZ*. The predicate is computed on the *registerY* with the *qindex*. The *registerGq* is loaded from the memory octuple word at the effective address.

Modifier(s)

- qindex (cf. qindex in section 6.10.2 on page 1423)
- lsumask (cf. lsumask in section 6.10.2 on page 1422)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

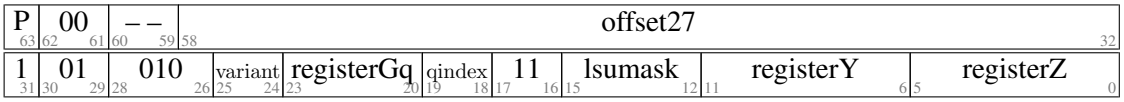
```
stage ID:
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsumask);
  new qindex = _ZX_2(qindex);
stage RR:
  new dri = registerGq << 2;
  new address = GPR[registerZ];
  new argument2 = GPR[registerY];
stage E1:
  new argument1_0 = XMR[registerGq]:0;
  new argument1_0_0 = argument1_0.64[0];
  new argument1_0_1 = argument1_0.64[1];
  new argument1_0_2 = argument1_0.64[2];
  new argument1_0_3 = argument1_0.64[3];
  new argument1_1 = XMR[registerGq]:1;
  new argument1_1_0 = argument1_1.64[0];
  new argument1_1_1 = argument1_1.64[1];
  new argument1_1_2 = argument1_1.64[2];
  new argument1_1_3 = argument1_1.64[3];
  new argument1_2 = XMR[registerGq]:2;
  new argument1_2_0 = argument1_2.64[0];
  new argument1_2_1 = argument1_2.64[1];
  new argument1_2_2 = argument1_2.64[2];
  new argument1_2_3 = argument1_2.64[3];
  new argument1_3 = XMR[registerGq]:3;
  new argument1_3_0 = argument1_3.64[0];
  new argument1_3_1 = argument1_3.64[1];
  new argument1_3_2 = argument1_3.64[2];
```


Kalray – all rights reserved

```
  new argument1_3_3 = argument1_3.64[3];
  new argument1 = 0;
  if (qindex == 0) {
    argument1 = join_64_x4(argument1_0_0, argument1_1_0, argument1_2_0,
        argument1_3_0);
  }
  if (qindex == 1) {
    argument1 = join_64_x4(argument1_0_1, argument1_1_1, argument1_2_1,
        argument1_3_1);
  }
  if (qindex == 2) {
    argument1 = join_64_x4(argument1_0_2, argument1_1_2, argument1_2_2,
        argument1_3_2);
  }
  if (qindex == 3) {
    argument1 = join_64_x4(argument1_0_3, argument1_1_3, argument1_2_3,
        argument1_3_3);
  }
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = (result1 & passbytes) | (argument1 & clearbytes);
  new result1_0 = _ZX_64(result1);
  new result1_1 = result1 >> 64;
  new result1_2 = result1 >> 128;
  new result1_3 = result1 >> 192;
stage E3:
  XMR[registerGq]:0 = insert_64(argument1_0, result1_0, qindex);
  XMR[registerGq]:1 = insert_64(argument1_1, result1_1, qindex);
  XMR[registerGq]:2 = insert_64(argument1_2, result1_2, qindex);
  XMR[registerGq]:3 = insert_64(argument1_3, result1_3, qindex);
```


Kalray – all rights reserved

Encoding Format LSU_XLOC4MB.O (Section 6.12.132), steering = 01



Syntax `xlo`*variantlsumaskqindex registerY*? *registerGq* = *offset27*[*registerZ*]

Description The effective address is given by adding the *lsumask* to the *registerZ*. The predicate is computed on the *registerY* with the *qindex*. The *registerGq* is loaded from the memory octuple word at the effective address.

Modifier(s)

- qindex (cf. qindex in section 6.10.2 on page 1423)
- lsumask (cf. lsumask in section 6.10.2 on page 1422)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

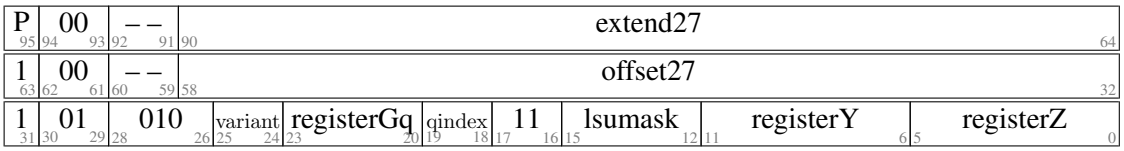
```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsumask);
  new qindex = _ZX_2(qindex);
stage RR:
  new dri = registerGq << 2;
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage E1:
  new argument1_0 = XMR[registerGq]:0;
  new argument1_0_0 = argument1_0.64[0];
  new argument1_0_1 = argument1_0.64[1];
  new argument1_0_2 = argument1_0.64[2];
  new argument1_0_3 = argument1_0.64[3];
  new argument1_1 = XMR[registerGq]:1;
  new argument1_1_0 = argument1_1.64[0];
  new argument1_1_1 = argument1_1.64[1];
  new argument1_1_2 = argument1_1.64[2];
  new argument1_1_3 = argument1_1.64[3];
  new argument1_2 = XMR[registerGq]:2;
  new argument1_2_0 = argument1_2.64[0];
  new argument1_2_1 = argument1_2.64[1];
  new argument1_2_2 = argument1_2.64[2];
  new argument1_2_3 = argument1_2.64[3];
```


Kalray – all rights reserved

```
  new argument1_3 = XMR[registerGq]:3;
  new argument1_3_0 = argument1_3.64[0];
  new argument1_3_1 = argument1_3.64[1];
  new argument1_3_2 = argument1_3.64[2];
  new argument1_3_3 = argument1_3.64[3];
  new argument1 = 0;
  if (qindex == 0) {
    argument1 = join_64_x4(argument1_0_0, argument1_1_0, argument1_2_0,
        argument1_3_0);
  }
  if (qindex == 1) {
    argument1 = join_64_x4(argument1_0_1, argument1_1_1, argument1_2_1,
        argument1_3_1);
  }
  if (qindex == 2) {
    argument1 = join_64_x4(argument1_0_2, argument1_1_2, argument1_2_2,
        argument1_3_2);
  }
  if (qindex == 3) {
    argument1 = join_64_x4(argument1_0_3, argument1_1_3, argument1_2_3,
        argument1_3_3);
  }
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = (result1 & passbytes) | (argument1 & clearbytes);
  new result1_0 = _ZX_64(result1);
  new result1_1 = result1 >> 64;
  new result1_2 = result1 >> 128;
  new result1_3 = result1 >> 192;
stage E3:
  XMR[registerGq]:0 = insert_64(argument1_0, result1_0, qindex);
  XMR[registerGq]:1 = insert_64(argument1_1, result1_1, qindex);
  XMR[registerGq]:2 = insert_64(argument1_2, result1_2, qindex);
  XMR[registerGq]:3 = insert_64(argument1_3, result1_3, qindex);
```


Kalray – all rights reserved

Encoding Format LSU_XLOC4MB.D (Section 6.12.133), steering = 01



Syntax `xlo`*variantlsumaskqindex registerY*? *registerGq* = *extend27_offset27*[*registerZ*]

Description The effective address is given by adding the *lsumask* to the *registerZ*. The predicate is computed on the *registerY* with the *qindex*. The *registerGq* is loaded from the memory octuple word at the effective address.

Modifier(s)

- qindex (cf. qindex in section 6.10.2 on page 1423)
- lsumask (cf. lsumask in section 6.10.2 on page 1422)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
  new lsumask = _ZX_4(lsumask);
  new qindex = _ZX_2(qindex);
stage RR:
  new dri = registerGq << 2;
  new base = GPR[registerZ];
  new argument2 = GPR[registerY];
  new address = base + offset;
stage E1:
  new argument1_0 = XMR[registerGq]:0;
  new argument1_0_0 = argument1_0.64[0];
  new argument1_0_1 = argument1_0.64[1];
  new argument1_0_2 = argument1_0.64[2];
  new argument1_0_3 = argument1_0.64[3];
  new argument1_1 = XMR[registerGq]:1;
  new argument1_1_0 = argument1_1.64[0];
  new argument1_1_1 = argument1_1.64[1];
  new argument1_1_2 = argument1_1.64[2];
  new argument1_1_3 = argument1_1.64[3];
  new argument1_2 = XMR[registerGq]:2;
  new argument1_2_0 = argument1_2.64[0];
  new argument1_2_1 = argument1_2.64[1];
  new argument1_2_2 = argument1_2.64[2];
```


Kalray – all rights reserved

```
  new argument1_2_3 = argument1_2.64[3];
  new argument1_3 = XMR[registerGq]:3;
  new argument1_3_0 = argument1_3.64[0];
  new argument1_3_1 = argument1_3.64[1];
  new argument1_3_2 = argument1_3.64[2];
  new argument1_3_3 = argument1_3.64[3];
  new argument1 = 0;
  if (qindex == 0) {
    argument1 = join_64_x4(argument1_0_0, argument1_1_0, argument1_2_0,
        argument1_3_0);
  }
  if (qindex == 1) {
    argument1 = join_64_x4(argument1_0_1, argument1_1_1, argument1_2_1,
        argument1_3_1);
  }
  if (qindex == 2) {
    argument1 = join_64_x4(argument1_0_2, argument1_1_2, argument1_2_2,
        argument1_3_2);
  }
  if (qindex == 3) {
    argument1 = join_64_x4(argument1_0_3, argument1_1_3, argument1_2_3,
        argument1_3_3);
  }
  new bytemask = lsumask_bytemask(lsumask, argument2);
  new passbytes = lsumask_passbytes(lsumask, argument2);
  new clearbytes = lsumask_clearbytes(lsumask, argument2);
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = (result1 & passbytes) | (argument1 & clearbytes);
  new result1_0 = _ZX_64(result1);
  new result1_1 = result1 >> 64;
  new result1_2 = result1 >> 128;
  new result1_3 = result1 >> 192;
stage E3:
  XMR[registerGq]:0 = insert_64(argument1_0, result1_0, qindex);
  XMR[registerGq]:1 = insert_64(argument1_1, result1_1, qindex);
  XMR[registerGq]:2 = insert_64(argument1_2, result1_2, qindex);
  XMR[registerGq]:3 = insert_64(argument1_3, result1_3, qindex);
```


Kalray – all rights reserved

Encoding Format LSU_XLBI (Section 6.12.188), steering = 01

Syntax `xlo`*variantdoscale registerG* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerG* is loaded from the memory octuple word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerG;
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 32 : 1;
  new address = base + (index * scaling);
stage E1:
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
stage CR:
  XVR[registerG] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XLOC4BI (Section 6.12.190), steering = 01

Syntax `xlo`*variantdoscaleqindex registerGq* = *registerY*[*registerZ*]

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *qindex*. The *registerGq* is loaded from the memory octuple word at the effective address.

Modifier(s)

- qindex (cf. qindex in section 6.10.2 on page 1423)
- doscale (cf. doscale in section 6.10.2 on page 1419)
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
  new doscale = _ZX_1(doscale);
  new qindex = _ZX_2(qindex);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerGq << 2;
  new base = GPR[registerZ];
  new index = GPR[registerY];
  new scaling = doscale ? 32 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument1_0 = XMR[registerGq]:0;
  new argument1_1 = XMR[registerGq]:1;
  new argument1_2 = XMR[registerGq]:2;
  new argument1_3 = XMR[registerGq]:3;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  new result1_0 = _ZX_64(result1);
  new result1_1 = result1 >> 64;
  new result1_2 = result1 >> 128;
  new result1_3 = result1 >> 192;
stage E3:
  XMR[registerGq]:0 = insert_64(argument1_0, result1_0, qindex);
  XMR[registerGq]:1 = insert_64(argument1_1, result1_1, qindex);
  XMR[registerGq]:2 = insert_64(argument1_2, result1_2, qindex);
  XMR[registerGq]:3 = insert_64(argument1_3, result1_3, qindex);
```


Kalray – all rights reserved

6.5.40 XPLB Extension Preload Byte

Encoding Format LSU_XPL2RBB (Section 6.12.134), lsucode6 = 0101

P	01	011	variant	registerGg	0	11	0101	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xplb`*variant registerGg*, *registerY* = [*registerZ*]

Description The *registerGg* is preloaded from the memory byte at the effective address.

Modifier(s)

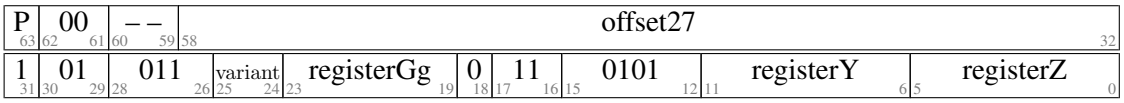
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.O (Section 6.12.135), lsucode6 = 0101



Syntax `xplb`*variant registerGg*, *registerY* = *offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory byte at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.D (Section 6.12.136), lsucode6 = 0101

P	00	– –	extend27
95	94 93	92 91	90 64
1	00	– –	offset27
63	62 61	60 59	58 32

1	01	011	variant	registerGg	0	11	0101	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xplb`*variant registerGg*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory byte at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB (Section 6.12.137), lsucode6 = 0101

P	01	011	variant	registerGh	01	11	0101	registerY	registerZ
31	30 29	28 26	25 24	23 20	19 18	17 16	15 12	11 6	5 0

Syntax `xplb`*variant registerGh*, *registerY* = [*registerZ*]

Description The *registerGh* is preloaded from the memory byte at the effective address.

Modifier(s)

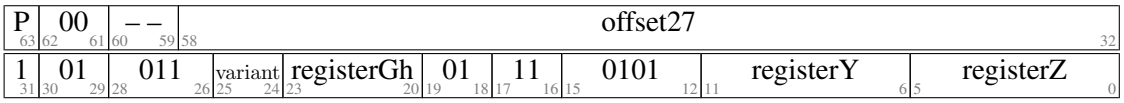
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.O (Section 6.12.138), lsucode6 = 0101



Syntax `xplb`*variant registerGh*, *registerY* = *offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory byte at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.D (Section 6.12.139), lsucode6 = 0101

P 95	00 94 93	– – 92 91	extend27 90 64							
1 63	00 62 61	– – 60 59	offset27 58 32							
1 31	01 30 29	011 28 26	variant 25 24	registerGh 23 20	01 19 18	11 17 16	0101 15 12	registerY 11 6	registerZ 5 0	

Syntax `xplb`*variant registerGh*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory byte at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB (Section 6.12.140), lsucode6 = 0101

P	01	011	variant	registerGi	011	11	0101	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xplb`*variant registerGi*, *registerY* = [*registerZ*]

Description The *registerGi* is preloaded from the memory byte at the effective address.

Modifier(s)

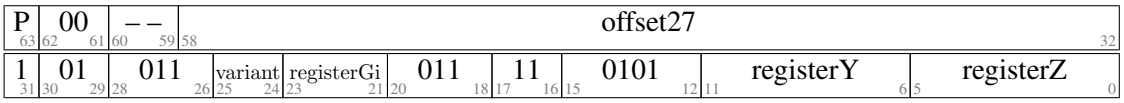
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.O (Section 6.12.141), lsucode6 = 0101



Syntax `xplb`*variant registerGi*, *registerY* = *offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory byte at the effective address.

Modifier(s)

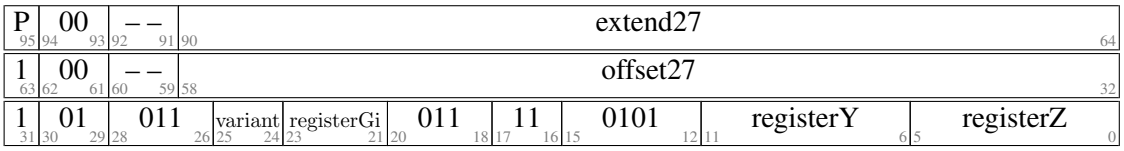
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.D (Section 6.12.142), lsucode6 = 0101



Syntax `xplb`*variant registerGi*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory byte at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB (Section 6.12.143), lsucode6 = 0101

Syntax `xplb`*variant registerGj*, *registerY* = [*registerZ*]

Description The *registerGj* is preloaded from the memory byte at the effective address.

Modifier(s)

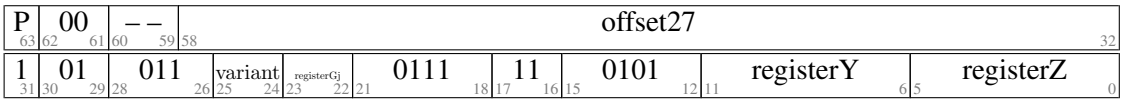
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.O (Section 6.12.144), lsucode6 = 0101



Syntax `xplb`*variant registerGj*, *registerY* = *offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory byte at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.D (Section 6.12.145), lsucode6 = 0101

P 95	00 94 93	– – 92 91	extend27 90 64							
1 63	00 62 61	– – 60 59	offset27 58 32							
1 31	01 30 29	011 28 26	variant 25 24	registerGj 23 22	0111 21 18	11 17 16	0101 15 12	registerY 11 6	registerZ 5 0	

Syntax `xplb`*variant registerGj*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory byte at the effective address.

Modifier(s)

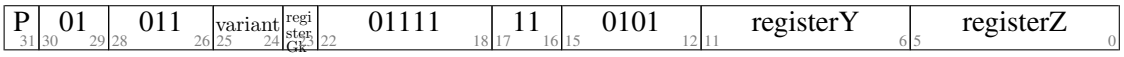
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB (Section 6.12.146), lsucode6 = 0101



Syntax `xplb`*variant registerGk*, *registerY* = [*registerZ*]

Description The *registerGk* is preloaded from the memory byte at the effective address.

Modifier(s)

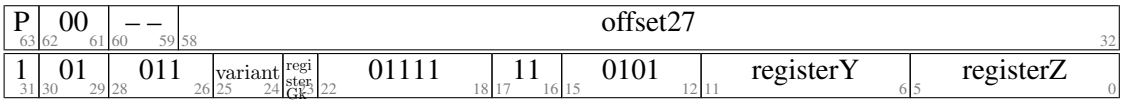
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.O (Section 6.12.147), lsucode6 = 0101



Syntax `xplb`*variant registerGk*, *registerY* = *offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory byte at the effective address.

Modifier(s)

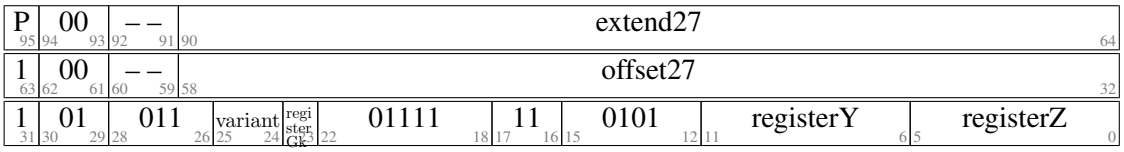
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.D (Section 6.12.148), lsucode6 = 0101



Syntax `xplb`*variant registerGk*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory byte at the effective address.

Modifier(s)

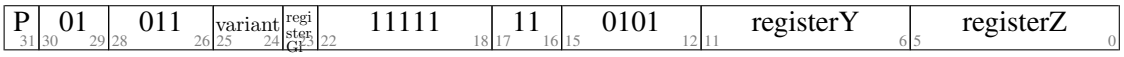
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB (Section 6.12.149), lsucode6 = 0101



Syntax `xplb`*variant registerGl*, *registerY* = [*registerZ*]

Description The *registerGl* is preloaded from the memory byte at the effective address.

Modifier(s)

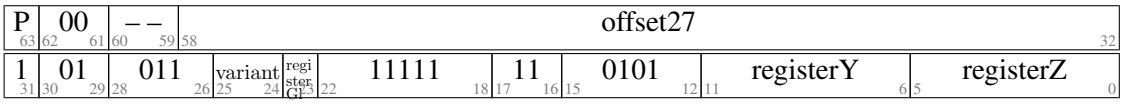
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.O (Section 6.12.150), lsucode6 = 0101



Syntax `xplb`*variant registerGl*, *registerY* = *offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory byte at the effective address.

Modifier(s)

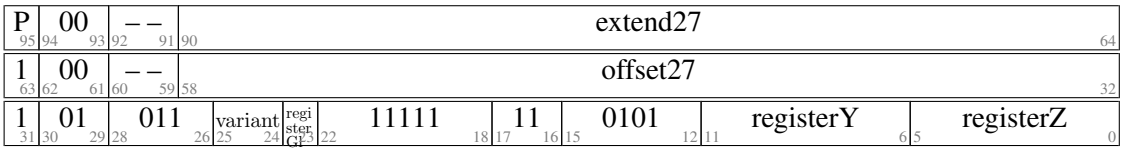
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.D (Section 6.12.151), lsucode6 = 0101



Syntax `xplb`*variant registerGl*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory byte at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 1;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x1 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.5.41 XPLD — Extension Preload Double Word

Encoding Format LSU_XPL2RBB (Section 6.12.134), lsucode6 = 0010

P	01	011	variant	registerGg	0	11	0010	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xpld`*variant registerGg*, *registerY* = [*registerZ*]

Description The *registerGg* is preloaded from the memory double word at the effective address.

Modifier(s)

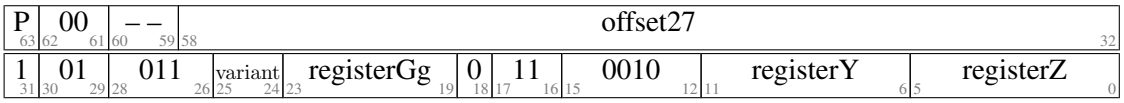
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.O (Section 6.12.135), lsucode6 = 0010



Syntax `xpld`*variant registerGg*, *registerY* = *offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory double word at the effective address.

Modifier(s)

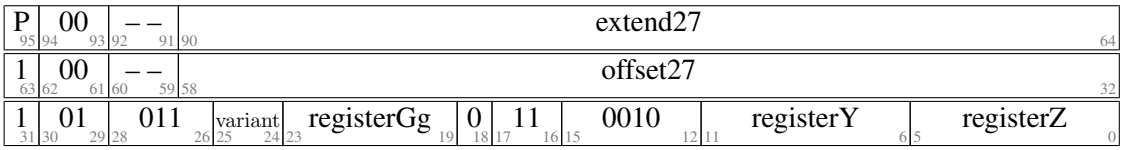
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.D (Section 6.12.136), lsucode6 = 0010



Syntax `xpld`*variant registerGg*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB (Section 6.12.137), lsucode6 = 0010

P	01	011	variant	registerGh	01	11	0010	registerY	registerZ
31	30 29	28 26	25 24	23 20	19 18	17 16	15 12	11 6	5 0

Syntax `xpld`*variant registerGh*, *registerY* = [*registerZ*]

Description The *registerGh* is preloaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.O (Section 6.12.138), lsucode6 = 0010

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	011	variant	registerGh	01	11	0010	registerY	registerZ
31	30 29	28 26	25 24	23 20	19 18	17 16	15 12	11 6	5 0

Syntax `xpld`*variant registerGh*, *registerY* = *offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.D (Section 6.12.139), lsucode6 = 0010

P 95	00 94 93	– – 92 91	extend27 90 64									
1 63	00 62 61	– – 60 59	offset27 58 32									
1 31	01 30 29	011 28 26	variant 25 24	registerGh 23 20	01 19 18	11 17 16	0010 15 12	registerY 11 6	registerZ 5 0			

Syntax `xpld`*variant registerGh*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB (Section 6.12.140), lsucode6 = 0010

P	01	011	variant	registerGi	011	11	0010	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xpld`*variant registerGi*, *registerY* = [*registerZ*]

Description The *registerGi* is preloaded from the memory double word at the effective address.

Modifier(s)

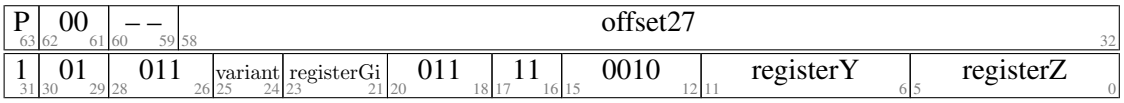
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.O (Section 6.12.141), lsucode6 = 0010



Syntax `xpld`*variant registerGi*, *registerY* = *offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.D (Section 6.12.142), lsucode6 = 0010

P 95	00 94 93	– – 92 91	extend27 90 64							
1 63	00 62 61	– – 60 59	offset27 58 32							
1 31	01 30 29	011 28 26	variant 25 24	registerGi 23 21	011 20 18	11 17 16	0010 15 12	registerY 11 6	registerZ 5 0	

Syntax `xpld`*variant registerGi*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB (Section 6.12.143), lsucode6 = 0010

Syntax `xpld`*variant registerGj*, *registerY* = [*registerZ*]

Description The *registerGj* is preloaded from the memory double word at the effective address.

Modifier(s)

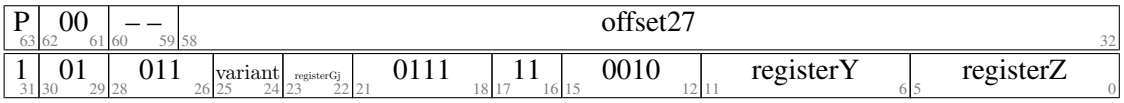
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.O (Section 6.12.144), lsucode6 = 0010



Syntax `xpld`*variant registerGj*, *registerY* = *offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory double word at the effective address.

Modifier(s)

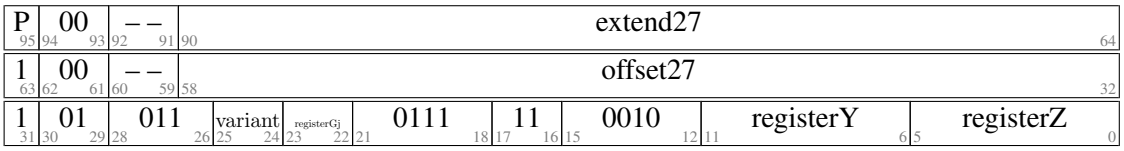
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.D (Section 6.12.145), lsucode6 = 0010



Syntax `xpld`*variant registerGj*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory double word at the effective address.

Modifier(s)

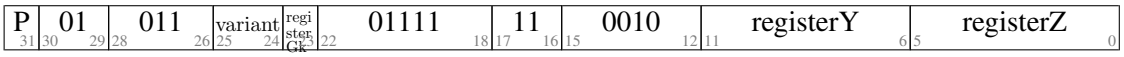
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB (Section 6.12.146), lsucode6 = 0010



Syntax `xpld`*variant registerGk*, *registerY* = [*registerZ*]

Description The *registerGk* is preloaded from the memory double word at the effective address.

Modifier(s)

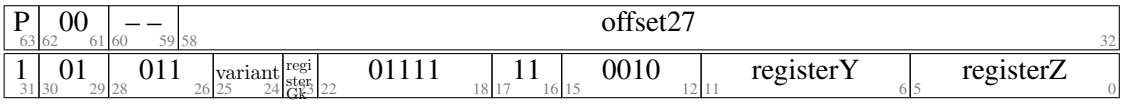
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.O (Section 6.12.147), lsucode6 = 0010



Syntax `xpld`*variant registerGk*, *registerY* = *offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory double word at the effective address.

Modifier(s)

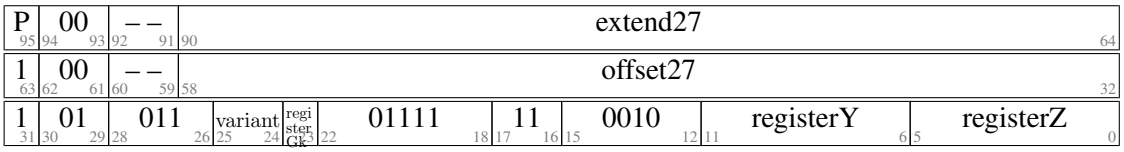
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.D (Section 6.12.148), lsucode6 = 0010



Syntax `xpld`*variant registerGk*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory double word at the effective address.

Modifier(s)

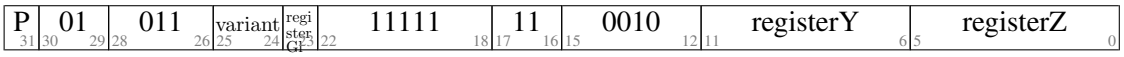
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB (Section 6.12.149), lsucode6 = 0010



Syntax `xpld`*variant registerGl*, *registerY* = [*registerZ*]

Description The *registerGl* is preloaded from the memory double word at the effective address.

Modifier(s)

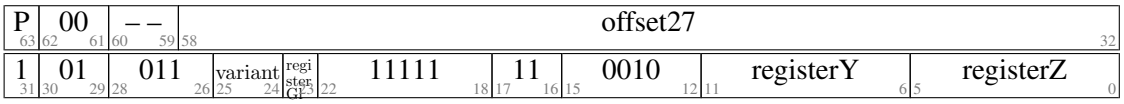
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.O (Section 6.12.150), lsucode6 = 0010



Syntax `xpld`*variant registerGl*, *registerY* = *offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory double word at the effective address.

Modifier(s)

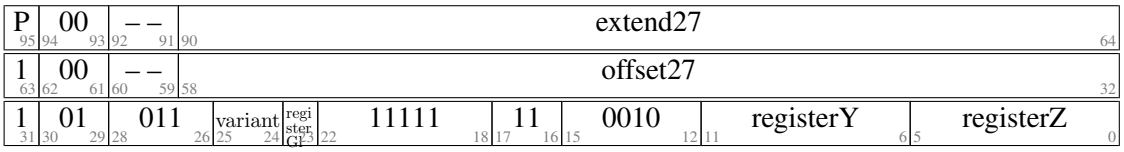
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.D (Section 6.12.151), lsucode6 = 0010



Syntax `xpld`*variant registerGl*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory double word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 255;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

6.5.42 XPLH Extension Preload Half

Encoding Format LSU_XPL2RBB (Section 6.12.134), lsucode6 = 0100

P	01	011	variant	registerGg	0	11	0100	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xplh`*variant registerGg*, *registerY* = [*registerZ*]

Description The *registerGg* is preloaded from the memory half word at the effective address.

Modifier(s)

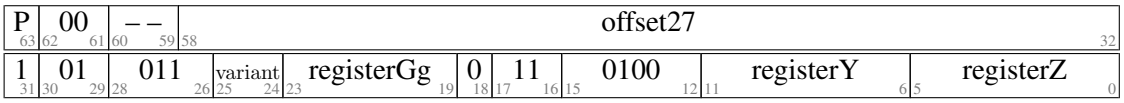
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.O (Section 6.12.135), lsucode6 = 0100



Syntax `xplh`*variant registerGg*, *registerY* = *offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory half word at the effective address.

Modifier(s)

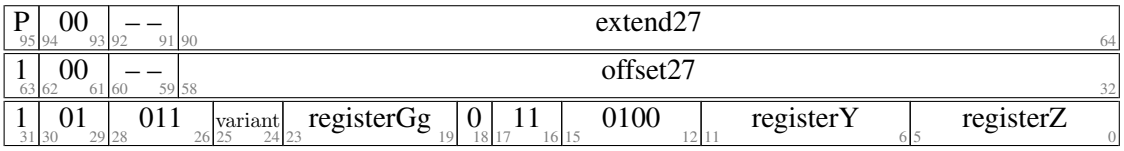
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.D (Section 6.12.136), lsucode6 = 0100



Syntax `xplh`*variant registerGg*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory half word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB (Section 6.12.137), lsucode6 = 0100

Syntax `xplh`*variant registerGh*, *registerY* = [*registerZ*]

Description The *registerGh* is preloaded from the memory half word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.O (Section 6.12.138), lsucode6 = 0100

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	011	variant	registerGh	01	11	0100	registerY	registerZ
31	30 29	28 26	25 24	23 20	19 18	17 16	15 12	11 6	5 0

Syntax `xplh`*variant registerGh*, *registerY* = *offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory half word at the effective address.

Modifier(s)

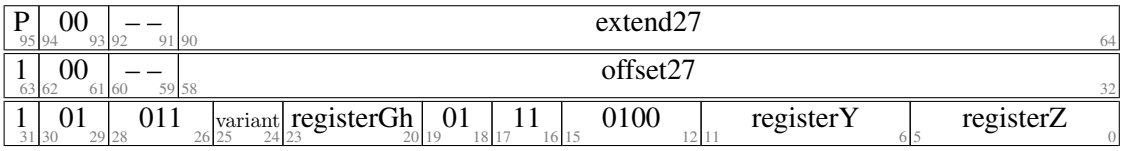
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.D (Section 6.12.139), lsucode6 = 0100



Syntax `xplh`*variant registerGh*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory half word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB (Section 6.12.140), lsucode6 = 0100

P	01	011	variant	registerGi	011	11	0100	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xplh`*variant registerGi*, *registerY* = [*registerZ*]

Description The *registerGi* is preloaded from the memory half word at the effective address.

Modifier(s)

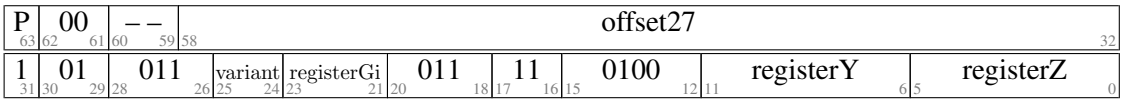
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.O (Section 6.12.141), lsucode6 = 0100



Syntax `xplh`*variant registerGi*, *registerY* = *offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory half word at the effective address.

Modifier(s)

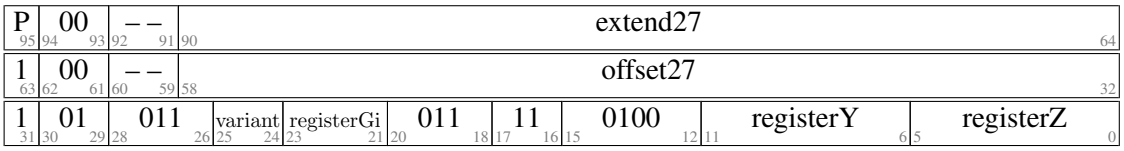
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.D (Section 6.12.142), lsucode6 = 0100



Syntax `xplh`*variant registerGi*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory half word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB (Section 6.12.143), lsucode6 = 0100

Syntax `xplh`*variant registerGj*, *registerY* = [*registerZ*]

Description The *registerGj* is preloaded from the memory half word at the effective address.

Modifier(s)

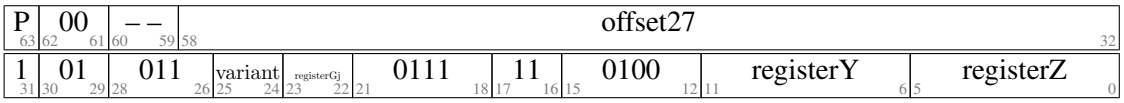
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.O (Section 6.12.144), lsucode6 = 0100



Syntax `xplh`*variant registerGj*, *registerY* = *offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory half word at the effective address.

Modifier(s)

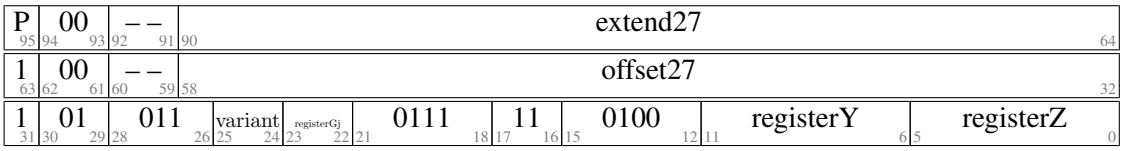
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.D (Section 6.12.145), lsucode6 = 0100



Syntax `xplh`*variant registerGj*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory half word at the effective address.

Modifier(s)

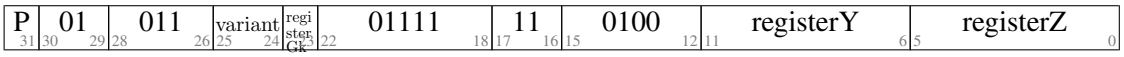
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB (Section 6.12.146), lsucode6 = 0100



Syntax `xplh`*variant registerGk*, *registerY* = [*registerZ*]

Description The *registerGk* is preloaded from the memory half word at the effective address.

Modifier(s)

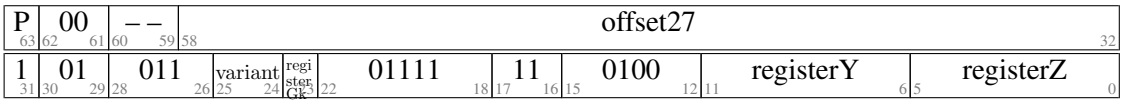
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.O (Section 6.12.147), lsucode6 = 0100



Syntax `xplh`*variant registerGk*, *registerY* = *offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory half word at the effective address.

Modifier(s)

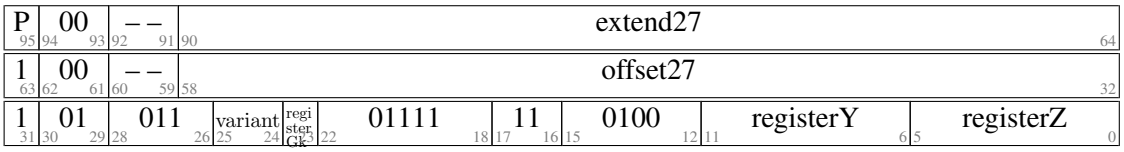
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.D (Section 6.12.148), lsucode6 = 0100



Syntax `xplh`*variant registerGk*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory half word at the effective address.

Modifier(s)

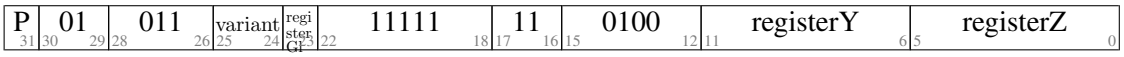
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB (Section 6.12.149), lsucode6 = 0100



Syntax `xplh`*variant registerGl*, *registerY* = [*registerZ*]

Description The *registerGl* is preloaded from the memory half word at the effective address.

Modifier(s)

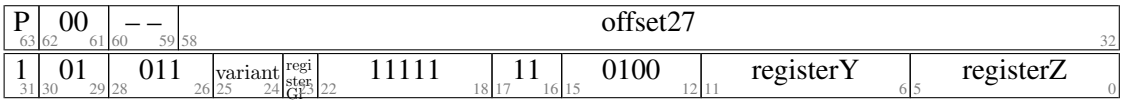
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.O (Section 6.12.150), lsucode6 = 0100



Syntax `xplh`*variant registerGl*, *registerY* = *offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory half word at the effective address.

Modifier(s)

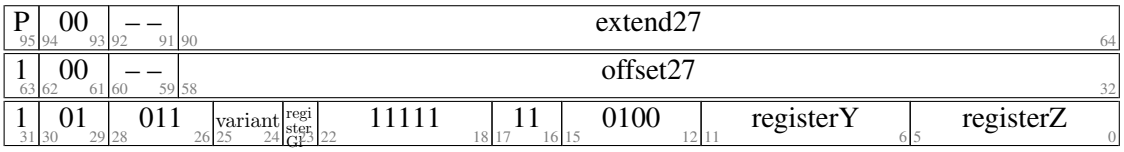
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.D (Section 6.12.151), lsucode6 = 0100



Syntax `xplh`*variant registerGl*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory half word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 3;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0x3 & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

6.5.43 XPLO Extension Preload Octuple Word

Encoding Format LSU_XPL2RBB (Section 6.12.134), lsucode6 = 0000

P	01	011	variant	registerGg	0	11	0000	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xplo`*variant registerGg*, *registerY* = [*registerZ*]

Description The *registerGg* is preloaded from the memory octuple word at the effective address.

Modifier(s)

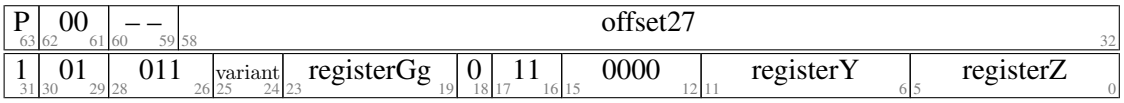
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.O (Section 6.12.135), lsucode6 = 0000



Syntax `xplo`*variant registerGg*, *registerY* = *offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory octuple word at the effective address.

Modifier(s)

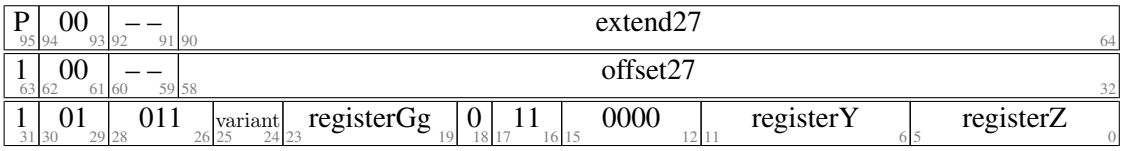
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.D (Section 6.12.136), lsucode6 = 0000



Syntax `xplo`*variant registerGg*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory octuple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB (Section 6.12.137), lsucode6 = 0000

Syntax `xplo`*variant registerGh*, *registerY* = [*registerZ*]

Description The *registerGh* is preloaded from the memory octuple word at the effective address.

Modifier(s)

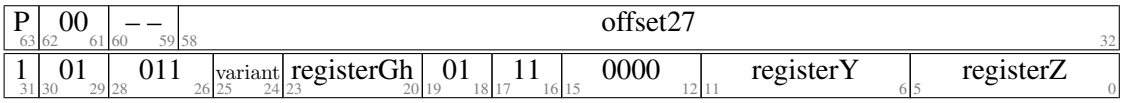
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.O (Section 6.12.138), lsucode6 = 0000



Syntax `xplo`*variant registerGh*, *registerY* = *offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory octuple word at the effective address.

Modifier(s)

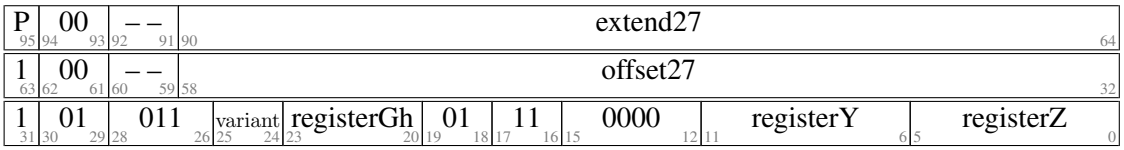
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.D (Section 6.12.139), lsucode6 = 0000



Syntax `xplo`*variant registerGh*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory octuple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB (Section 6.12.140), lsucode6 = 0000

P	01	011	variant	registerGi	011	11	0000	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xplo`*variant registerGi*, *registerY* = [*registerZ*]

Description The *registerGi* is preloaded from the memory octuple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.O (Section 6.12.141), lsucode6 = 0000

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	011	variant	registerGi	011	11	0000	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xplo`*variant registerGi*, *registerY* = *offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory octuple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.D (Section 6.12.142), lsucode6 = 0000

P	00	– –	extend27
95	94 93	92 91	90 64
1	00	– –	offset27
63	62 61	60 59	58 32

1	01	011	variant	registerGi	011	11	0000	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xplo`*variant registerGi*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory octuple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB (Section 6.12.143), lsucode6 = 0000

Syntax `xplo`*variant registerGj*, *registerY* = [*registerZ*]

Description The *registerGj* is preloaded from the memory octuple word at the effective address.

Modifier(s)

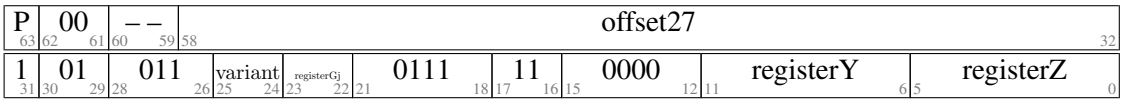
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.O (Section 6.12.144), lsucode6 = 0000



Syntax `xplo`*variant registerGj*, *registerY* = *offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory octuple word at the effective address.

Modifier(s)

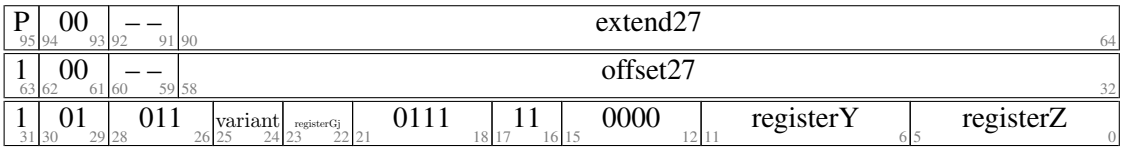
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.D (Section 6.12.145), lsucode6 = 0000



Syntax `xplo`*variant registerGj*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory octuple word at the effective address.

Modifier(s)

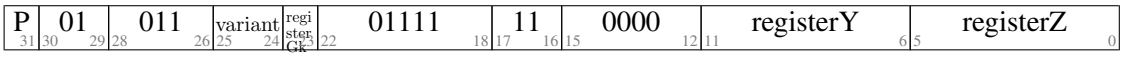
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB (Section 6.12.146), lsucode6 = 0000



Syntax `xplo`*variant registerGk*, *registerY* = [*registerZ*]

Description The *registerGk* is preloaded from the memory octuple word at the effective address.

Modifier(s)

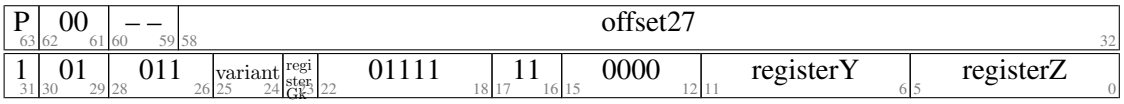
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.O (Section 6.12.147), lsucode6 = 0000



Syntax `xplo`*variant registerGk*, *registerY* = *offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory octuple word at the effective address.

Modifier(s)

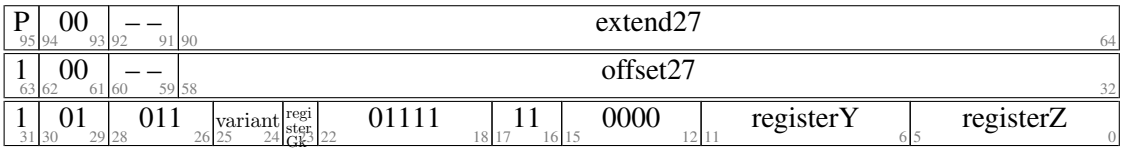
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.D (Section 6.12.148), lsucode6 = 0000



Syntax `xplo`*variant registerGk*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory octuple word at the effective address.

Modifier(s)

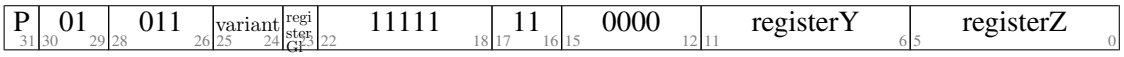
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB (Section 6.12.149), lsucode6 = 0000



Syntax `xplo`*variant registerGl*, *registerY* = [*registerZ*]

Description The *registerGl* is preloaded from the memory octuple word at the effective address.

Modifier(s)

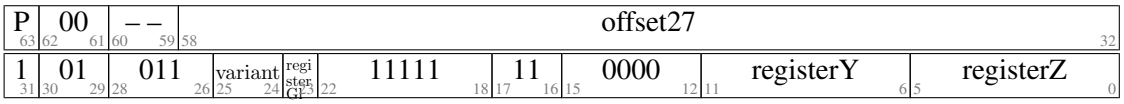
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.O (Section 6.12.150), lsucode6 = 0000



Syntax `xplo`*variant registerGl*, *registerY* = *offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory octuple word at the effective address.

Modifier(s)

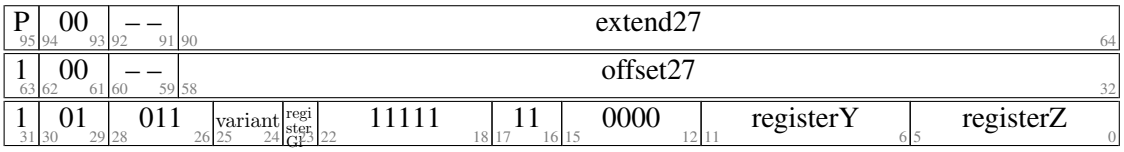
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.D (Section 6.12.151), lsucode6 = 0000



Syntax `xplo`*variant registerGl*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory octuple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 4294967295;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFFFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

6.5.44 XPLQ Extension Preload Quadruple Word

Encoding Format LSU_XPL2RBB (Section 6.12.134), lsucode6 = 0001

P	01	011	variant	registerGg	0	11	0001	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xplq`*variant registerGg*, *registerY* = [*registerZ*]

Description The *registerGg* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

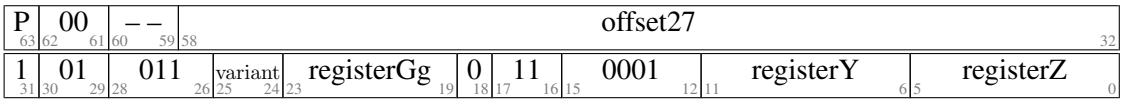
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.O (Section 6.12.135), lsucode6 = 0001



Syntax `xplq`*variant registerGg*, *registerY* = *offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

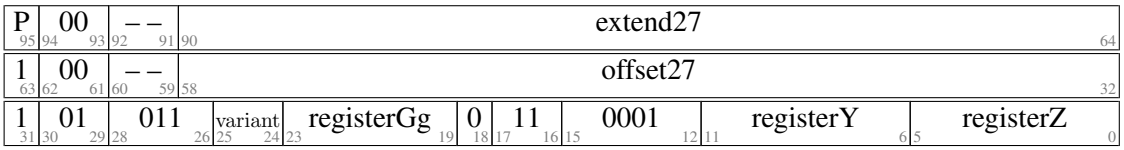
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.D (Section 6.12.136), lsucode6 = 0001



Syntax `xplq`*variant registerGg*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB (Section 6.12.137), lsucode6 = 0001

Syntax `xplq`*variant registerGh*, *registerY* = [*registerZ*]

Description The *registerGh* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.O (Section 6.12.138), lsucode6 = 0001

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	011	variant	registerGh	01	11	0001	registerY	registerZ
31	30 29	28 26	25 24	23 20	19 18	17 16	15 12	11 6	5 0

Syntax `xplq`*variant registerGh*, *registerY* = *offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.D (Section 6.12.139), lsucode6 = 0001

P 95	00 94 93	– – 92 91	extend27 90 64							
1 63	00 62 61	– – 60 59	offset27 58 32							
1 31	01 30 29	011 28 26	variant 25 24	registerGh 23 20	01 19 18	11 17 16	0001 15 12	registerY 11 6	registerZ 5 0	

Syntax `xplq`*variant registerGh*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB (Section 6.12.140), lsucode6 = 0001

P	01	011	variant	registerGi	011	11	0001	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xplq`*variant registerGi*, *registerY* = [*registerZ*]

Description The *registerGi* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

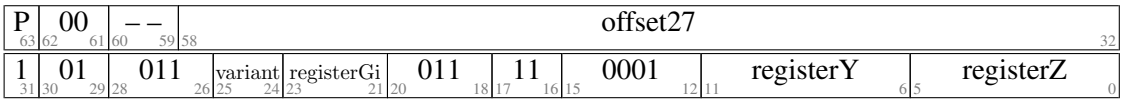
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.O (Section 6.12.141), lsucode6 = 0001



Syntax `xplq`*variant registerGi*, *registerY* = *offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

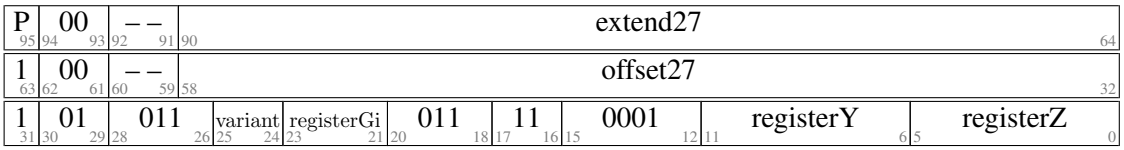
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.D (Section 6.12.142), lsucode6 = 0001



Syntax `xplq`*variant registerGi*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB (Section 6.12.143), lsucode6 = 0001

Syntax `xplq`*variant registerGj*, *registerY* = [*registerZ*]

Description The *registerGj* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

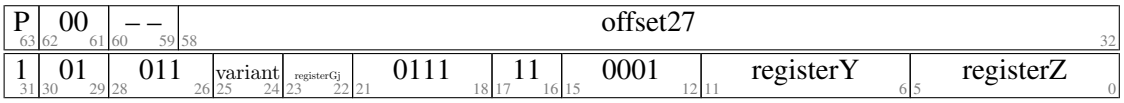
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.O (Section 6.12.144), lsucode6 = 0001



Syntax `xplq`*variant registerGj*, *registerY* = *offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

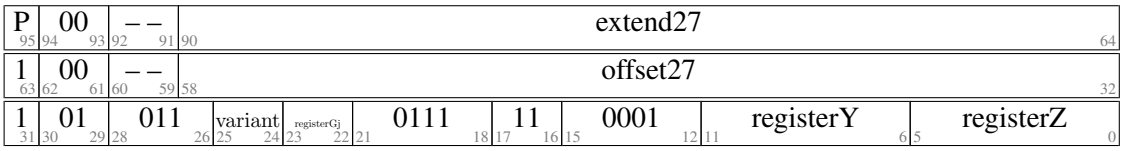
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.D (Section 6.12.145), lsucode6 = 0001



Syntax `xplq`*variant registerGj*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

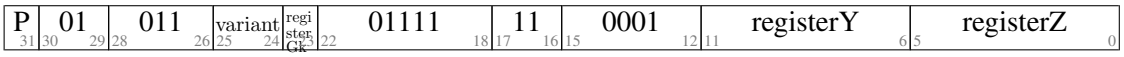
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB (Section 6.12.146), lsucode6 = 0001



Syntax `xplq`*variant registerGk*, *registerY* = [*registerZ*]

Description The *registerGk* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

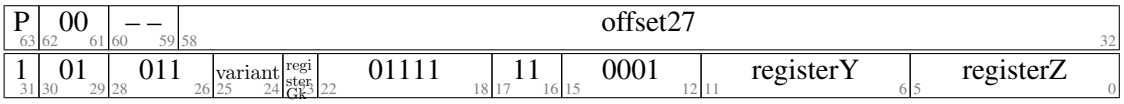
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.O (Section 6.12.147), lsucode6 = 0001



Syntax `xplq`*variant registerGk*, *registerY* = *offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

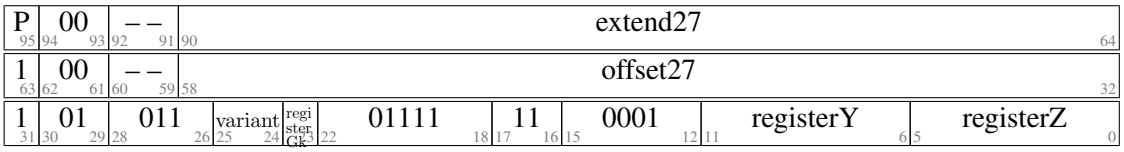
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.D (Section 6.12.148), lsucode6 = 0001



Syntax `xplq`*variant registerGk*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

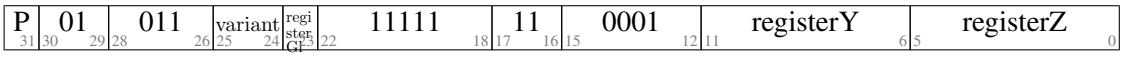
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB (Section 6.12.149), lsucode6 = 0001



Syntax `xplq`*variant registerGl*, *registerY* = [*registerZ*]

Description The *registerGl* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

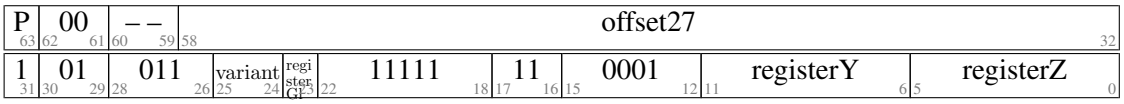
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.O (Section 6.12.150), lsucode6 = 0001



Syntax `xplq`*variant registerGl*, *registerY* = *offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

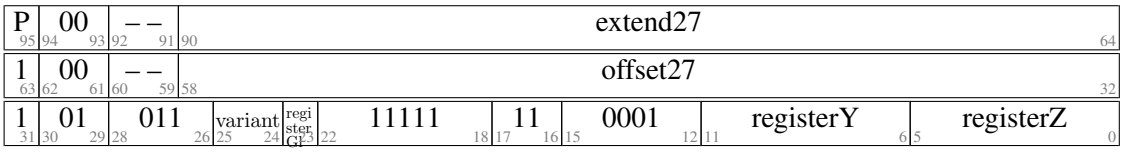
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.D (Section 6.12.151), lsucode6 = 0001



Syntax `xplq`*variant registerGl*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory quadruple word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 65535;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xFFFF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

6.5.45 XPLW Extension Preload Word

Encoding Format LSU_XPL2RBB (Section 6.12.134), lsucode6 = 0011

P	01	011	variant	registerGg	0	11	0011	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xplw`*variant registerGg*, *registerY* = [*registerZ*]

Description The *registerGg* is preloaded from the memory word at the effective address.

Modifier(s)

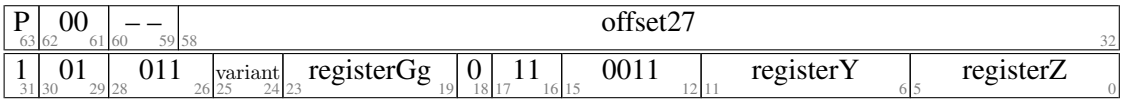
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.O (Section 6.12.135), lsucode6 = 0011



Syntax `xplw`*variant registerGg*, *registerY* = *offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory word at the effective address.

Modifier(s)

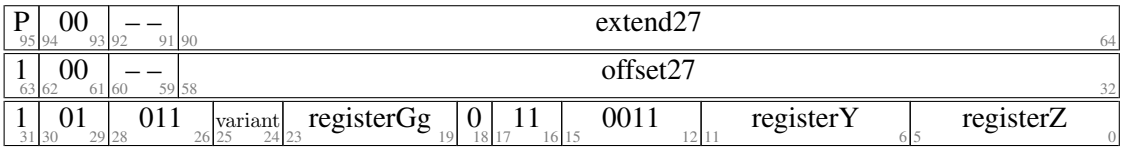
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL2RBB.D (Section 6.12.136), lsucode6 = 0011



Syntax `xplw`*variant registerGg*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGg* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 1;
  new where = (registerGg << 1) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB (Section 6.12.137), lsucode6 = 0011

Syntax `xplw`*variant registerGh*, *registerY* = [*registerZ*]

Description The *registerGh* is preloaded from the memory word at the effective address.

Modifier(s)

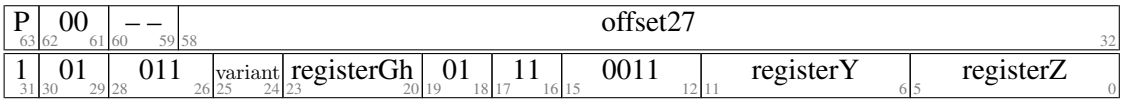
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.O (Section 6.12.138), lsucode6 = 0011



Syntax `xplw`*variant registerGh*, *registerY* = *offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL4RBB.D (Section 6.12.139), lsucode6 = 0011

P 95	00 94 93	– – 92 91	extend27 90 64							
1 63	00 62 61	– – 60 59	offset27 58 32							
1 31	01 30 29	011 28 26	variant 25 24	registerGh 23 20	01 19 18	11 17 16	0011 15 12	registerY 11 6	registerZ 5 0	

Syntax `xplw`*variant registerGh*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGh* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 3;
  new where = (registerGh << 2) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB (Section 6.12.140), lsucode6 = 0011

P	01	011	variant	registerGi	011	11	0011	registerY	registerZ
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0

Syntax `xplw`*variant registerGi*, *registerY* = [*registerZ*]

Description The *registerGi* is preloaded from the memory word at the effective address.

Modifier(s)

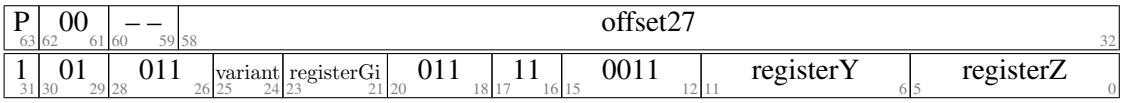
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL8RBB.O (Section 6.12.141), lsucode6 = 0011



Syntax `xplw`*variant registerGi*, *registerY* = *offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

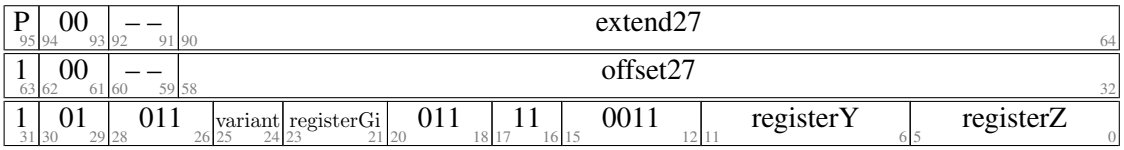
Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format LSU_XPL8RBB.D (Section 6.12.142), lsucode6 = 0011



Syntax `xplw`*variant registerGi*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGi* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 7;
  new where = (registerGi << 3) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB (Section 6.12.143), lsucode6 = 0011

Syntax `xplw`*variant registerGj*, *registerY* = [*registerZ*]

Description The *registerGj* is preloaded from the memory word at the effective address.

Modifier(s)

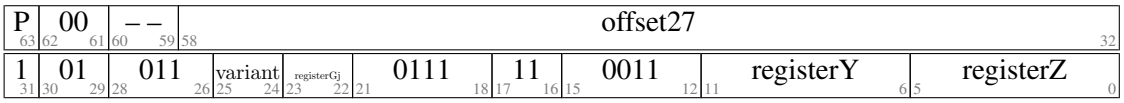
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.O (Section 6.12.144), lsucode6 = 0011



Syntax `xplw`*variant registerGj*, *registerY* = *offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL16RBB.D (Section 6.12.145), lsucode6 = 0011

P 95	00 94 93	– – 92 91	extend27 90 64							
1 63	00 62 61	– – 60 59	offset27 58 32							
1 31	01 30 29	011 28 26	variant 25 24	registerGj 23 22	0111 21 18	11 17 16	0011 15 12	registerY 11 6	registerZ 5 0	

Syntax `xplw`*variant registerGj*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGj* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 15;
  new where = (registerGj << 4) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB (Section 6.12.146), lsucode6 = 0011

P	01	011	variant	register Gk	01111	11	0011	registerY	registerZ
31	30 29	28 26	25 24	23 22	21 18	17 16	15 12	11 6	5 0

Syntax `xplw`*variant registerGk*, *registerY* = [*registerZ*]

Description The *registerGk* is preloaded from the memory word at the effective address.

Modifier(s)

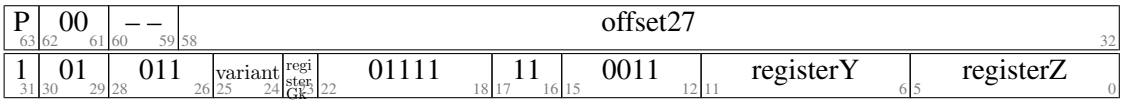
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.O (Section 6.12.147), lsucode6 = 0011



Syntax `xplw`*variant registerGk*, *registerY* = *offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory word at the effective address.

Modifier(s)

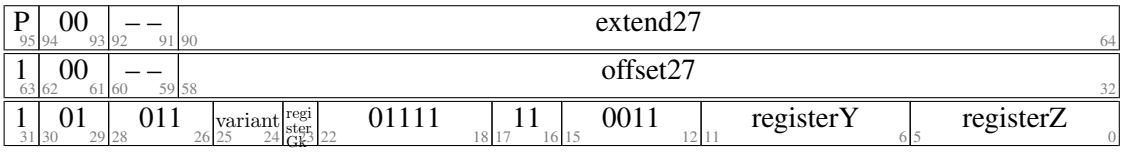
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL32RBB.D (Section 6.12.148), lsucode6 = 0011



Syntax `xplw`*variant registerGk*, *registerY* = *extend27_offset27*[*registerZ*]

Description The *registerGk* is preloaded from the memory word at the effective address.

Modifier(s)

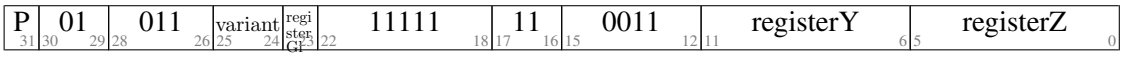
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.Y (Section 6.14.29)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 31;
  new where = (registerGk << 5) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB (Section 6.12.149), lsucode6 = 0011



Syntax `xplw`*variant registerGl*, *registerY* = [*registerZ*]

Description The *registerGl* is preloaded from the memory word at the effective address.

Modifier(s)

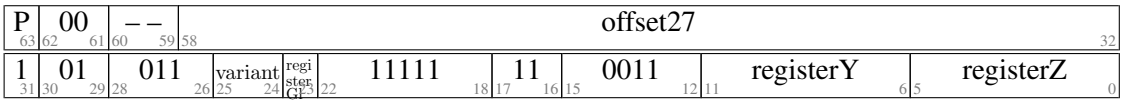
- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU (Section 6.14.27)

```
stage ID:
  new variant = _ZX_2(variant);
stage RR:
  new address = GPR[registerZ];
  new target = GPR[registerY];
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.O (Section 6.12.150), lsucode6 = 0011



Syntax `xplw`*variant registerGl*, *registerY* = *offset27*[*registerZ*]

Description The *registerGl* is preloaded from the memory word at the effective address.

Modifier(s)

- variant (cf. variant in section 6.10.2 on page 1432)

Execution Reservation table LSU.X (Section 6.14.28)

```
stage ID:
  new offset = _SX_27(offset27);
  new variant = _ZX_2(variant);
stage RR:
  new base = GPR[registerZ];
  new target = GPR[registerY];
  new address = base + offset;
  new shift = (target & 31) * 8;
  new index = (target >> 5) & 63;
  new where = (registerGl << 6) + index;
  new dri = where;
stage E1:
  new argument1 = XVR[where];
  new bytemask = 15;
  new targetbytes = bits2bytes(bytemask) << shift;
  CS.XMF = 1;
  new result1 = MEM.load(address, 0xF & bytemask, variant, dri);
  result1 = ((result1 << shift) & targetbytes) | (argument1 & ~targetbytes);
stage CR:
  XVR[where] = result1;
```


Kalray – all rights reserved

Encoding Format LSU_XPL64RBB.D (Section 6.12.151), lsucode6 = 0011

| | | |

6.5.46 XSO Extension Store Octuple Word

Encoding Format LSU_XSOBO (Section 6.12.99), steering = 01

P	01	10101	registerE	01	signed10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

Syntax `xso` *signed10*[*registerZ*] = *registerE*

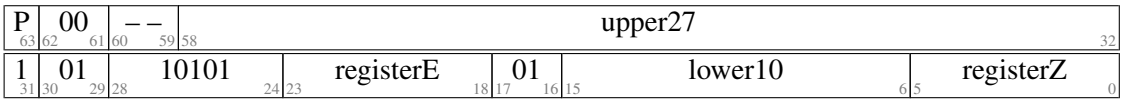
Description The effective address is computed by adding the *registerZ* to the *signed10*. The *registerE* is stored into the memory octuple word at the effective address.

Execution Reservation table LSU_CRRP (Section 6.14.39)

```
stage ID:
  new offset = _SX_10(signed10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerE;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = XVR[registerE];
  CS.XMF = 1;
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, dri);
```


Kalray – all rights reserved

Encoding Format LSU_XSOBO.X (Section 6.12.100), steering = 01



Syntax `xso` *upper27_lower10*[*registerZ*] = *registerE*

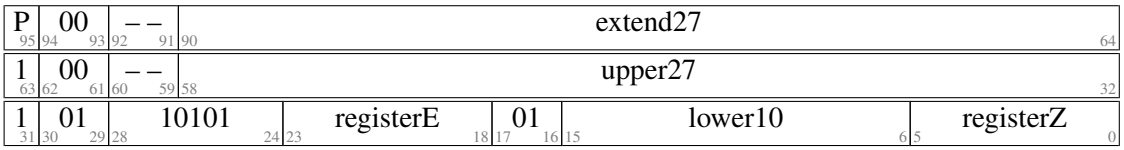
Description The effective address is computed by adding the *registerZ* to the *upper27_lower10*. The *registerE* is stored into the memory octuple word at the effective address.

Execution Reservation table LSU_CRRP.X (Section 6.14.40)

```
stage ID:
  new offset = _SX_37(upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerE;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = XVR[registerE];
  CS.XMF = 1;
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, dri);
```


Kalray – all rights reserved

Encoding Format LSU_XSOBO.Y (Section 6.12.101), steering = 01



Syntax `xso` *extend27_upper27_lower10*[*registerZ*] = *registerE*

Description The effective address is computed by adding the *registerZ* to the *extend27_upper27_lower10*. The *registerE* is stored into the memory octuple word at the effective address.

Execution Reservation table LSU_CRRP.Y (Section 6.14.41)

```
stage ID:
  new offset = _WX_64(extend27_upper27_lower10);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerE;
  new base = GPR[registerZ];
  new address = base + offset;
stage E1:
  new argument3 = XVR[registerE];
  CS.XMF = 1;
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, dri);
```


Kalray – all rights reserved

Encoding Format LSU_XSOMB (Section 6.12.164), steering = 01

P	01	10101	registerE	11	lsumask	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

Syntax `xso`*lsumask registerY*? [*registerZ*] = *registerE*

Description The effective address is given by the *registerZ*. The condition is computed on the *registerY* with the *lsumask*. The *registerE* is stored into the memory octuple word at the effective address.

Modifier(s)

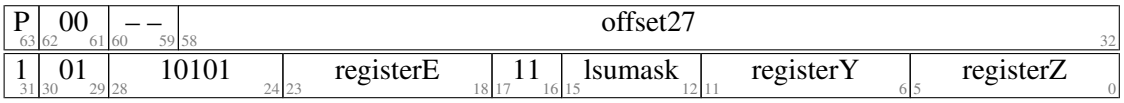
- lsumask (cf. lsumask in section 6.10.2 on page 1422)

Execution Reservation table LSU_CRRP (Section 6.14.39)

```
stage ID:
  new lsumask = _ZX_4(lsumask);
stage RR:
  new dri = registerE;
  new address = GPR[registerZ];
  new argument1 = GPR[registerY];
stage E1:
  new argument3 = XVR[registerE];
  new bytemask = lsumask_bytemask(lsumask, argument1);
  CS.XMF = 1;
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, dri);
```


Kalray – all rights reserved

Encoding Format LSU_XSOMB.O (Section 6.12.165), steering = 01



Syntax `xso`*lsumask registerY*? *offset27*[*registerZ*] = *registerE*

Description The effective address is given by adding the *offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsumask*. The *registerE* is stored into the memory octuple word at the effective address.

Modifier(s)

- lsumask (cf. lsumask in section 6.10.2 on page 1422)

Execution Reservation table LSU_CRRP.X (Section 6.14.40)

```
stage ID:
  new offset = _SX_27(offset27);
  new lsumask = _ZX_4(lsumask);
stage RR:
  new dri = registerE;
  new base = GPR[registerZ];
  new argument1 = GPR[registerY];
  new address = base + offset;
stage E1:
  new argument3 = XVR[registerE];
  new bytemask = lsumask_bytemask(lsumask, argument1);
  CS.XMF = 1;
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, dri);
```


Kalray – all rights reserved

Encoding Format LSU_XSOMB.D (Section 6.12.166), steering = 01

P 95	00 94 93	– – 92 91	extend27 90 64						
1 63	00 62 61	– – 60 59	offset27 58 32						
1 31	01 30 29	10101 28 24	registerE 23 18	11 17 16	lsumask 15 12	registerY 11 6	registerZ 5 0		

Syntax `xso`*lsumask registerY*? *extend27_offset27*[*registerZ*] = *registerE*

Description The effective address is given by adding the *extend27_offset27* to the *registerZ*. The condition is computed on the *registerY* with the *lsumask*. The *registerE* is stored into the memory octuple word at the effective address.

Modifier(s)

- lsumask (cf. lsumask in section 6.10.2 on page 1422)

Execution Reservation table LSU_CRRP.Y (Section 6.14.41)

```
stage ID:
  new offset = _SX_54(extend27_offset27);
  new lsumask = _ZX_4(lsumask);
stage RR:
  new dri = registerE;
  new base = GPR[registerZ];
  new argument1 = GPR[registerY];
  new address = base + offset;
stage E1:
  new argument3 = XVR[registerE];
  new bytemask = lsumask_bytemask(lsumask, argument1);
  CS.XMF = 1;
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, dri);
```


Kalray – all rights reserved

Encoding Format LSU_XSOBI (Section 6.12.194), steering = 01

Syntax `xso`*doscale registerY*[*registerZ*] = *registerE*

Description The effective address is computed by adding the *registerZ* to the *registerY* and scaled by the *doscale*. The *registerE* is stored into the memory octuple word at the effective address.

Modifier(s)

- doscale (cf. doscale in section 6.10.2 on page 1419)

Execution Reservation table LSU_CRRP (Section 6.14.39)

```
stage ID:
  new doscale = _ZX_1(doscale);
stage RR:
  new bytemask = 0xFFFFFFFF;
  new dri = registerE;
  new index = GPR[registerY];
  new base = GPR[registerZ];
  new scaling = doscale ? 32 : 1;
  new address = base + (index * scaling);
stage E1:
  new argument3 = XVR[registerE];
  CS.XMF = 1;
stage E3:
  MEM.store(address, 0xFFFFFFFF & bytemask, 0, argument3, dri);
```


Kalray – all rights reserved

6.6 ALU Instructions

6.6.1 ABDBO — Absolute Difference Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0010

P	11	1	0010	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdbo` *registerW* = *registerZ*, *registerY*

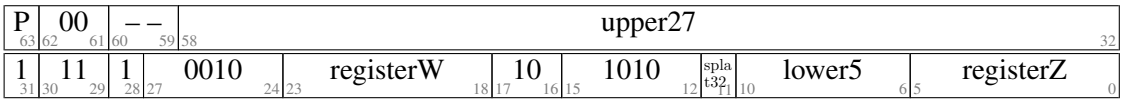
Description The *registerZ* and the *registerY* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *registerY* bytes. The absolute value of the resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _ABS(_SX_8(argument3.8[I]) - _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0010



Syntax `abdbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *upper27_lower5* bytes. The absolute value of the resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _ABS(_SX_8(argument3.8[I]) - _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.2 ABDD — Absolute Difference of Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0001

P	11	1	0001	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY*. The absolute value of the result is stored into the *registerW*.

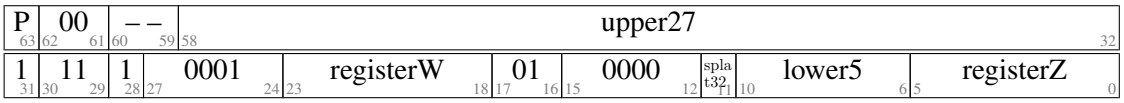
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_64(argument3) - _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0001



Syntax `abdd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is subtracted from the *upper27_lower5*. The absolute value of the result is stored into the *registerW*.

Modifier(s)

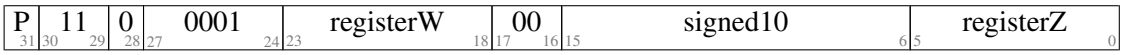
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_64(argument3) - _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 0001



Syntax `abdd` *registerW* = *registerZ*, *signed10*

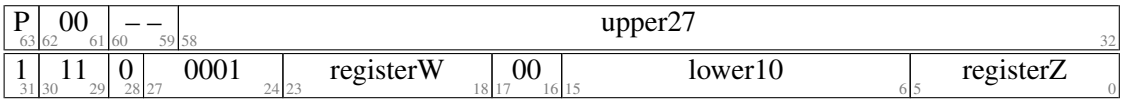
Description The *registerZ* is subtracted from the *signed10*. The absolute value of the result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_64(argument3) - _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 0001



Syntax `abdd` *registerW* = *registerZ*, *upper27_lower10*

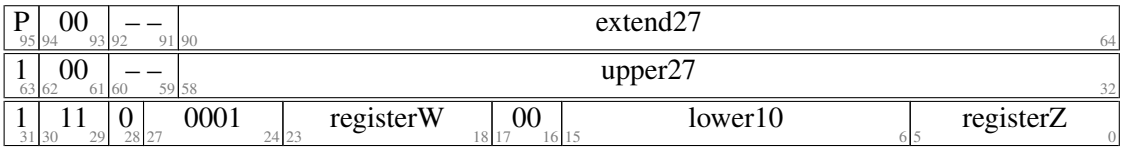
Description The *registerZ* is subtracted from the *upper27_lower10*. The absolute value of the result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_64(argument3) - _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 0001



Syntax `abdd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The *registerZ* is subtracted from the *extend27_upper27_lower10*. The absolute value of the result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_64(argument3) - _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.3 ABDHQ Absolute Difference Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0001

P	11	1	0001	registerW	01	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdhq` *registerW* = *registerZ*, *registerY*

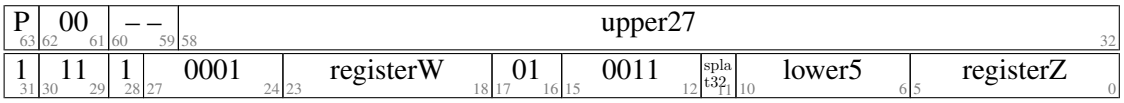
Description The *registerZ* and the *registerY* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted from the *registerY* half words. The absolute value of the four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ABS(_SX_16(argument3.16[I]) - _SX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0001



Syntax `abdhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted from the *upper27_lower5* half words. The absolute value of the four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ABS(_SX_16(argument3.16[I]) - _SX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.4 ABDSBO Absolute Difference Saturated of Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0011

P	11	1	0011	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdsbo` *registerW* = *registerZ*, *registerY*

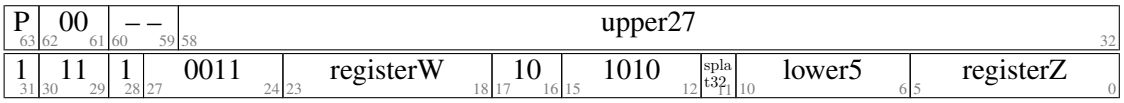
Description The *registerZ* and the *registerY* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *registerY* bytes. The absolute value of the resulting bytes are saturated, packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_ABS(_SX_8(argument3.8[I]) - _SX_8(argument2.8[I]))
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0011



Syntax `abdsbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *upper27_lower5* bytes. The absolute value of the resulting bytes are saturated, packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_ABS(_SX_8(argument3.8[I]) - _SX_8(argument2.8[I]))
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.5 ABDSD — Absolute Difference Saturated of Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0000

P	11	1	0000	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdsd` *registerW* = *registerZ*, *registerY*

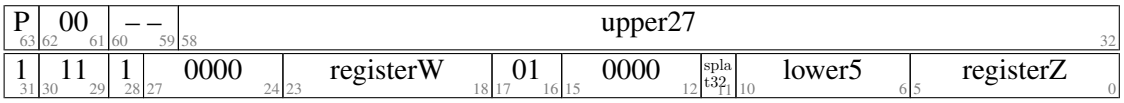
Description The *registerZ* is subtracted from the *registerY*. The absolute value of the result is saturated and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SAT_64(_ABS(_SX_64(argument3) - _SX_64(argument2)));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0000



Syntax `abdsd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is subtracted from the *upper27_lower5*. The absolute value of the result is saturated and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _SAT_64(_ABS(_SX_64(argument3) - _SX_64(argument2)));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.6 ABDSHQ Absolute Difference Saturated of Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0000

P	11	1	0000	registerW	01	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdshq` *registerW* = *registerZ*, *registerY*

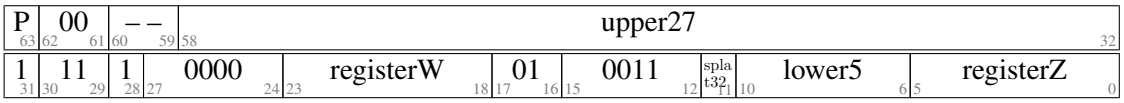
Description The *registerZ* and the *registerY* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted from the *registerY* half words. The absolute value of the four resulting half words are saturated, packed into 64 bits and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_ABS(_SX_16(argument3.16[I]) - _SX_16(argument2
        .16[I])))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0000



Syntax `abdshq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted from the *upper27_lower5* half words. The absolute value of the four resulting half words are saturated, packed into 64 bits and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_ABS(_SX_16(argument3.16[I]) - _SX_16(argument2
        .16[I])))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.7 ABDSW Absolute Difference Saturated of Words

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0000

P	11	1	0000	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdsw` *registerW* = *registerZ*, *registerY*

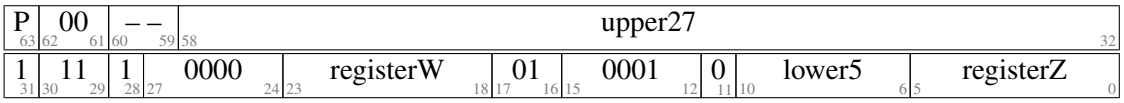
Description The *registerZ* is subtracted from the *registerY*. The absolute value of the result is saturated and stored into the *registerW* 32 upper bits cleared.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SAT_32(_ABS(_SX_32(argument3) - _SX_32(argument2)));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0.W (Section 6.12.224), exucode2 = 0000



Syntax `abdsw` *registerW = registerZ, upper27_lower5*

Description The *registerZ* is subtracted from the *upper27_lower5*. The absolute value of the result is saturated and stored into the *registerW* 32 upper bits cleared.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SAT_32(_ABS(_SX_32(argument3) - _SX_32(argument2)));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.8 ABDSWP Absolute Difference Saturated of Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0000

P	11	1	0000	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdswp` *registerW* = *registerZ*, *registerY*

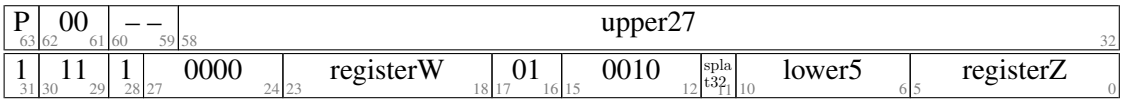
Description The *registerZ* and the *registerY* are considered as two words packed into 64 bits. The *registerZ* words are subtracted from the *registerY* words. The absolute value of the two resulting words are saturated, packed into 64 bits and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_ABS(_SX_32(argument3.32[I]) - _SX_32(argument2
        .32[I])))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0000



Syntax `abdswp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as two words packed into 64 bits. The *registerZ* words are subtracted from the *upper27_lower5* words. The absolute value of the two resulting words are saturated, packed into 64 bits and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_ABS(_SX_32(argument3.32[I]) - _SX_32(argument2
        .32[I])))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.9 ABDUBO Absolute Difference Byte Octuple

Encoding Format ALU_BOWRRS (Section 6.12.284), exucode2 = 1111

P	11	1	1111	registerW	10	1100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdubo` *registerW* = *registerZ*, *registerY*

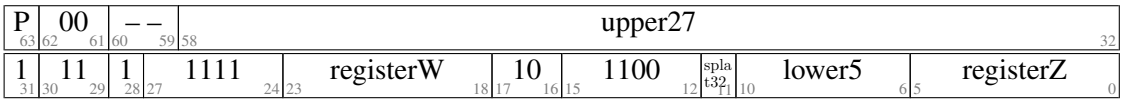
Description The *registerZ* and the *registerY* are considered as eight unsigned bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *registerY* bytes. The absolute value of the resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _ABS(_ZX_8(argument3.8[I]) - _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRRS.M (Section 6.12.285), exucode2 = 1111



Syntax `abdubo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as eight unsigned bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *upper27_lower5* bytes. The absolute value of the resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _ABS(_ZX_8(argument3.8[I]) - _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.10 ABDUD — Absolute Difference Unsigned of Double Words

Encoding Format ALU_DWRRS (Section 6.12.242), exucode2 = 1111

P	11	1	1111	registerW	01	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdud` *registerW* = *registerZ*, *registerY*

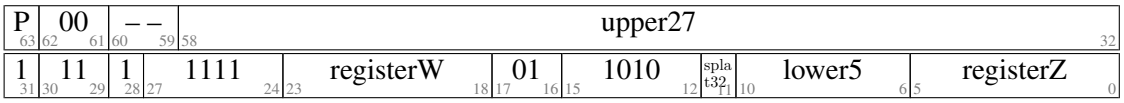
Description The *registerZ* is subtracted from the *registerY*, assuming unsigned integers. The absolute value of the result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_ZX_64(argument3) - _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRRS.M (Section 6.12.243), exucode2 = 1111



Syntax `abdud` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is subtracted from the *upper27_lower5*, assuming unsigned integers. The absolute value of the result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

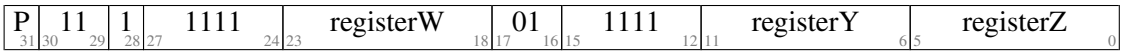
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_ZX_64(argument3) - _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.11 ABDUHQ Absolute Difference Unsigned Half Words Quadruple

Encoding Format ALU_HQWRRS (Section 6.12.254), exucode2 = 1111



Syntax `abduhq` *registerW* = *registerZ*, *registerY*

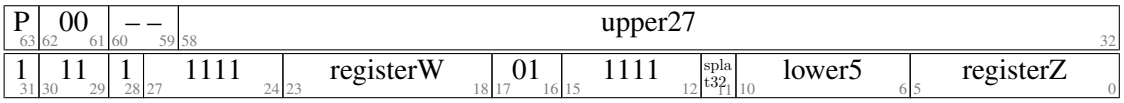
Description The *registerZ* and the *registerY* are considered as four unsigned half words packed into 64 bits. The *registerZ* half words are subtracted from the *registerY* half words. The absolute value of the four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ABS(_ZX_16(argument3.16[I]) - _ZX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRRS.M (Section 6.12.255), exucode2 = 1111



Syntax `abduhq` *registerW = registerZ, upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as four unsigned half words packed into 64 bits. The *registerZ* half words are subtracted from the *upper27_lower5* half words. The absolute value of the four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ABS(_ZX_16(argument3.16[I]) - _ZX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.12 ABDUW Absolute Difference Unsigned of Words

Encoding Format ALU_WRRSW (Section 6.12.246), exucode2 = 1111

P	11	1	1111	registerW	01	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abduw` *registerW* = *registerZ*, *registerY*

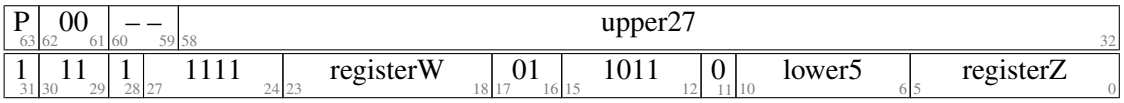
Description The *registerZ* is subtracted from the *registerY*, assuming unsigned integers. The absolute value of the result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_ZX_32(argument3) - _ZX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRRSW.W (Section 6.12.247), exucode2 = 1111



Syntax `abduw` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is subtracted from the *upper27_lower5*, assuming unsigned integers. The absolute value of the result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_ZX_32(argument3) - _ZX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.13 ABDUWP Absolute Difference Unsigned Words Packed

Encoding Format ALU_WPWRRS (Section 6.12.250), exucode2 = 1111

P	11	1	1111	registerW	01	1110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abduwp` *registerW* = *registerZ*, *registerY*

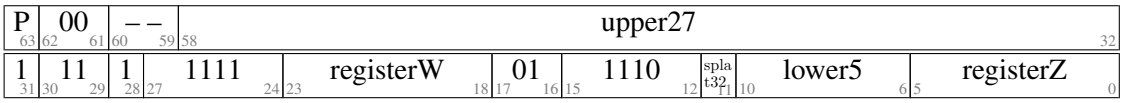
Description The *registerZ* and the *registerY* are considered as two unsigned words packed into 64 bits. The *registerZ* words are subtracted from the *registerY* words. The absolute value of the two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ABS(_ZX_32(argument3.32[I]) - _ZX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRRS.M (Section 6.12.251), exucode2 = 1111



Syntax `abduwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as two unsigned words packed into 64 bits. The *registerZ* words are subtracted from the *upper27_lower5* words. The absolute value of the two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ABS(_ZX_32(argument3.32[I]) - _ZX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.14 ABDW — Absolute Difference of Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 0001

P	11	1	0001	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `abdw` *registerW* = *registerZ*, *signed10*

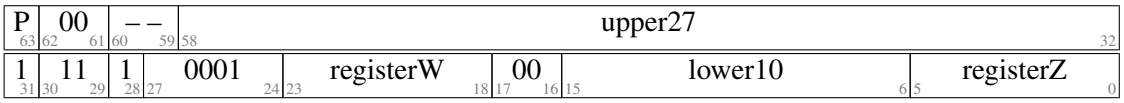
Description The *registerZ* is subtracted from the *signed10*. The absolute value of the result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_32(argument3) - _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 0001



Syntax `abdw` *registerW* = *registerZ*, *upper27_lower10*

Description The *registerZ* is subtracted from the *upper27_lower10*. The absolute value of the result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_32(argument3) - _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0001

P	11	1	0001	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY*. The absolute value of the result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ABS(_SX_32(argument3) - _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.15 ABDWP — Absolute Difference Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0001

P	11	1	0001	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `abdwp` *registerW* = *registerZ*, *registerY*

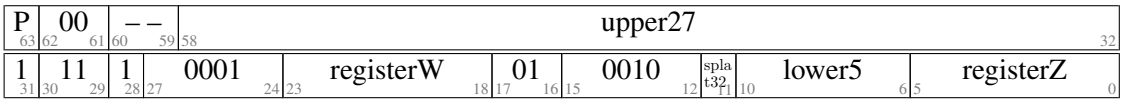
Description The *registerZ* and the *registerY* are considered as two words packed into 64 bits. The *registerZ* words are subtracted from the *registerY* words. The absolute value of the two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ABS(_SX_32(argument3.32[I]) - _SX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0001



Syntax `abdwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as two words packed into 64 bits. The *registerZ* words are subtracted from the *upper27_lower5* words. The absolute value of the two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ABS(_SX_32(argument3.32[I]) - _SX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.16 ADDBO — Add Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0000

P	11	1	0000	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addbo` *registerW* = *registerZ*, *registerY*

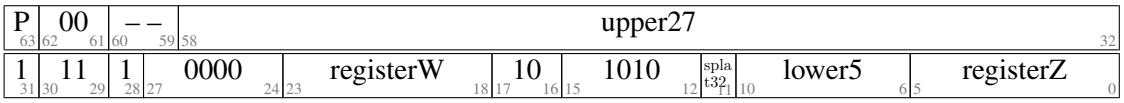
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerY* bytes and the *registerZ* bytes are added. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = argument3.8[I] + argument2.8[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0000



Syntax `addbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *upper27_lower5* bytes and the *registerZ* bytes are added. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = argument3.8[I] + argument2.8[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.17 ADDCD — Add Double Word to Double Word with Carry

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1110

P	11	1	1110	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addcd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is added to the *registerY* and the **IC** bit of the **CS** register. The result is stored into the *registerW* and the **IC** bit is updated.

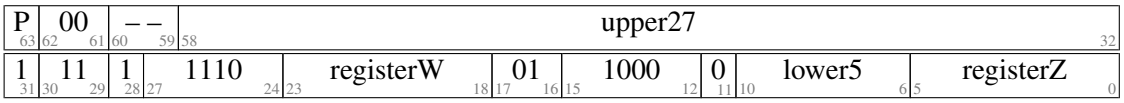
Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_64(argument2) + _ZX_64(argument3)) + CS.IC;
  new carry = (result1 >> 64) & 1;
  new counter = CS.CC;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  CS.CC = counter + carry;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1110



Syntax `addcd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is added to the *upper27_lower5* and the **IC** bit of the **CS** register. The result is stored into the *registerW* and the **IC** bit is updated.

Execution Reservation table ALU_FULL.X (Section 6.14.3)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_64(argument2) + _ZX_64(argument3)) + CS.IC;
  new carry = (result1 >> 64) & 1;
  new counter = CS.CC;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  CS.CC = counter + carry;
```


Kalray – all rights reserved

6.6.18 ADDCD.I Add Double Word to Double Word with Carry Initial

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1110

P	11	1	1110	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addcd.i` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is added to the *registerY*. The result is stored into the *registerW* and the **IC** bit of the **CS** register is updated.

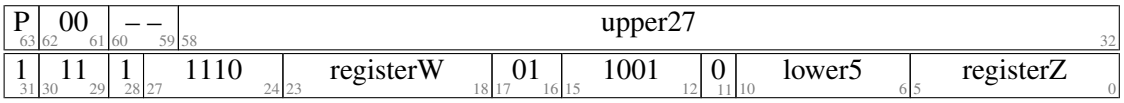
Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2) + _ZX_64(argument3);
  new carry = (result1 >> 64) & 1;
  new counter = CS.CC;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  CS.CC = counter + carry;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1110



Syntax `addcd.i` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is added to the *upper27_lower5*. The result is stored into the *registerW* and the **IC** bit of the **CS** register is updated.

Execution Reservation table ALU_FULL.X (Section 6.14.3)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2) + _ZX_64(argument3);
  new carry = (result1 >> 64) & 1;
  new counter = CS.CC;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  CS.CC = counter + carry;
```


Kalray – all rights reserved

6.6.19 ADDD — Add Double Word to Double Word

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0010

P	11	1	0010	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addd` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are added. The result is stored into the *registerW*.

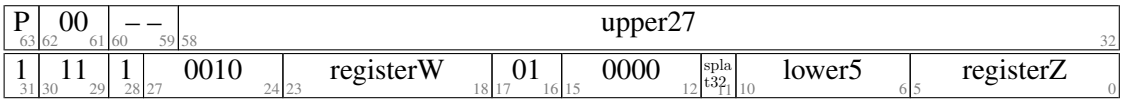
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0010



Syntax `addd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are added. The result is stored into the *registerW*.

Modifier(s)

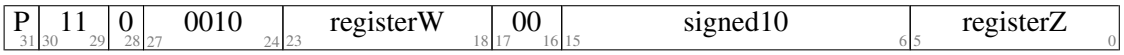
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 0010



Syntax `addd` *registerW* = *registerZ*, *signed10*

Description The *signed10* and the *registerZ* are added. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 0010

P	00	– –	upper27				
63	62 61	60 59	58 32				
1	11	0	0010	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `addd` *registerW* = *registerZ*, *upper27_lower10*

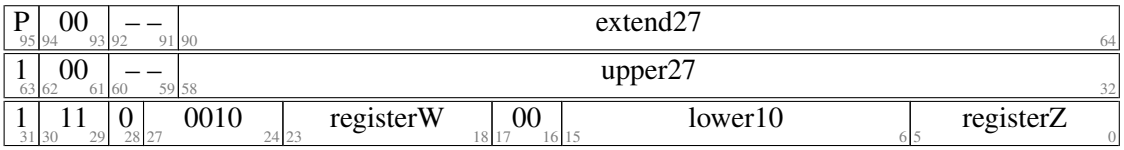
Description The *upper27_lower10* and the *registerZ* are added. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 0010



Syntax `addd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The *extend27_upper27_lower10* and the *registerZ* are added. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.20 ADDHQ — Add Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0010

P	11	1	0010	registerW	01	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addhq` *registerW* = *registerZ*, *registerY*

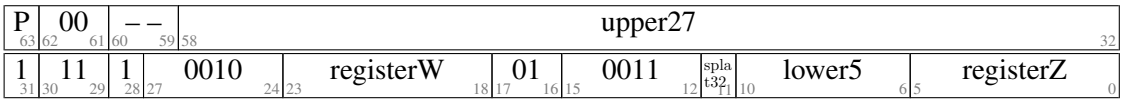
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are added. The four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument3.16[I] + argument2.16[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0010



Syntax `addhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are added. The four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

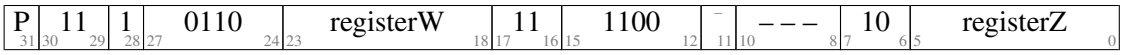
```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument3.16[I] + argument2.16[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.21 ADDRBOD — Add Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 0110



Syntax `addrbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The sum of these eight sign-extended bytes is computed and stored into the *registerW*.

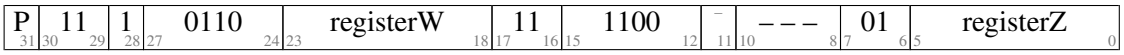
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = _SX_8(argument2.8[0]) + _SX_8(argument2.8[1]);
  new result1_y = _SX_8(argument2.8[2]) + _SX_8(argument2.8[3]);
  new result1_z = _SX_8(argument2.8[4]) + _SX_8(argument2.8[5]);
  new result1_t = _SX_8(argument2.8[6]) + _SX_8(argument2.8[7]);
  new result1_l = result1_x + result1_y;
  new result1_m = result1_z + result1_t;
  new result1 = result1_l + result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.22 ADDRHQD Add Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 0110



Syntax `addrhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The sum of these four sign-extended half words is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = _SX_16(argument2.16[0]) + _SX_16(argument2.16[1]);
  new result1_m = _SX_16(argument2.16[2]) + _SX_16(argument2.16[3]);
  new result1 = result1_l + result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.23 ADDRWPD Add Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 0110

P	11	1	0110	registerW	11	1100	–	– – –	00	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `addrwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The sum of these two sign-extended words is computed and stored into the *registerW*.

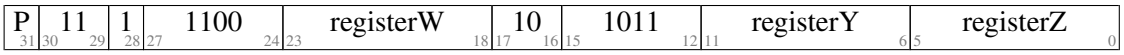
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _SX_32(argument2.32[0]) + _SX_32(argument2.32[1]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.24 ADDSBO — Add Saturated Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 1100



Syntax `addsbo` *registerW* = *registerZ*, *registerY*

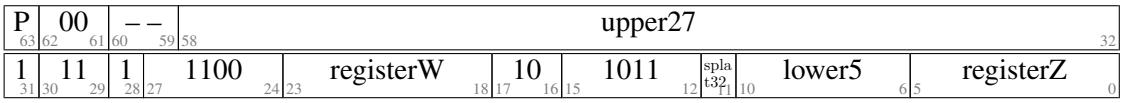
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerY* bytes and the *registerZ* bytes are added with saturation to 8 bits. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_SX_8(argument3.8[I]) + _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 1100



Syntax `addsbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *upper27_lower5* bytes and the *registerZ* bytes are added with saturation to 8 bits. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_SX_8(argument3.8[I]) + _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.25 ADDSD — Add Saturated Double Words

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1100

P	11	1	1100	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addsd` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are added with 64-bit saturation.

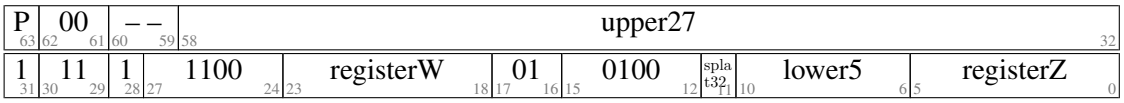
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SAT_64(_SX_64(argument3) + _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1100



Syntax `addsd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are added with 64-bit saturation.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _SAT_64(_SX_64(argument3) + _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.26 ADDSHQ — Add Saturated Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 1100

P	11	1	1100	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addshq` *registerW* = *registerZ*, *registerY*

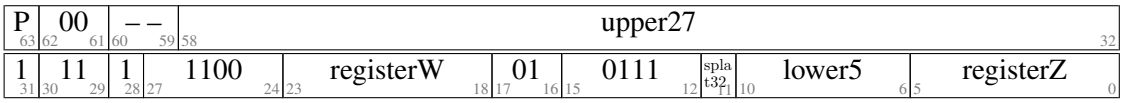
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are added with saturation to 16 bits. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_SX_16(argument3.16[I]) + _SX_16(argument2.16[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 1100



Syntax `addshq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are added with saturation to 16 bits. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

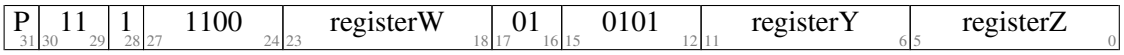
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_SX_16(argument3.16[I]) + _SX_16(argument2.16[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.27 ADDSW — Add Saturated Words

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1100



Syntax `addsw` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are added with 32-bit saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

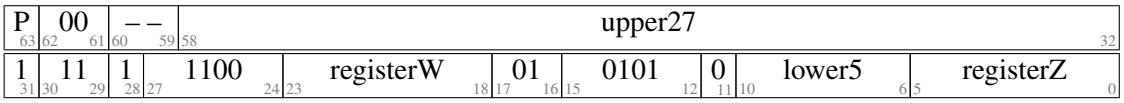
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SAT_32(_SX_32(argument3) + _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1100



Syntax `addsw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are added with 32-bit saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SAT_32(_SX_32(argument3) + _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.28 ADDSWP — Add Saturated Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 1100

P	11	1	1100	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addswp` *registerW* = *registerZ*, *registerY*

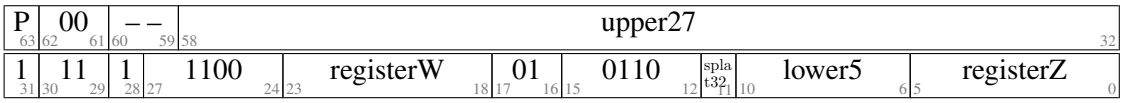
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are added with saturation to 32 bits. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_SX_32(argument3.32[I]) + _SX_32(argument2.32[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 1100



Syntax `addswp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are added with saturation to 32 bits. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

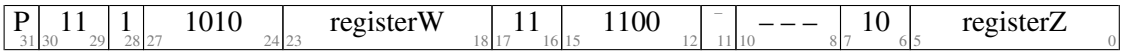
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_SX_32(argument3.32[I]) + _SX_32(argument2.32[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.29 ADDURBOD Add Unsigned Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 1010



Syntax `addurbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The sum of these eight zero-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = argument2.8[0] + argument2.8[1];
  new result1_y = argument2.8[2] + argument2.8[3];
  new result1_z = argument2.8[4] + argument2.8[5];
  new result1_t = argument2.8[6] + argument2.8[7];
  new result1_l = result1_x + result1_y;
  new result1_m = result1_z + result1_t;
  new result1 = result1_l + result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.30 ADDURHQD Add Unsigned Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 1010

P	11	1	1010	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `addurhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The sum of these four zero-extended half words is computed and stored into the *registerW*.

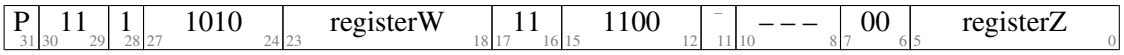
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = _ZX_16(argument2.16[0]) + _ZX_16(argument2.16[1]);
  new result1_m = _ZX_16(argument2.16[2]) + _ZX_16(argument2.16[3]);
  new result1 = result1_l + result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.31 ADDURWPD Add Unsigned Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 1010



Syntax `addurwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The sum of these two zero-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] + argument2.32[1];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.32 ADDUSBO Add Unsigned Saturated Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 1110

P	11	1	1110	registerW	10	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addusbo` *registerW* = *registerZ*, *registerY*

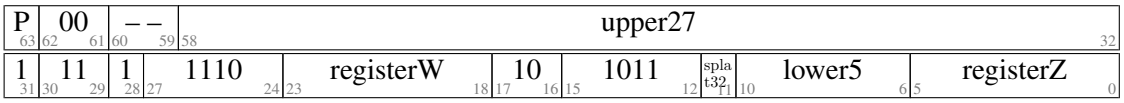
Description The *registerY* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *registerY* bytes and the *registerZ* bytes are added with unsigned saturation to 8 bits. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SATU_8(_ZX_8(argument3.8[I]) + _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 1110



Syntax `addusbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *upper27_lower5* bytes and the *registerZ* bytes are added with unsigned saturation to 8 bits. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SATU_8(_ZX_8(argument3.8[I]) + _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.33 ADDUSD Add Unsigned Saturated Double Words

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1110

P	11	1	1110	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addusd` *registerW* = *registerZ*, *registerY*

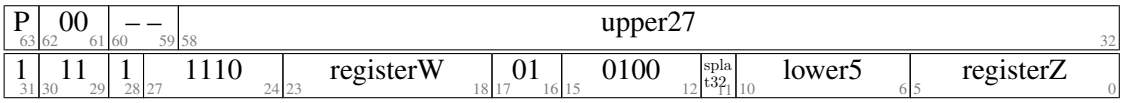
Description The *registerY* and the *registerZ* are added with 64-bit unsigned saturation.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SATU_64(_ZX_64(argument3) + _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1110



Syntax `addusd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are added with 64-bit unsigned saturation.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _SATU_64(_ZX_64(argument3) + _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.34 ADDUSHQ Add Unsigned Saturated Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 1110

P	11	1	1110	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addushq` *registerW* = *registerZ*, *registerY*

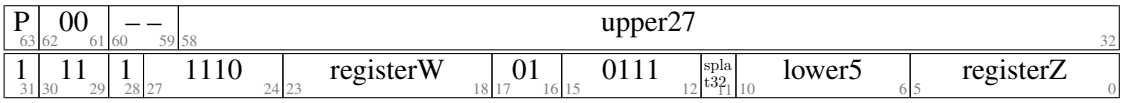
Description The *registerY* and the *registerZ* are considered as four unsigned half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are added with unsigned saturation to 16 bits. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SATU_16(_ZX_16(argument3.16[I]) + _ZX_16(argument2.16[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 1110



Syntax `addushq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four unsigned half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are added with unsigned saturation to 16 bits. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SATU_16(_ZX_16(argument3.16[I]) + _ZX_16(argument2.16[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.35 ADDUSW Add Unsigned Saturated Words

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1110

P	11	1	1110	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addusw` *registerW* = *registerZ*, *registerY*

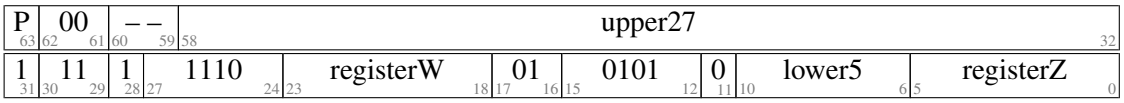
Description The *registerY* and the *registerZ* are added with 32-bit unsigned saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SATU_32(_ZX_32(argument3) + _ZX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1110



Syntax `addusw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are added with 32-bit unsigned saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SATU_32(_ZX_32(argument3) + _ZX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.36 ADDUSWP Add Unsigned Saturated Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 1110

P	11	1	1110	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `adduswp` *registerW* = *registerZ*, *registerY*

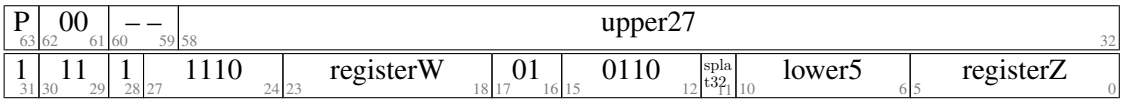
Description The *registerY* and the *registerZ* are considered as two unsigned words packed into 64 bits. The *registerY* words and the *registerZ* words are added with unsigned saturation to 32 bits. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SATU_32(_ZX_32(argument3.32[I]) + _ZX_32(argument2.32[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 1110



Syntax `adduswp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two unsigned words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are added with unsigned saturation to 32 bits. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SATU_32(_ZX_32(argument3.32[I]) + _ZX_32(argument2.32[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.37 ADDUWD — Add Unsigned Word to Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1100

P	11	1	1100	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `adduwd` *registerW* = *registerZ*, *registerY*

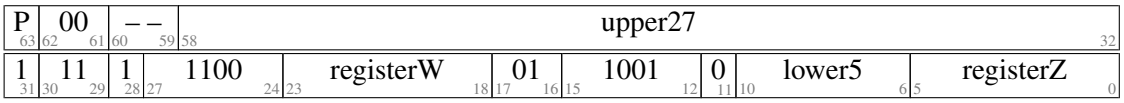
Description The *registerZ* is zero-extended from 32 bits and added to the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + argument2.32[0];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1100



Syntax `adduwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + argument2.32[0];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.38 ADDW — Add Word to Word

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 0010

P	11	1	0010	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `addw` *registerW* = *registerZ*, *signed10*

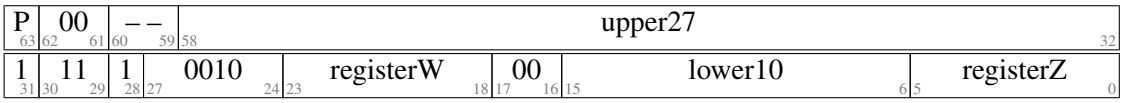
Description The *signed10* and the *registerZ* are added. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 0010



Syntax `addw` *registerW* = *registerZ*, *upper27_lower10*

Description The *upper27_lower10* and the *registerZ* are added. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0010

P	11	1	0010	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addw` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are added. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + argument2;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.39 ADDWD — Add Word to Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1100

P	11	1	1100	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addwd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is sign-extended from 32 bits and added to the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + _SX_32(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1100

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1100	registerW	01	1000	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `addwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + _SX_32(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.40 ADDWP — Add Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0010

P	11	1	0010	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addwp` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are added. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument3.32[I] + argument2.32[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0010

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	11	1	0010				registerW						01	0010			splat32	lower5					registerZ									
31	30	29	28	27						24	23				18	17	16	15			12	11	10			6	5					0

Syntax `addwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are added. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

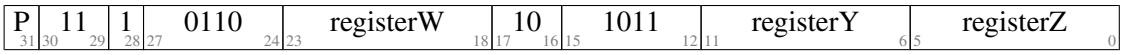
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument3.32[I] + argument2.32[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.41 ADDX16BO Add Integer to Integer Times 16 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0110



Syntax `addx16bo` *registerW* = *registerZ*, *registerY*

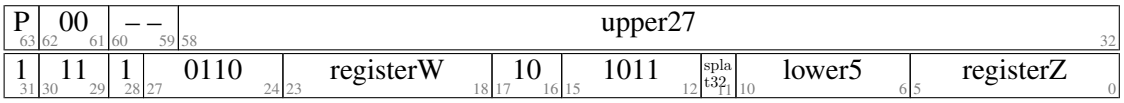
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by four bits and added to the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0110



Syntax `addx16bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by four bits and added to the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.42 ADDX16D — Add Double Word to Double Word Times 16

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0110

P	11	1	0110	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx16d` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by four bits and added to the *registerY*. The result is stored into the *registerW*.

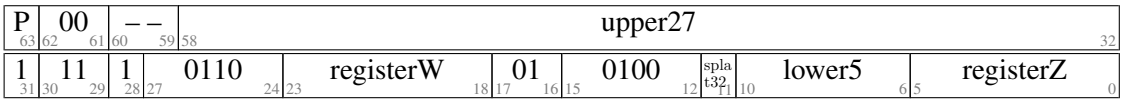
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0110



Syntax `addx16d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by four bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

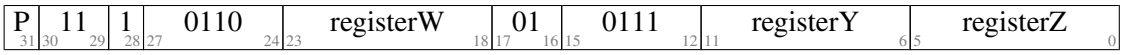
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.43 ADDX16HQ Add Integer to Integer Times 16 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0110



Syntax `addx16hq` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by four bits and added to the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0110

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	11	1	0110				registerW					01	0111			splat32	lower5						registerZ									
31	30	29	28	27						24	23				18	17	16	15			12	11	10				6	5					0

Syntax `addx16hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by four bits and added to the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.44 ADDX16UWD Add Unsigned Word Times 16 to Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0110

P	11	1	0110	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx16uwd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is zero-extended from 32 bits, shifted left by four bits and added to the *registerY*. The result is stored into the *registerW*.

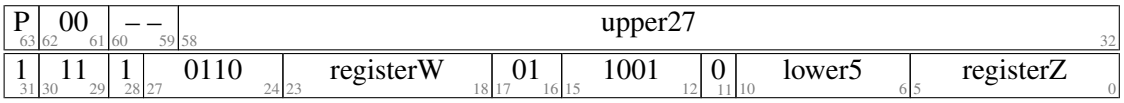
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0110



Syntax `addx16uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by four bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

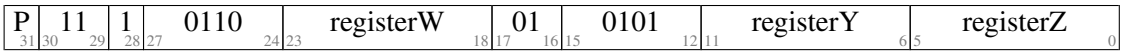
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.45 ADDX16W — Add Word to Word Times 16

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0110



Syntax `addx16w` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by four bits and added to the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

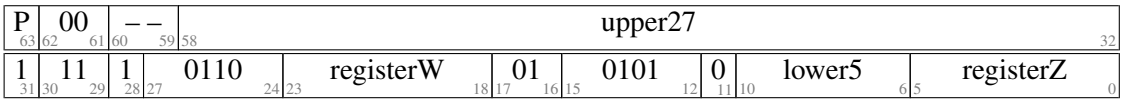
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 4);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0110



Syntax `addx16w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by four bits and added to the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 4);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.46 ADDX16WD — Add Word Times 16 to Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0110

P	11	1	0110	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx16wd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is sign-extended from 32 bits, shifted left by four bits and added to the *registerY*. The result is stored into the *registerW*.

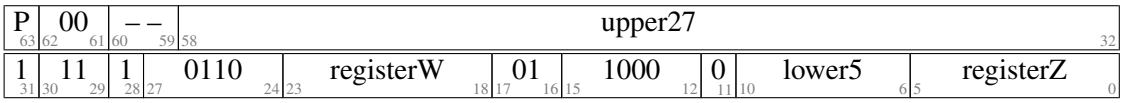
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0110



Syntax `addx16wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by four bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.47 ADDX16WP Add Integer to Integer Times 16 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0110

P	11	1	0110	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx16wp` *registerW* = *registerZ*, *registerY*

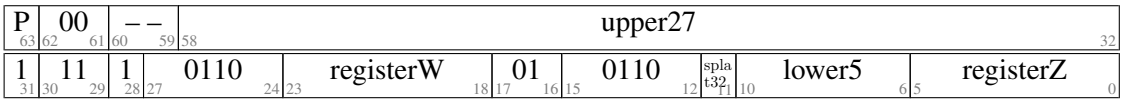
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by four bits and added to the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0110



Syntax `addx16wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by four bits and added to the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

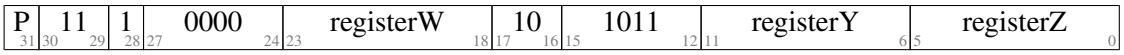
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.48 ADDX2BO — Add Integer to Integer Times 2 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0000



Syntax `addx2bo` *registerW* = *registerZ*, *registerY*

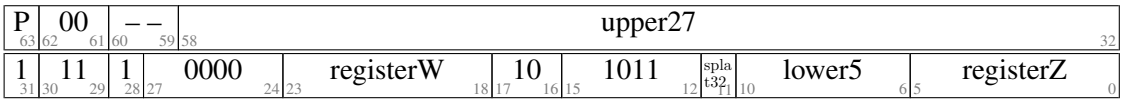
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by one bit and added to the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0000



Syntax `addx2bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by one bit and added to the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.49 ADDX2D — Add Double Word to Double Word Times 2

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0000

P	11	1	0000	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx2d` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by one bit and added to the *registerY*. The result is stored into the *registerW*.

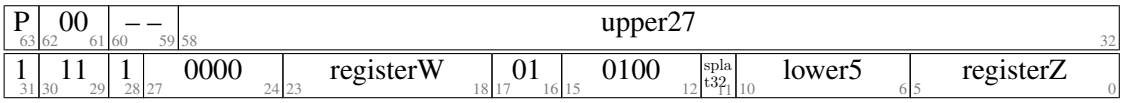
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0000



Syntax `addx2d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by one bit and added to the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.50 ADDX2HQ Add Integer to Integer Times 2 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0000

P	11	1	0000	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx2hq` *registerW* = *registerZ*, *registerY*

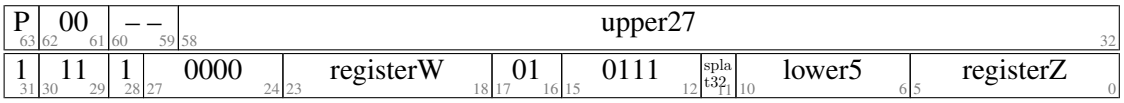
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by one bit and added to the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0000



Syntax `addx2hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by one bit and added to the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.51 ADDX2UWD — Add Unsigned Word Times 2 to Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0000

P	11	1	0000	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx2uwd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is zero-extended from 32 bits, shifted left by one bit and added to the *registerY*. The result is stored into the *registerW*.

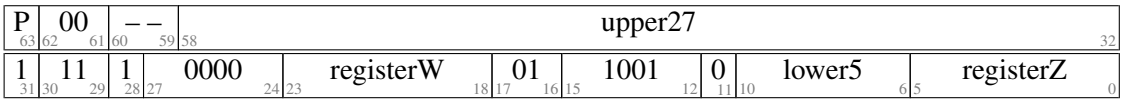
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0000



Syntax `addx2uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by one bit and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.52 ADDX2W — Add Word to Word Times 2

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0000

P	11	1	0000	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx2w` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by one bit and added to the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

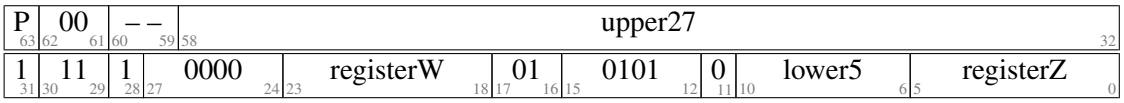
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 1);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0000



Syntax `addx2w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by one bit and added to the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 1);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.53 ADDX2WD — Add Word Times 2 to Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0000

P	11	1	0000	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx2wd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is sign-extended from 32 bits, shifted left by one bit and added to the *registerY*. The result is stored into the *registerW*.

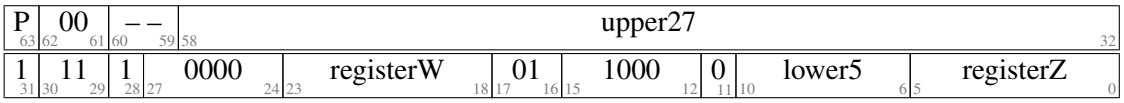
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0000



Syntax `addx2wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by one bit and added to the *upper27_lower5*. The result is stored into the *registerW*.

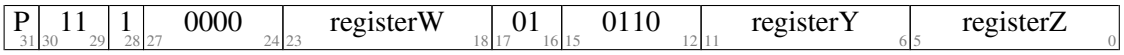
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.54 ADDX2WP Add Integer to Integer Times 2 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0000



Syntax `addx2wp` *registerW* = *registerZ*, *registerY*

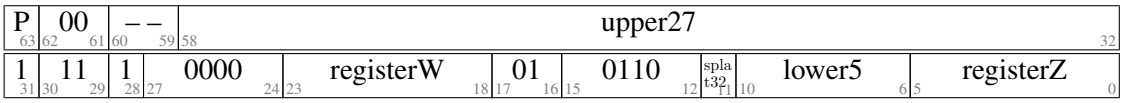
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by one bit and added to the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0000



Syntax `addx2wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by one bit and added to the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.55 ADDX32D — Add Double Word to Double Word Times 32

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1000

P	11	1	1000	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx32d` *registerW* = *registerZ*, *registerY*

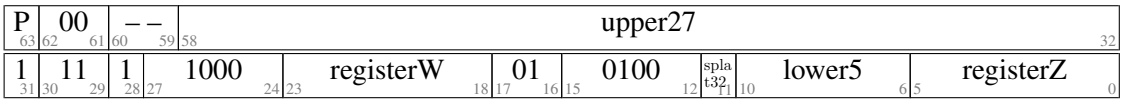
Description The *registerZ* is shifted left by five bits and added to the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1000



Syntax `addx32d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by five bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.56 ADDX32UWD Add Unsigned Word Times 32 to Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1000

P	11	1	1000	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx32uwd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is zero-extended from 32 bits, shifted left by five bits and added to the *registerY*. The result is stored into the *registerW*.

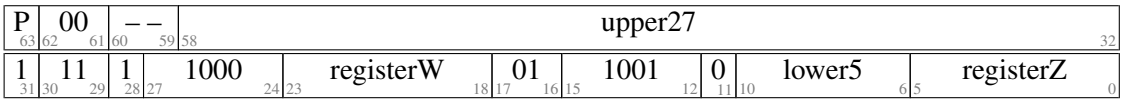
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1000



Syntax `addx32uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by five bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.57 ADDX32W — Add Word to Word Times 32

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1000

P	11	1	1000	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx32w` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by five bits and added to the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

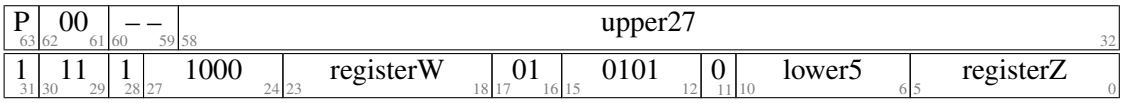
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 5);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1000



Syntax `addx32w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by five bits and added to the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 5);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.58 ADDX32WD — Add Word Times 32 to Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1000

P	11	1	1000	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx32wd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is sign-extended from 32 bits, shifted left by five bits and added to the *registerY*. The result is stored into the *registerW*.

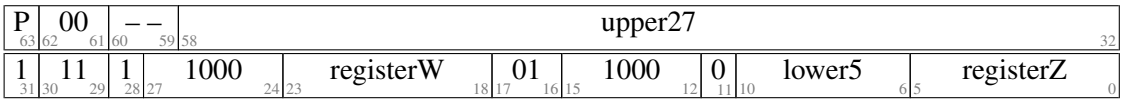
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1000



Syntax `addx32wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by five bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

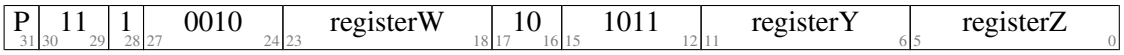
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.59 ADDX4BO — Add Integer to Integer Times 4 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0010



Syntax `addx4bo` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by two bits and added to the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

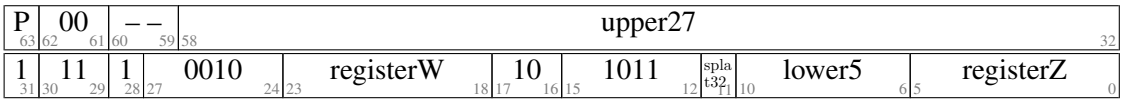
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0010



Syntax `addx4bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by two bits and added to the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.60 ADDX4D — Add Double Word to Double Word Times 4

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0010

P	11	1	0010	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx4d` *registerW* = *registerZ*, *registerY*

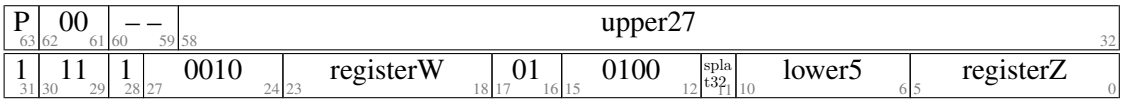
Description The *registerZ* is shifted left by two bits and added to the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0010



Syntax `addx4d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by two bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.61 ADDX4HQ Add Integer to Integer Times 4 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0010

P	11	1	0010	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx4hq` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by two bits and added to the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0010

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	11	1	0010				registerW					01	0111			splat32	lower5					registerZ										
31	30	29	28	27					24	23		18	17	16	15	12	11	10			6	5										0

Syntax `addx4hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by two bits and added to the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.62 ADDX4UWD — Add Unsigned Word Times 4 to Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0010

P	11	1	0010	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx4uwd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is zero-extended from 32 bits, shifted left by two bits and added to the *registerY*. The result is stored into the *registerW*.

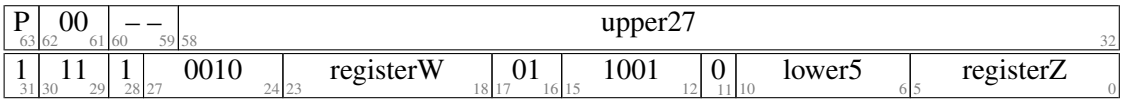
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0010



Syntax `addx4uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by two bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.63 ADDX4W — Add Word to Word Times 4

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0010

P	11	1	0010	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx4w` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by two bits and added to the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

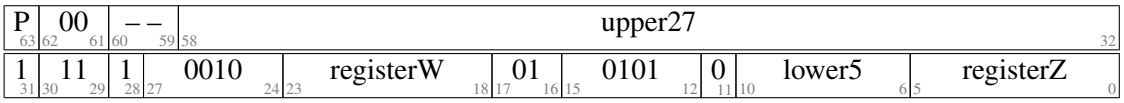
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 2);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0010



Syntax `addx4w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by two bits and added to the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 2);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.64 ADDX4WD — Add Word Times 4 to Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0010

P	11	1	0010	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx4wd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is sign-extended from 32 bits, shifted left by two bits and added to the *registerY*. The result is stored into the *registerW*.

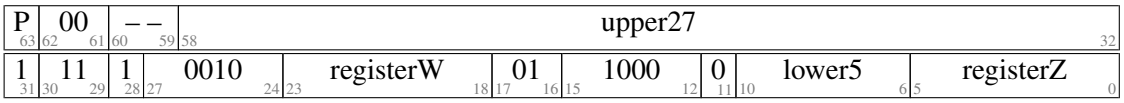
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0010



Syntax `addx4wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by two bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.65 ADDX4WP Add Integer to Integer Times 4 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0010

P	11	1	0010	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx4wp` *registerW* = *registerZ*, *registerY*

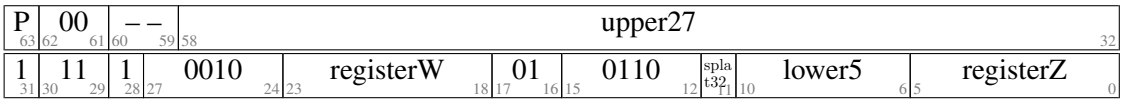
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by two bits and added to the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0010



Syntax `addx4wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by two bits and added to the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.66 ADDX64D — Add Double Word to Double Word Times 64

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1010

P	11	1	1010	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx64d` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by six bits and added to the *registerY*. The result is stored into the *registerW*.

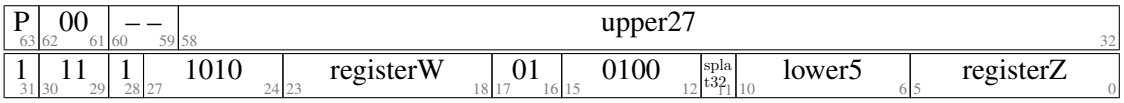
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1010



Syntax `addx64d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by six bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.67 ADDX64UWD Add Unsigned Word Times 64 to Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1010

P	11	1	1010	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx64uwd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is zero-extended from 32 bits, shifted left by six bits and added to the *registerY*. The result is stored into the *registerW*.

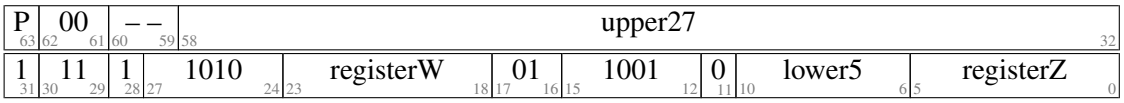
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1010



Syntax `addx64uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by six bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.68 ADDX64W — Add Word to Word Times 64

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1010

P	11	1	1010	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx64w` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by six bits and added to the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

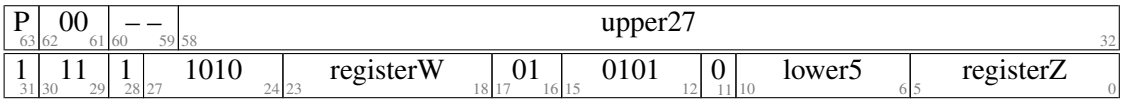
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 6);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1010



Syntax `addx64w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by six bits and added to the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 6);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.69 ADDX64WD — Add Word Times 64 to Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1010

P	11	1	1010	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx64wd` *registerW* = *registerZ*, *registerY*

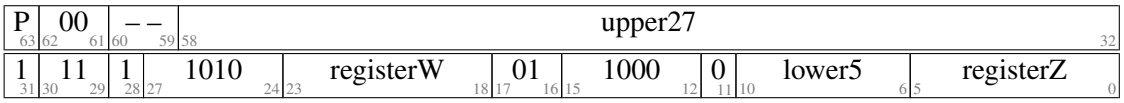
Description The *registerZ* is sign-extended from 32 bits, shifted left by six bits and added to the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1010



Syntax `addx64wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by six bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.70 ADDX8BO — Add Integer to Integer Times 8 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0100

P	11	1	0100	registerW	10	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx8bo` *registerW* = *registerZ*, *registerY*

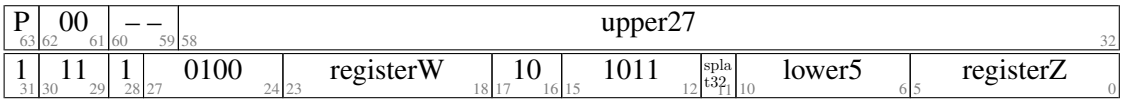
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by three bits and added to the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0100



Syntax `addx8bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by three bits and added to the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) + (_SX_8(argument2.8[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.71 ADDX8D — Add Double Word to Double Word Times 8

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0100

P	11	1	0100	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx8d` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by three bits and added to the *registerY*. The result is stored into the *registerW*.

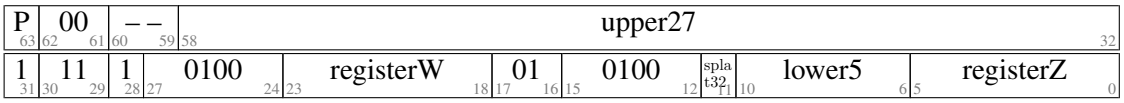
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0100



Syntax `addx8d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by three bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

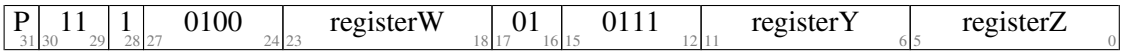
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.72 ADDX8HQ Add Integer to Integer Times 8 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0100



Syntax `addx8hq` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by three bits and added to the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0100

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0100	registerW	01	0111	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `addx8hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by three bits and added to the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) + (_SX_16(argument2.16[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.73 ADDX8UWD Add Unsigned Word Times 8 to Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0100

P	11	1	0100	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx8uwd` *registerW* = *registerZ*, *registerY*

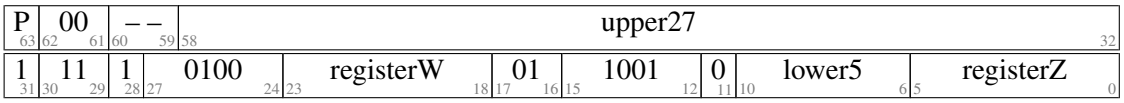
Description The *registerZ* is zero-extended from 32 bits, shifted left by three bits and added to the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0100



Syntax `addx8uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by three bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (argument2.32[0] << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.74 ADDX8W — Add Word to Word Times 8

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0100

P	11	1	0100	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx8w` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by three bits and added to the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0100

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0100	registerW	01	0101	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `addx8w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by three bits and added to the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 + (argument2 << 3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.75 ADDX8WD — Add Word Times 8 to Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0100

P	11	1	0100	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx8wd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is sign-extended from 32 bits, shifted left by three bits and added to the *registerY*. The result is stored into the *registerW*.

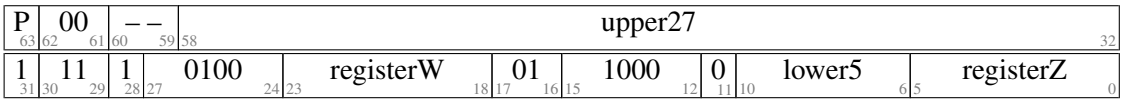
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0100



Syntax `addx8wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by three bits and added to the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) + (_SX_32(argument2) << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.76 ADDX8WP Add Integer to Integer Times 8 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0100

P	11	1	0100	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `addx8wp` *registerW* = *registerZ*, *registerY*

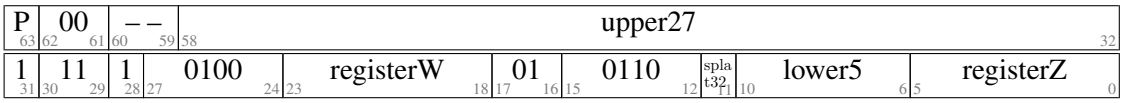
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by three bits and added to the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0100



Syntax `addx8wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by three bits and added to the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) + (_SX_32(argument2.32[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.77 ANDD — Bitwise And Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1000

P	11	1	1000	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `andd` *registerW* = *registerZ*, *registerY*

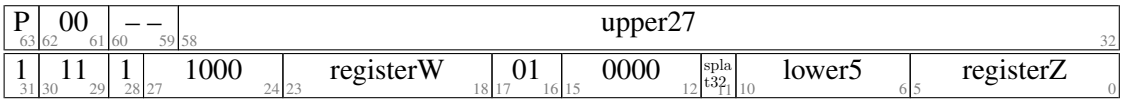
Description Bitwise and of the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1000



Syntax `andd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise and of the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

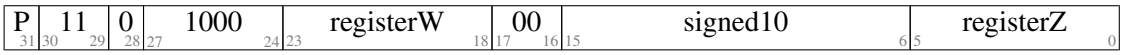
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1000



Syntax `andd` *registerW* = *registerZ*, *signed10*

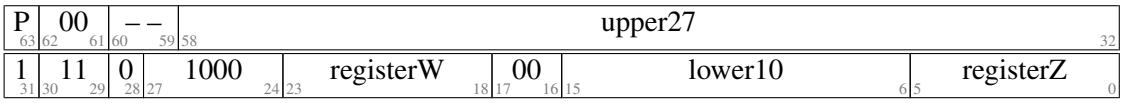
Description Bitwise and of the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1000



Syntax `andd` *registerW* = *registerZ*, *upper27_lower10*

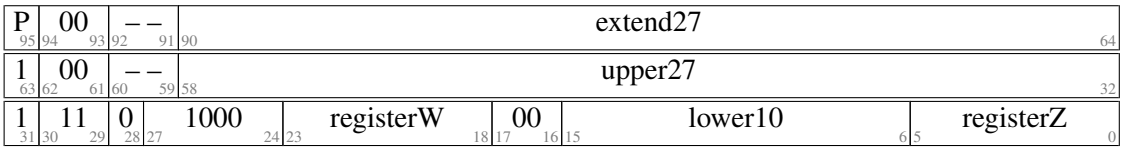
Description Bitwise and of the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1000



Syntax `andd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description Bitwise and of the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.78 ANDND Bitwise And with Not Operand Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1110

P	11	1	1110	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `andnd` *registerW* = *registerZ*, *registerY*

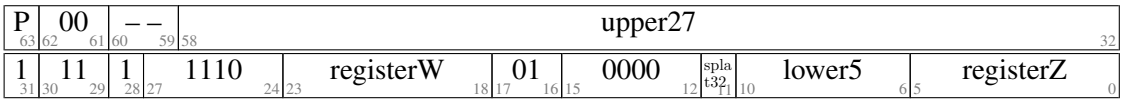
Description Bitwise and of not the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1110



Syntax `andnd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise and of not the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1110

P	11	0	1110	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `andnd` *registerW* = *registerZ*, *signed10*

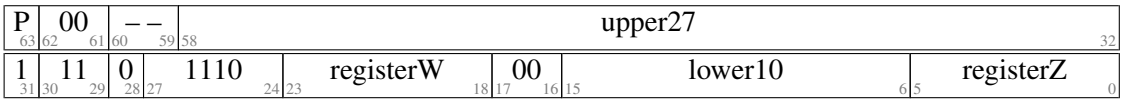
Description Bitwise and of not the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1110



Syntax `andnd` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise and of not the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1110

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	11 30 29	0 28	1110 27 24	registerW 23 18	00 17 16	lower10 15 6	registerZ 5 0

Syntax `andnd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description Bitwise and of not the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

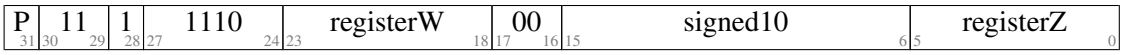
Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.79 ANDNW Bitwise And with Not Operand Between Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1110



Syntax `andnw` *registerW* = *registerZ*, *signed10*

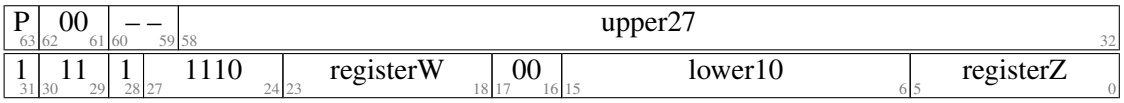
Description Bitwise and of not the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1110



Syntax `andnw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise and of not the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1110

P	11	1	1110	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `andnw` *registerW* = *registerZ*, *registerY*

Description Bitwise and of not the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 & argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.80 ANDRBOD Bitwise AND Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 0000

P	11	1	0000	registerW	11	1100	–	– – –	10	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `andrbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The bitwise AND of these eight zero-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = argument2.8[0] & argument2.8[1];
  new result1_y = argument2.8[2] & argument2.8[3];
  new result1_z = argument2.8[4] & argument2.8[5];
  new result1_t = argument2.8[6] & argument2.8[7];
  new result1_l = result1_x & result1_y;
  new result1_m = result1_z & result1_t;
  new result1 = result1_l & result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.81 ANDRHQD Bitwise AND Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 0000

P	11	1	0000	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `andrhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The bitwise AND of these four zero-extended half words is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = argument2.16[0] & argument2.16[1];
  new result1_m = argument2.16[2] & argument2.16[3];
  new result1 = result1_l & result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.82 ANDRWPD Bitwise AND Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 0000

P	11	1	0000	registerW	11	1100	–	– – –	00	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `andrwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The bitwise AND of these two zero-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] & argument2.32[1];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.83 ANDW — Bitwise And Between Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1000

P	11	1	1000	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `andw` *registerW* = *registerZ*, *signed10*

Description Bitwise and of the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

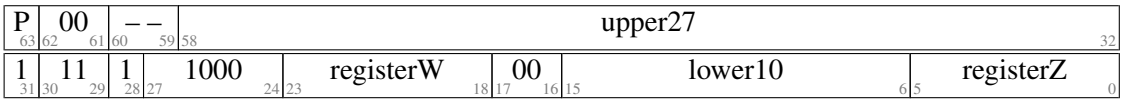
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1000



Syntax `andw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise and of the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1000

Syntax `andw` *registerW* = *registerZ*, *registerY*

Description Bitwise and of the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2 & argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.84 AVGBO Average of Byte Octuple

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 0100

P	11	1	0100	registerW	10	1111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgbo` *registerW* = *registerZ*, *registerY*

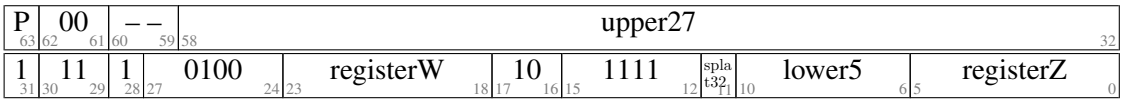
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerY* bytes and the *registerZ* bytes are added. Each sum is shifted right arithmetic one bit. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = (_SX_8(argument3.8[I]) + _SX_8(argument2.8[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR.M (Section 6.12.290), exucode2 = 0100



Syntax `avgbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *upper27_lower5* bytes and the *registerZ* bytes are added. Each sum is shifted right arithmetic one bit. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = (_SX_8(argument3.8[I]) + _SX_8(argument2.8[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.85 AVGHQ Average of Half Words Quadruple

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 0100

P	11	1	0100	registerW	10	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avghq` *registerW* = *registerZ*, *registerY*

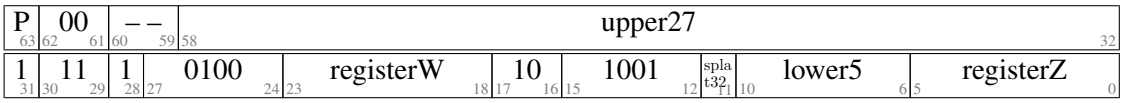
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are added. Each sum is shifted right arithmetic one bit. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = (_SX_16(argument3.16[I]) + _SX_16(argument2.16[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR.M (Section 6.12.277), exucode2 = 0100



Syntax `avghq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are added. Each sum is shifted right arithmetic one bit. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = (_SX_16(argument3.16[I]) + _SX_16(argument2.16[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.86 AVGRBO Average Rounded of Byte Octuple

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 0101

P	11	1	0101	registerW	10	1111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgrbo` *registerW* = *registerZ*, *registerY*

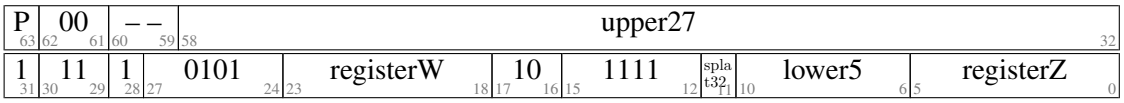
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerY* bytes and the *registerZ* bytes are added. Each sum is incremented and shifted right arithmetic one bit. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = ((_SX_8(argument3.8[I]) + _SX_8(argument2.8[I])) + 1) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR.M (Section 6.12.290), exucode2 = 0101



Syntax `avgrbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *upper27_lower5* bytes and the *registerZ* bytes are added. Each sum is incremented and shifted right arithmetic one bit. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = ((_SX_8(argument3.8[I]) + _SX_8(argument2.8[I])) + 1) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.87 AVGRHQ Average Rounded of Half Words Quadruple

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 0101

P 31	11 30 29	1 28	0101 27 24	registerW 23 18	10 17 16	1001 15 12	registerY 11 6	registerZ 5 0

Syntax `avgrhq` *registerW* = *registerZ*, *registerY*

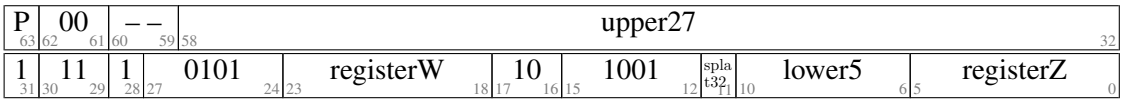
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are added. Each sum is incremented and shifted right arithmetic one bit. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = ((_SX_16(argument3.16[I]) + _SX_16(argument2.16[I])) + 1)
        >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR.M (Section 6.12.277), exucode2 = 0101



Syntax `avgrhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are added. Each sum is incremented and shifted right arithmetic one bit. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = ((_SX_16(argument3.16[I]) + _SX_16(argument2.16[I])) + 1)
        >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.88 AVGRUBO Average Rounded Unsigned of Byte Octuple

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 0111

P	11	1	0111	registerW	10	1111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgrubo` *registerW* = *registerZ*, *registerY*

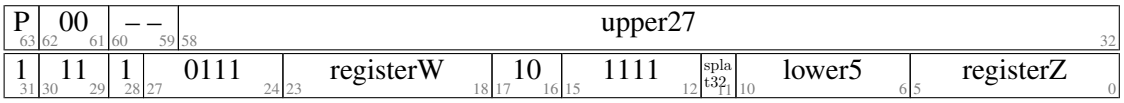
Description The *registerY* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *registerY* bytes and the *registerZ* bytes are added. Each sum is incremented and shifted right logical one bit. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = ((_ZX_8(argument3.8[I]) + _ZX_8(argument2.8[I])) + 1) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR.M (Section 6.12.290), exucode2 = 0111



Syntax `avgrubo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *upper27_lower5* bytes and the *registerZ* bytes are added. Each sum is incremented and shifted right logical one bit. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = ((_ZX_8(argument3.8[I]) + _ZX_8(argument2.8[I])) + 1) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.89 AVGRUHQ Average Rounded Unsigned of Half Words Quadruple

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 0111

P	11	1	0111	registerW	10	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgruhq` *registerW* = *registerZ*, *registerY*

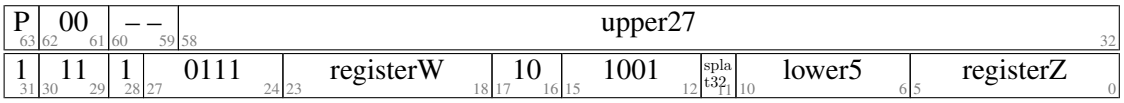
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are added. Each sum is incremented and shifted right logical one bit. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = ((_ZX_16(argument3.16[I]) + _ZX_16(argument2.16[I])) + 1)
        >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR.M (Section 6.12.277), exucode2 = 0111



Syntax `avgruhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are added. Each sum is incremented and shifted right logical one bit. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = ((_ZX_16(argument3.16[I]) + _ZX_16(argument2.16[I])) + 1)
        >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.90 AVGRUW Average Rounded of Unsigned Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0111

P	11	1	0111	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgruw` *registerW* = *registerZ*, *registerY*

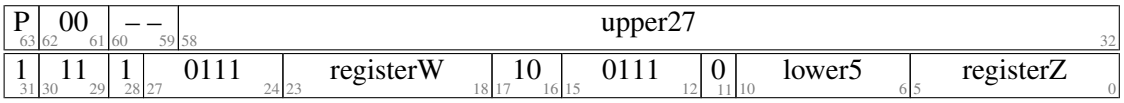
Description The *registerY* and the *registerZ* are added. The result is incremented, shifted right logical one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ((argument3.32[0] + argument2.32[0]) + 1) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0111



Syntax `avgruw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are added. The result is incremented, shifted right logical one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = ((argument3.32[0] + argument2.32[0]) + 1) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.91 AVGRUWP Average Rounded Unsigned of Words Packed

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 0111

P	11	1	0111	registerW	10	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgruwp` *registerW* = *registerZ*, *registerY*

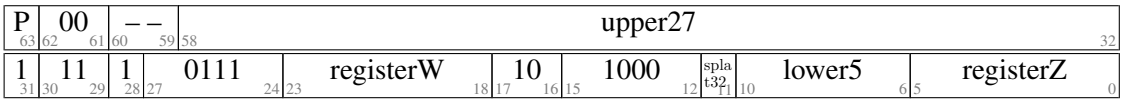
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are added. Each sum is incremented and shifted right logical one bit. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = ((_ZX_32(argument3.32[I]) + _ZX_32(argument2.32[I])) + 1)
         >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR.M (Section 6.12.275), exucode2 = 0111



Syntax `avgruwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are added. Each sum is incremented and shifted right logical one bit. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = ((_ZX_32(argument3.32[I]) + _ZX_32(argument2.32[I])) + 1)
        >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.92 AVGRW Average Rounded of Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0101

P	11	1	0101	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgrw` *registerW* = *registerZ*, *registerY*

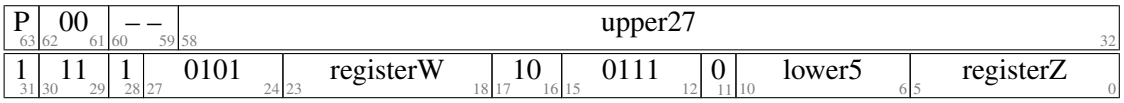
Description The *registerY* and the *registerZ* are added. The result is incremented, shifted right arithmetic one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ((_SX_32(argument3) + _SX_32(argument2)) + 1) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0101



Syntax `avgrw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are added. The result is incremented, shifted right arithmetic one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = ((_SX_32(argument3) + _SX_32(argument2)) + 1) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.93 AVGRWP Average Rounded of Words Packed

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 0101

P	11	1	0101	registerW	10	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgrwp` *registerW* = *registerZ*, *registerY*

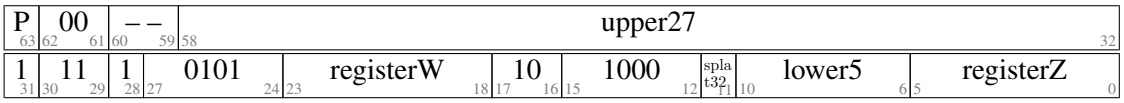
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are added. Each sum is incremented and shifted right arithmetic one bit. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = ((_SX_32(argument3.32[I]) + _SX_32(argument2.32[I])) + 1)
         >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR.M (Section 6.12.275), exucode2 = 0101



Syntax `avgrwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are added. Each sum is incremented and shifted right arithmetic one bit. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = ((_SX_32(argument3.32[I]) + _SX_32(argument2.32[I])) + 1)
        >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.94 AVGUBO Average Unsigned of Byte Octuple

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 0110

P	11	1	0110	registerW	10	1111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgubo` *registerW* = *registerZ*, *registerY*

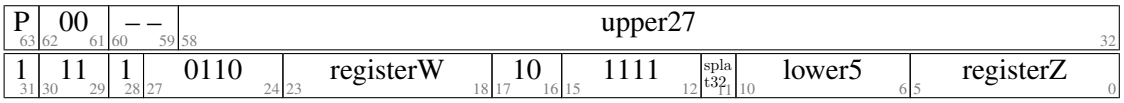
Description The *registerY* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *registerY* bytes and the *registerZ* bytes are added. Each sum is shifted right logical one bit. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = (_ZX_8(argument3.8[I]) + _ZX_8(argument2.8[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR.M (Section 6.12.290), exucode2 = 0110



Syntax `avgubo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *upper27_lower5* bytes and the *registerZ* bytes are added. Each sum is shifted right logical one bit. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = (_ZX_8(argument3.8[I]) + _ZX_8(argument2.8[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.95 AVGUHQ Average Unsigned of Half Words Quadruple

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 0110

P 31	11 30 29	1 28	0110 27 24	registerW 23 18	10 17 16	1001 15 12	registerY 11 6	registerZ 5 0

Syntax `avguhq` *registerW* = *registerZ*, *registerY*

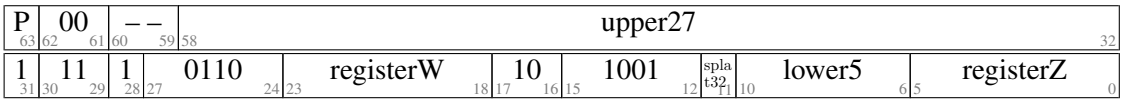
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are added. Each sum is shifted right logical one bit. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = (_ZX_16(argument3.16[I]) + _ZX_16(argument2.16[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR.M (Section 6.12.277), exucode2 = 0110



Syntax `avguhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are added. Each sum is shifted right logical one bit. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = (_ZX_16(argument3.16[I]) + _ZX_16(argument2.16[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.96 AVGUW — Average of Unsigned Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0110

P	11	1	0110	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avguw` *registerW* = *registerZ*, *registerY*

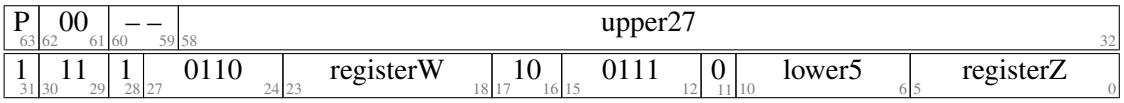
Description The *registerY* and the *registerZ* are added. The result is shifted right logical one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (argument3.32[0] + argument2.32[0]) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0110



Syntax `avguw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are added. The result is shifted right logical one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = (argument3.32[0] + argument2.32[0]) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.97 AVGUWP Average Unsigned of Words Packed

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 0110

P	11	1	0110	registerW	10	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avguwp` *registerW* = *registerZ*, *registerY*

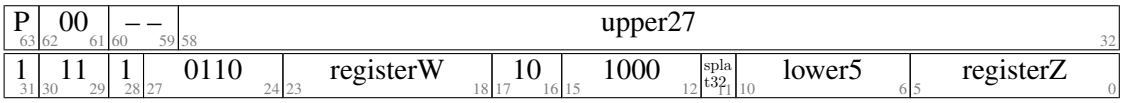
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are added. Each sum is shifted right logical one bit. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = (_ZX_32(argument3.32[I]) + _ZX_32(argument2.32[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR.M (Section 6.12.275), exucode2 = 0110



Syntax `avguwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are added. Each sum is shifted right logical one bit. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = (_ZX_32(argument3.32[I]) + _ZX_32(argument2.32[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.98 AVGW Average of Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0100

P	11	1	0100	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgw` *registerW* = *registerZ*, *registerY*

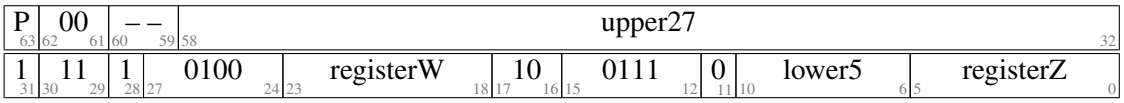
Description The *registerY* and the *registerZ* are added. The result is shifted right arithmetic one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (_SX_32(argument3) + _SX_32(argument2)) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0100



Syntax `avgw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are added. The result is shifted right arithmetic one bit and stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = (_SX_32(argument3) + _SX_32(argument2)) >> 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.99 AVGWP — Average of Words Packed

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 0100

P	11	1	0100	registerW	10	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `avgwp` *registerW* = *registerZ*, *registerY*

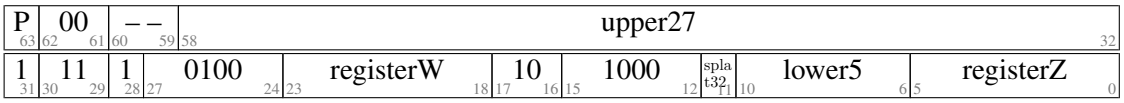
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are added. Each sum is shifted right arithmetic one bit. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = (_SX_32(argument3.32[I]) + _SX_32(argument2.32[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR.M (Section 6.12.275), exucode2 = 0100



Syntax `avgwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are added. Each sum is shifted right arithmetic one bit. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = (_SX_32(argument3.32[I]) + _SX_32(argument2.32[I])) >> 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.100 CBSD — Count Bit Set of Double Word

Encoding Format ALU_BWRD (Section 6.12.263), exucode2 = 0010

P	11	1	0010	registerW	10	0010	– – – – – –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `cbsd` *registerW* = *registerZ*

Description Bit set population count value of the *registerZ* is stored into the *registerW*.

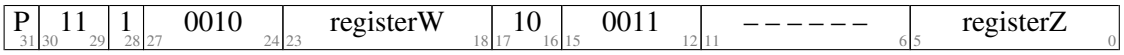
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(_CBS_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.101 CBSW — Count Bit Set of Word

Encoding Format ALU_BWRW (Section 6.12.265), exucode2 = 0010



Syntax `cbsw` *registerW* = *registerZ*

Description Bit set population count value of the *registerZ* is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(_CBS_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.102 CBSWP — Count Bit Sets of Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0010

P	11	1	0010	registerW	10	0100	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `cbswp` *registerW* = *registerZ*

Description Bit set population count values of the *registerZ* interpreted as word pair are stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(_CBS_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.103 CLRF — Clear Field

Encoding Format ALU_WRB (Section 6.12.215), wricode1 = 11

P	11	0	11	stopbit2	registerW	11	stopbit4	startbit	registerZ
31	30 29	28	27 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `clrf` *registerW* = *registerZ*, *stopbit2_stopbit4*, *startbit*

Description The start bit position and the stop bit position are extracted from the *startbit* and the *stopbit2_stopbit4* respectively. A bit mask is constructed with ones ranging from the start bit position to the stop bit position inclusive. If the start bit position is greater than the stop bit position, ones range from the start bit position to bit 63, then from bit 0 to stop bit position. The bitwise and of the *registerZ* and the bit mask complement is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new startbit = _ZX_6(startbit);
  new stopbit = _ZX_6(stopbit2_stopbit4);
stage RR:
  new bias = startbit <= stopbit;
  new mask = _ZX_64(((2 << stopbit) - ((2 - bias) << startbit)) + (bias - 1))
      ;
  new argument2 = GPR[registerZ];
  new result1 = ~mask & argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.104 CLSD — Count Leading Sign of Double Word

Encoding Format ALU_BWRD (Section 6.12.263), exucode2 = 0001

P	11	1	0001	registerW	10	0010	– – – – – –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `clsd` *registerW* = *registerZ*

Description The leading sign count value of the *registerZ* is stored into the *registerW*. When applied to zero or -1, the result is 63.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(_CLS_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.105 CLSW — Count Leading Sign of Word

Encoding Format ALU_BWRW (Section 6.12.265), exucode2 = 0001

P	11	1	0001	registerW	10	0011	– – – – – –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `clsw` *registerW* = *registerZ*

Description The leading sign count value of the *registerZ* is stored into the *registerW*. When applied to zero or -1, the result is 31.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(_CLS_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.106 CLSWP Count Leading Signs of Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0001

P	11	1	0001	registerW	10	0100	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `clswp` *registerW* = *registerZ*

Description The leading sign count values of the *registerZ* interpreted as word pair are stored into the *registerW*. When applied to zero or -1 at the word level, the result is 31.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(_CLS_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.107 CLZD — Count Leading Zero of Double Word

Encoding Format ALU_BWRD (Section 6.12.263), exucode2 = 0000

P	11	1	0000	registerW	10	0010	– – – – – –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `clzd` *registerW* = *registerZ*

Description The leading zero count value of the *registerZ* is stored into the *registerW*. When applied to zero, the result is 64.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(_CLZ_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.108 CLZW — Count Leading Zero of Word

Encoding Format ALU_BWRW (Section 6.12.265), exucode2 = 0000

P	11	1	0000	registerW	10	0011	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `clzw` *registerW* = *registerZ*

Description The leading zero count value of the *registerZ* is stored into the *registerW*. When applied to zero, the result is 32.

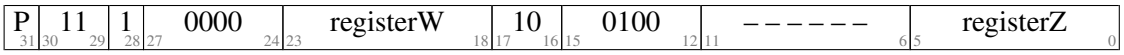
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(_CLZ_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.109 CLZWP — Count Leading Zeroes of Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0000



Syntax `clzwp` *registerW* = *registerZ*

Description The leading zero count values of the *registerZ* interpreted as a word pair are stored into the *registerW*. When applied to zero at the word level, the result is 32.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(_CLZ_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.110 CMOVEBO Conditional Move Byte Octuple

Encoding Format ALU_BOCMWRR (Section 6.12.286), exubit27 = 0

P	11	1	0	simdcond	registerW	10	1101	registerY	registerZ
31	30 29	28	27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `cmovebo`*simdcond registerZ? registerW = registerY*

Description The *registerW*, *registerZ* and the *registerY* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are tested using the condition *simdcond*. If true, the corresponding byte of the *registerY* is stored into the corresponding byte of the *registerW*.

Modifier(s)

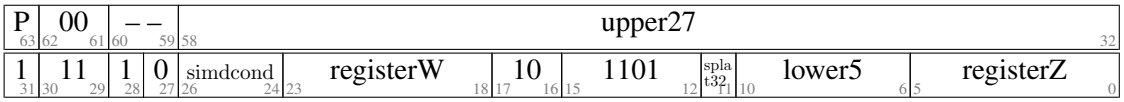
- simdcond (cf. simdcond in section 6.10.2 on page 1429)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(simdcond);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  if (simdcond_8(argument4, argument2.8[0])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[1])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[2])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[3])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[4])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[5])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[6])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[7])) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

Encoding Format ALU_BOCMWRR.M (Section 6.12.287), exubit27 = 0



Syntax `cmovebo`*simdcond registerZ*? *registerW* = *upper27_lower5splat32*

Description The *registerW*, *registerZ* and the *upper27_lower5* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are tested using the condition *simdcond*. If true, the corresponding byte of the *upper27_lower5* is stored into the corresponding byte of the *registerW*.

Modifier(s)

- simdcond (cf. simdcond in section 6.10.2 on page 1429)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_3(simdcond);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  if (simdcond_8(argument4, argument2.8[0])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[1])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[2])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[3])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[4])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[5])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_8(argument4, argument2.8[6])) {
    GPR[registerW] = argument3;
```


Kalray – all rights reserved

©2025 Kalray

```
  }
  if (simdcond_8(argument4, argument2.8[7])) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

6.6.111 CMOVED — Conditional Move Double Word

Encoding Format ALU_DCMWRI (Section 6.12.209), exucode4 = 10

P	11	0	scalarcond	registerW	10	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `cmoved`*scalarcond registerZ*? *registerW* = *signed10*

Description The *registerZ* is tested using the condition *scalarcond*. If true, the *signed10* is stored into the *registerW*. This instruction is used to replace conditional branches during if-conversion.

Modifier(s)

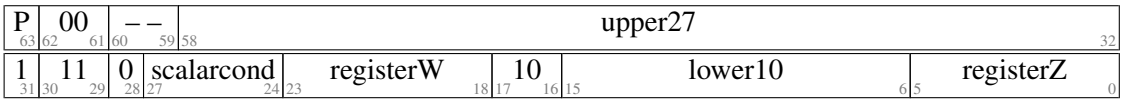
- scalarcond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(scalarcond);
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
stage E1:
  if (scalarcond(argument4, argument2)) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

Encoding Format ALU_DCMWRI.X (Section 6.12.210), exucode4 = 10



Syntax `cmoved`*scalarcond registerZ*? *registerW* = *upper27_lower10*

Description The *registerZ* is tested using the condition *scalarcond*. If true, the *upper27_lower10* is stored into the *registerW*. This instruction is used to replace conditional branches during if-conversion.

Modifier(s)

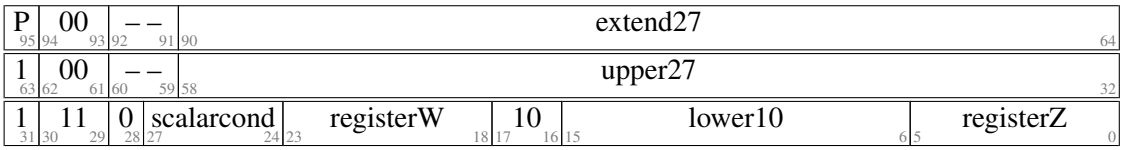
- scalarcond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_4(scalarcond);
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
stage E1:
  if (scalarcond(argument4, argument2)) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

Encoding Format ALU_DCMWRI.Y (Section 6.12.211), exucode4 = 10



Syntax `cmoved`*scalarcond registerZ*? *registerW* = *extend27_upper27_lower10*

Description The *registerZ* is tested using the condition *scalarcond*. If true, the *extend27_upper27_lower10* is stored into the *registerW*. This instruction is used to replace conditional branches during if-conversion.

Modifier(s)

- scalarcond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage ID:
  new argument4 = _ZX_4(scalarcond);
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
stage E1:
  if (scalarcond(argument4, argument2)) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

Encoding Format ALU_DCMWRR (Section 6.12.256), exucode4 = 10

P	11	1	scalarcond	registerW	10	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `cmoved`*scalarcond registerZ*? *registerW* = *registerY*

Description The *registerZ* is tested using the condition *scalarcond*. If true, the *registerY* is stored into the *registerW*. This instruction is used to replace conditional branches during if-conversion.

Modifier(s)

- scalarcond (cf. scalarcond in section 6.10.2 on page 1425)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(scalarcond);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  if (scalarcond(argument4, argument2)) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

6.6.112 CMOVEHQ Conditional Move Half Words Quadruple

Encoding Format ALU_CMWRRHQ (Section 6.12.261), exubit27 = 1

P	11	1	1	simdcond	registerW	10	0001	registerY	registerZ
31	30 29	28	27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `cmovehq`*simdcond registerZ*? *registerW* = *registerY*

Description The *registerW*, *registerZ* and the *registerY* are considered as half words packed into 64 bits. The *registerZ* half words are tested using the condition *simdcond*. If true, the corresponding half word of the *registerY* is stored into the corresponding half word of the *registerW*.

Modifier(s)

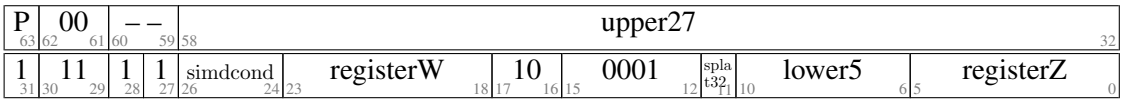
- simdcond (cf. simdcond in section 6.10.2 on page 1429)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(simdcond);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  if (simdcond_16(argument4, argument2.16[0])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_16(argument4, argument2.16[1])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_16(argument4, argument2.16[2])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_16(argument4, argument2.16[3])) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

Encoding Format ALU_CMWRRHQ.M (Section 6.12.262), exubit27 = 1



Syntax `cmovehq`*simdcond registerZ*? *registerW* = *upper27_lower5splat32*

Description The *registerW*, *registerZ* and the *upper27_lower5* are considered as half words packed into 64 bits. The *registerZ* half words are tested using the condition *simdcond*. If true, the corresponding half word of the *upper27_lower5* is stored into the corresponding half word of the *registerW*.

Modifier(s)

- simdcond (cf. simdcond in section 6.10.2 on page 1429)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

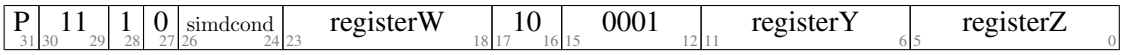
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_3(simdcond);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  if (simdcond_16(argument4, argument2.16[0])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_16(argument4, argument2.16[1])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_16(argument4, argument2.16[2])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_16(argument4, argument2.16[3])) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

6.6.113 CMOVEWP — Conditional Move Words Packed

Encoding Format ALU_WPCMWRR (Section 6.12.259), exubit27 = 0



Syntax `cmovewp`*simdcond registerZ*? *registerW* = *registerY*

Description The *registerW*, *registerZ* and the *registerY* are considered as words packed into 64 bits. The *registerZ* words are tested using the condition *simdcond*. If true, the corresponding word of the *registerY* is stored into the corresponding word of the *registerW*.

Modifier(s)

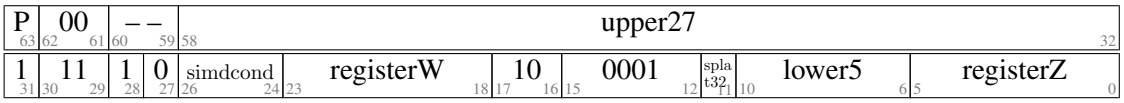
- simdcond (cf. simdcond in section 6.10.2 on page 1429)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(simdcond);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  if (simdcond_32(argument4, argument2.32[0])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_32(argument4, argument2.32[1])) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

Encoding Format ALU_WPCMWRR.M (Section 6.12.260), exubit27 = 0



Syntax `cmovewp`*simdcond registerZ*? *registerW* = *upper27_lower5splat32*

Description The *registerW*, *registerZ* and the *upper27_lower5* are considered as words packed into 64 bits. The *registerZ* words are tested using the condition *simdcond*. If true, the corresponding word of the *upper27_lower5* is stored into the corresponding word of the *registerW*.

Modifier(s)

- simdcond (cf. simdcond in section 6.10.2 on page 1429)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

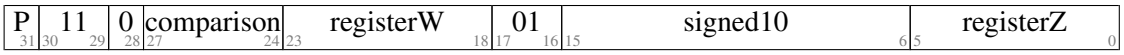
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_3(simdcond);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  if (simdcond_32(argument4, argument2.32[0])) {
    GPR[registerW] = argument3;
  }
  if (simdcond_32(argument4, argument2.32[1])) {
    GPR[registerW] = argument3;
  }
```


Kalray – all rights reserved

6.6.114 COMPD — Compare Double Words

Encoding Format ALU_CWRI (Section 6.12.203), steering = 11



Syntax `compd`*comparison registerW* = *registerZ*, *signed10*

Description The *registerZ* is compared to the *signed10* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

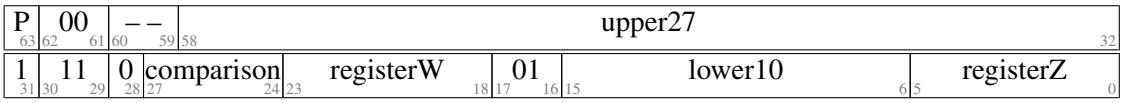
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_CWRI.X (Section 6.12.204), steering = 11



Syntax `compd`*comparison registerW = registerZ, upper27_lower10*

Description The *registerZ* is compared to the *upper27_lower10* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

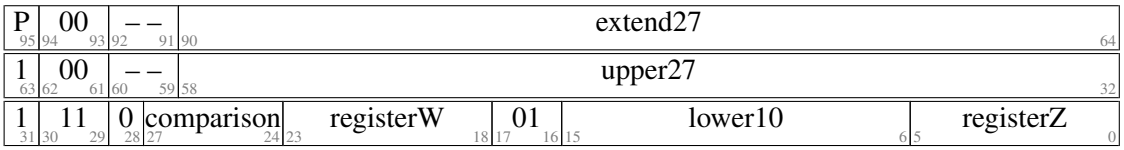
- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_CWRI.Y (Section 6.12.205), steering = 11



Syntax `compd`*comparison registerW = registerZ, extend27_upper27_lower10*

Description The *registerZ* is compared to the *extend27_upper27_lower10* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_CWRR (Section 6.12.241), steering = 11

P	11	1	comparison	registerW	01	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compd`*comparison registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.115 COMPNBO — Compare Negate Byte Octuple

Encoding Format ALU_BOCNWRR (Section 6.12.282), exucode4 = 10

P	11	1	comparison	registerW	10	1100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compnbo`*comparison registerW = registerZ, registerY*

Description The *registerZ* and the *registerY* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are compared to the *registerY* bytes using condition *comparison*. The resulting booleans are negated, packed and stored into the *registerW*.

Modifier(s)

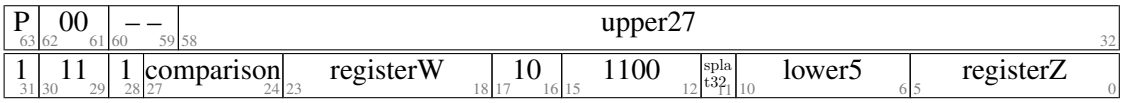
- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = -comparison_8(argument4, argument2.8[I], argument3.8[I])
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOCNWRR.M (Section 6.12.283), exucode4 = 10



Syntax `compnbo`*comparison registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are compared to the *upper27_lower5* bytes using condition *comparison*. The resulting booleans are negated, packed and stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = -comparison_8(argument4, argument2.8[I], argument3.8[I])
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.116 COMPND Compare Negate Double Words

Encoding Format ALU_CNWRR (Section 6.12.307), exucode5 = 1011

P	11	1	comparison	registerW	11	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compnd`*comparison registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *comparison*. The negated boolean result is stored into the *registerW*.

Modifier(s)

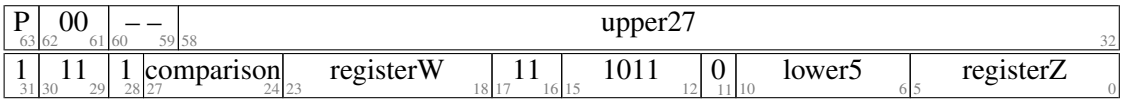
- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = -comparison_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_CNWRR.W (Section 6.12.308), exucode5 = 1011



Syntax `compnd`*comparison registerW = registerZ, upper27_lower5*

Description The *registerZ* is compared to the *upper27_lower5* using condition *comparison*. The negated boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = -comparison_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.117 COMPNHQ — Compare Negate Half Words Quadruple

Encoding Format ALU_HQCWRR (Section 6.12.252), exucode4 = 01

P	11	1	comparison	registerW	01	1111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compnhq`*comparison registerW = registerZ, registerY*

Description The *registerZ* and the *registerY* are considered as four half words packed into 64 bits. The *registerZ* half words are compared to the *registerY* half words using condition *comparison*. The resulting booleans are negated, packed and stored into the *registerW*.

Modifier(s)

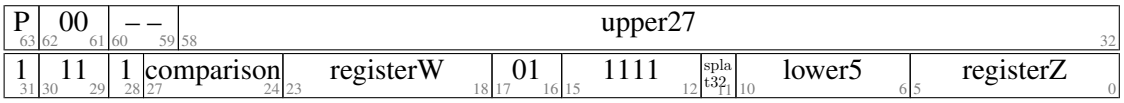
- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = -comparison_16(argument4, argument2.16[I], argument3.16[I
        ])
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQCWRR.M (Section 6.12.253), exucode4 = 01



Syntax `compnhq`*comparison registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as four half words packed into 64 bits. The *registerZ* half words are compared to the *upper27_lower5* half words using condition *comparison*. The resulting booleans are negated, packed and stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = -comparison_16(argument4, argument2.16[I], argument3.16[I
        ])
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.118 COMPNW — Compare Negate Words

Encoding Format ALU_CNWRR (Section 6.12.307), exucode5 = 1010

P	11	1	comparison	registerW	11	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compnw`*comparison registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *comparison*. The negated boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

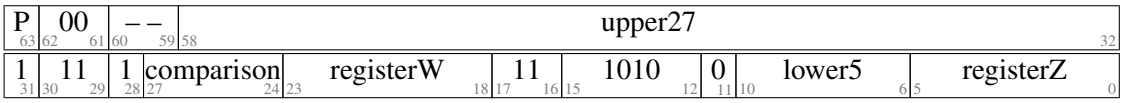
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(-comparison_32(argument4, argument2, argument3));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_CNWRR.W (Section 6.12.308), exucode5 = 1010



Syntax `compnw`*comparison registerW = registerZ, upper27_lower5*

Description The *registerZ* is compared to the *upper27_lower5* using condition *comparison*. The negated boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(-comparison_32(argument4, argument2, argument3));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.119 COMPNWP — Compare Negate Word Pair

Encoding Format ALU_WPCNWRR (Section 6.12.248), exucode4 = 01

P	11	1	comparison	registerW	01	1110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compnwp`*comparison registerW = registerZ, registerY*

Description The *registerZ* and the *registerY* are considered as two words packed into 64 bits. The *registerZ* words are compared to the *registerY* words using condition *comparison*. The resulting booleans are negated, packed and stored into the *registerW*.

Modifier(s)

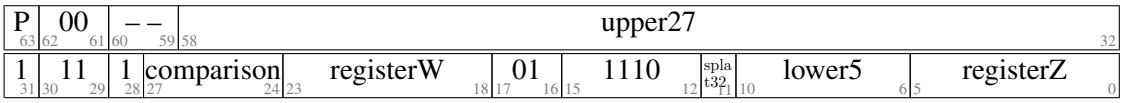
- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  result1.32[0] = -comparison_32(argument4, argument2.32[0], argument3.32[0])
      ;
  result1.32[1] = -comparison_32(argument4, argument2.32[1], argument3.32[1])
      ;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPCNWRR.M (Section 6.12.249), exucode4 = 01



Syntax `compnwp`*comparison registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as two words packed into 64 bits. The *registerZ* words are compared to the *upper27_lower5* words using condition *comparison*. The resulting booleans are negated, packed and stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  result1.32[0] = -comparison_32(argument4, argument2.32[0], argument3.32[0])
      ;
  result1.32[1] = -comparison_32(argument4, argument2.32[1], argument3.32[1])
      ;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.120 COMPUWD Compare Unsigned Word to Double Word

Encoding Format ALU_DCWRR (Section 6.12.244), exucode5 = 1101

P	11	1	comparison	registerW	01	1101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compuwd`*comparison registerW* = *registerZ*, *registerY*

Description The *registerZ* is zero-extended from 32 bits and compared to the *registerY* using condition *comparison*. The boolean result is stored into the *registerZ*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

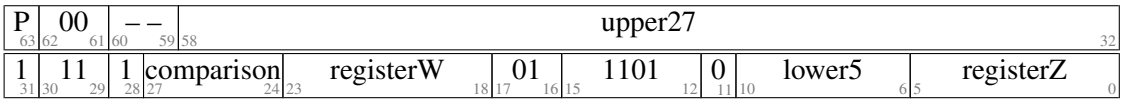
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, argument2.32[0], argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DCWRR.W (Section 6.12.245), exucode5 = 1101



Syntax `compuwd`*comparison registerW = registerZ, upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits and compared to the *upper27_lower5* using condition *comparison*. The boolean result is stored into the *registerZ*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

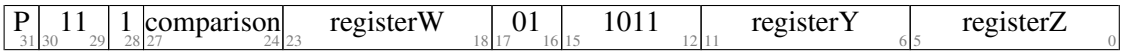
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, argument2.32[0], argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.121 COMPW — Compare Words

Encoding Format ALU_DCWRR (Section 6.12.244), exucode5 = 1011



Syntax `compw`*comparison registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

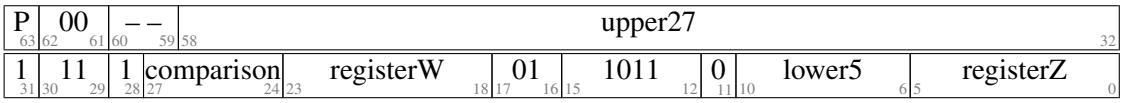
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = comparison_32(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DCWRR.W (Section 6.12.245), exucode5 = 1011



Syntax `compw`*comparison registerW = registerZ, upper27_lower5*

Description The *registerZ* is compared to the *upper27_lower5* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = comparison_32(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.122 COMPWD — Compare Word to Double Word

Encoding Format ALU_DCWRR (Section 6.12.244), exucode5 = 1100

P	11	1	comparison	registerW	01	1100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `compwd`*comparison registerW* = *registerZ*, *registerY*

Description The *registerZ* is sign-extended from 32 bits and compared to the *registerY* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

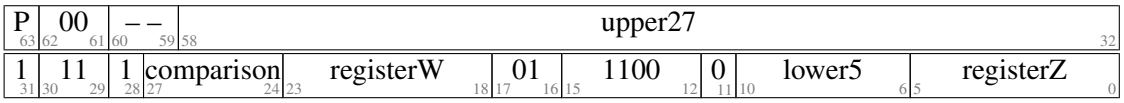
- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, _SX_32(argument2), argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DCWRR.W (Section 6.12.245), exucode5 = 1100



Syntax `compwd`*comparison registerW = registerZ, upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits and compared to the *upper27_lower5* using condition *comparison*. The boolean result is stored into the *registerW*.

Modifier(s)

- comparison (cf. comparison in section 6.10.2 on page 1417)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_4(comparison);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = comparison_64(argument4, _SX_32(argument2), argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.123 CTZD — Count Trailing Zero of Double Word

Encoding Format ALU_BWRD (Section 6.12.263), exucode2 = 0011

P	11	1	0011	registerW	10	0010	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `ctzd` *registerW* = *registerZ*

Description The trailing zero count value of the *registerZ* is stored into the *registerW*. When applied to zero, the result is 64.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(_CTZ_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.124 CTZW — Count Trailing Zero of Word

Encoding Format ALU_BWRW (Section 6.12.265), exucode2 = 0011

P	11	1	0011	registerW	10	0011	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `ctzw` *registerW* = *registerZ*

Description The trailing zero count value of the *registerZ* is stored into the *registerW*. When applied to zero, the result is 32.

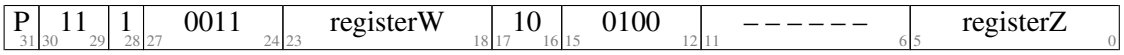
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(_CTZ_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.125 CTZWP — Count Trailing Zeroes of Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0011



Syntax `ctzwp` *registerW* = *registerZ*

Description The trailing zero count values of the *registerZ* interpreted as word pair are stored into the *registerW*. When applied to zero at the word level, the result is 32.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(_CTZ_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.126 EORD Bitwise Exclusive Or Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1100

P	11	1	1100	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `eord` *registerW* = *registerZ*, *registerY*

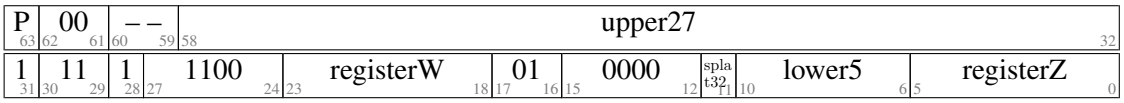
Description Bitwise exclusive or of the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1100



Syntax `eord` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise exclusive or of the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1100

P	11	0	1100	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `eord` *registerW* = *registerZ*, *signed10*

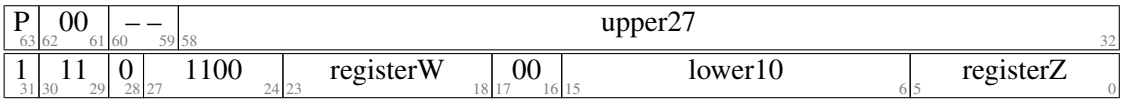
Description Bitwise exclusive or of the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1100



Syntax `eord` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise exclusive or of the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1100

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	11 30 29	0 28	1100 27 24	registerW 23 18	00 17 16	lower10 15 6	registerZ 5 0

Syntax `eord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description Bitwise exclusive or of the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

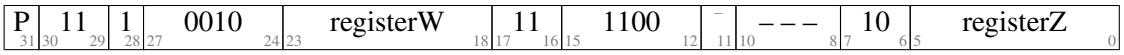
Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.127 EORRBOD Bitwise Exclusive OR Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 0010



Syntax `eorrbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The bitwise XOR of these eight zero-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = argument2.8[0] ^ argument2.8[1];
  new result1_y = argument2.8[2] ^ argument2.8[3];
  new result1_z = argument2.8[4] ^ argument2.8[5];
  new result1_t = argument2.8[6] ^ argument2.8[7];
  new result1_l = result1_x ^ result1_y;
  new result1_m = result1_z ^ result1_t;
  new result1 = result1_l ^ result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.128 EORRHQDBitwise Exclusive OR Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 0010

P	11	1	0010	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `eorrhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The bitwise XOR of these four zero-extended half words is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = argument2.16[0] ^ argument2.16[1];
  new result1_m = argument2.16[2] ^ argument2.16[3];
  new result1 = result1_l ^ result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.129 EORRWPD Bitwise Exclusive OR Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 0010

P	11	1	0010	registerW	11	1100	–	– – –	00	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `eorrwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The bitwise XOR of these two zero-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] ^ argument2.32[1];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.130 EORW — Bitwise Exclusive Or Between Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1100

P	11	1	1100	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `eorw` *registerW* = *registerZ*, *signed10*

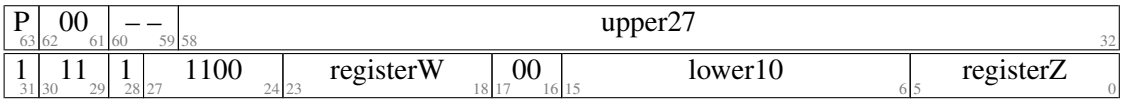
Description Bitwise exclusive or of the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1100



Syntax `eorw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise exclusive or of the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1100

P	11	1	1100	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `eorw` *registerW* = *registerZ*, *registerY*

Description Bitwise exclusive or of the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

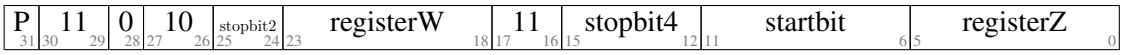
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.131 EXTFS Extract Bit Field and Sign Extend to Double Word

Encoding Format ALU_WRB (Section 6.12.215), wricode1 = 10



Syntax `extfs` *registerW* = *registerZ*, *stopbit2_stopbit4*, *startbit*

Description The start bit position and the stop bit position are extracted from the *startbit* and the *stopbit2_stopbit4* respectively. The bit-field of the *registerZ* extending from the start bit position to the stop bit position is extracted, sign-extended from the stop bit position, shifted right by the start bit position and stored into the *registerW*. Behavior is undefined if the start bit position is greater than the stop bit position.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new startbit = _ZX_6(startbit);
  new stopbit = _ZX_6(stopbit2_stopbit4);
stage RR:
  new bias = startbit <= stopbit;
  new mask = _ZX_64(((2 << stopbit) - ((2 - bias) << startbit)) + (bias - 1))
      ;
  new argument2 = GPR[registerZ];
  new masked1 = _ZX_64(argument2 & mask);
  new masked2 = _SX_64(argument2 | ~mask);
  new negative = argument2 & (mask & ~(mask >> 1));
  new masked = negative == 0 ? masked1 : masked2;
  new result1 = masked >> startbit;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.132 EXTFZ Extract Bit Field and Zero Extend to Double Word

Encoding Format ALU_WRB (Section 6.12.215), wricode1 = 01

P	11	0	01	stopbit2	registerW	11	stopbit4	startbit	registerZ
31	30 29	28	27 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `extfz` *registerW* = *registerZ*, *stopbit2_stopbit4*, *startbit*

Description The start bit position and the stop bit position are extracted from the *startbit* and the *stopbit2_stopbit4* respectively. The bit-field of the *registerZ* extending from the start bit position to the stop bit position is extracted, zero-extended after the stop bit position, shifted right by the start bit position and stored into the *registerW*. Behavior is undefined if the start bit position is greater than the stop bit position.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new startbit = _ZX_6(startbit);
  new stopbit = _ZX_6(stopbit2_stopbit4);
stage RR:
  new bias = startbit <= stopbit;
  new mask = _ZX_64(((2 << stopbit) - ((2 - bias) << startbit)) + (bias - 1))
      ;
  new argument2 = GPR[registerZ];
  new masked = argument2 & mask;
  new result1 = masked >> startbit;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.133 INSF — Insert Bit Field into Word

Encoding Format ALU_WRB (Section 6.12.215), wricode1 = 00

P	11	0	00	stopbit2	registerW	11	stopbit4	startbit	registerZ
31	30 29	28	27 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `insf` *registerW* = *registerZ*, *stopbit2_stopbit4*, *startbit*

Description The start bit position and the stop bit position are extracted from the *startbit* and the *stopbit2_stopbit4* respectively. A bit-mask is computed with ones set between the start bit position and the stop bit position inclusive. The *registerZ* is shifted left by the start bit position and inserted into the *registerW* according to the bit-mask. Behavior is undefined if the start bit position is greater than the stop bit position.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new startbit = _ZX_6(startbit);
  new stopbit = _ZX_6(stopbit2_stopbit4);
stage RR:
  new bias = startbit <= stopbit;
  new mask = _ZX_64(((2 << stopbit) - ((2 - bias) << startbit)) + (bias - 1))
      ;
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new result1 = ((argument2 << startbit) & mask) | (argument1 & ~mask);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.134 IORD Bitwise Inclusive Or Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1010

P	11	1	1010	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `iord` *registerW* = *registerZ*, *registerY*

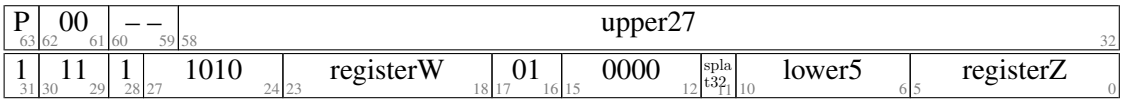
Description Bitwise inclusive or of the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1010



Syntax `iord` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise inclusive or of the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1010

P	11	0	1010	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `iord` *registerW* = *registerZ*, *signed10*

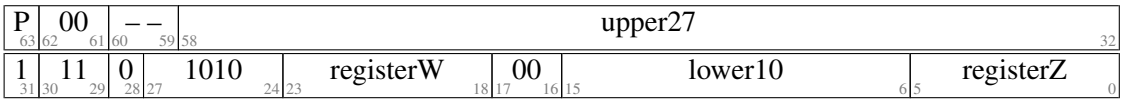
Description Bitwise inclusive or of the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1010



Syntax `iord` *registerW* = *registerZ*, *upper27_lower10*

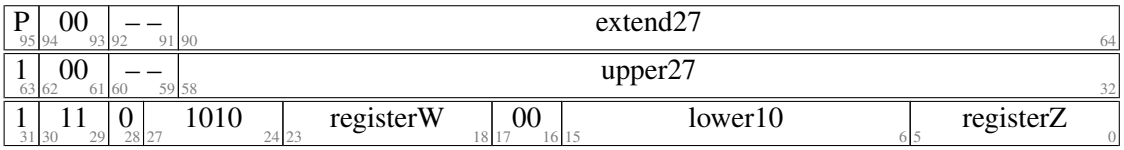
Description Bitwise inclusive or of the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1010



Syntax `iord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description Bitwise inclusive or of the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.135 IORND Bitwise Inclusive Or with Not Operand Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1111

P	11	1	1111	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `iornd` *registerW* = *registerZ*, *registerY*

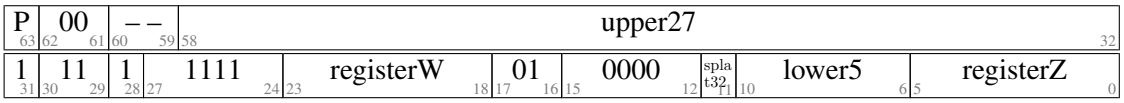
Description Bitwise inclusive or of not the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1111



Syntax `iornd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise inclusive or of not the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

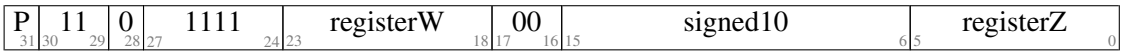
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1111



Syntax `iornd` *registerW* = *registerZ*, *signed10*

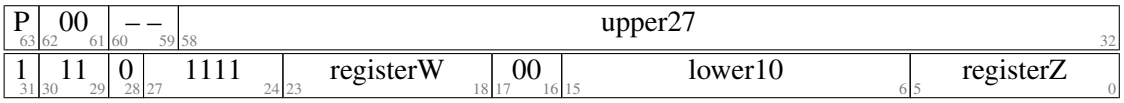
Description Bitwise inclusive or of not the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1111



Syntax `iornd` *registerW* = *registerZ*, *upper27_lower10*

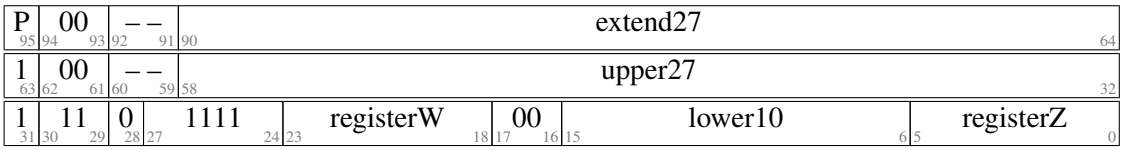
Description Bitwise inclusive or of not the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1111



Syntax `iornd` *registerW = registerZ, extend27_upper27_lower10*

Description Bitwise inclusive or of not the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.136 IORNW Bitwise Inclusive Or with Not Operand Between Double Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1111

P	11	1	1111	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `iornw` *registerW* = *registerZ*, *signed10*

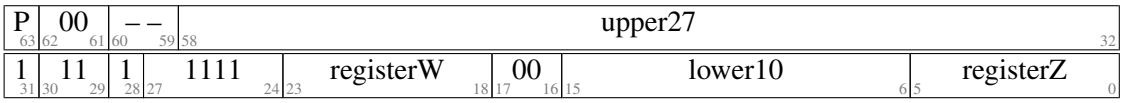
Description Bitwise inclusive or of not the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1111



Syntax `iornw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise inclusive or of not the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1111

P	11	1	1111	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `iornw` *registerW* = *registerZ*, *registerY*

Description Bitwise inclusive or of not the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~argument2 | argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.137 IORRBOD Bitwise OR Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 0001

P	11	1	0001	registerW	11	1100	–	– – –	10	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `iorrbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The bitwise OR of these eight zero-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = argument2.8[0] | argument2.8[1];
  new result1_y = argument2.8[2] | argument2.8[3];
  new result1_z = argument2.8[4] | argument2.8[5];
  new result1_t = argument2.8[6] | argument2.8[7];
  new result1_l = result1_x | result1_y;
  new result1_m = result1_z | result1_t;
  new result1 = result1_l | result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.138 IORRHQD Bitwise OR Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 0001

P	11	1	0001	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `iorrhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The bitwise OR of these four zero-extended half words is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = argument2.16[0] | argument2.16[1];
  new result1_m = argument2.16[2] | argument2.16[3];
  new result1 = result1_l | result1_m;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.139 IORRWPD Bitwise OR Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 0001

P	11	1	0001	registerW	11	1100	–	– – –	00	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `iorrwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The bitwise OR of these two zero-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] | argument2.32[1];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.140 IORW — Bitwise Inclusive Or Between Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1010

P	11	1	1010	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `iorw` *registerW* = *registerZ*, *signed10*

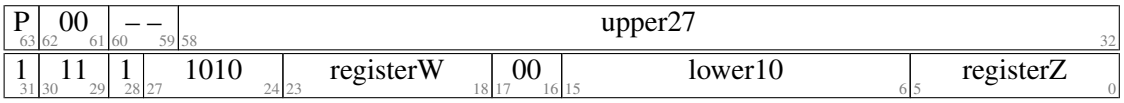
Description Bitwise inclusive or of the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1010



Syntax `iorw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise inclusive or of the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1010

P	11	1	1010	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `iorw` *registerW* = *registerZ*, *registerY*

Description Bitwise inclusive or of the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2 | argument3;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.141 LANDD — Logical And of Double Words

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 0000

P	11	1	0000	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `landd` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are converted to booleans, which are AND-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument3) && _ZX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.142 LANDW Logical And of Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0000

P	11	1	0000	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `landw` *registerW* = *registerZ*, *registerY*

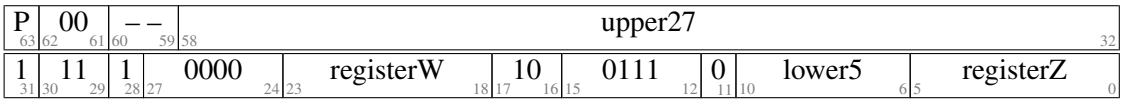
Description The *registerY* and the *registerZ* are converted to booleans, which are AND-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3.32[0] && argument2.32[0];
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0000



Syntax `landw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are converted to booleans, which are AND-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE.X (Section 6.14.6)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3.32[0] && argument2.32[0];
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.143 LIORD Logical Or of Double Words

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 0010

P	11	1	0010	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `liord` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are converted to booleans, which are OR-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument3) || _ZX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.144 LIORW — Logical Or of Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0010

P	11	1	0010	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `liorw` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are converted to booleans, which are OR-ed. The result is stored into the *registerW*.

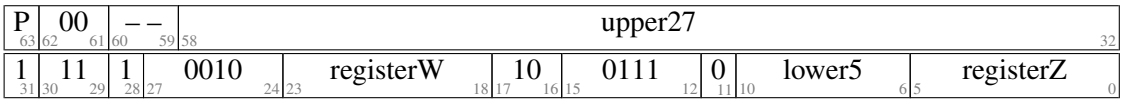
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3.32[0] || argument2.32[0];
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0010



Syntax `liorw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are converted to booleans, which are OR-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE.X (Section 6.14.6)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3.32[0] || argument2.32[0];
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.145 LNANDD Logical Not And of Double Words

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 0001

P	11	1	0001	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `lnandd` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are converted to booleans, which are NOT AND-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = !(_ZX_64(argument3) && _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.146 LNANDW Logical Not And of Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0001

P	11	1	0001	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `lnandw` *registerW* = *registerZ*, *registerY*

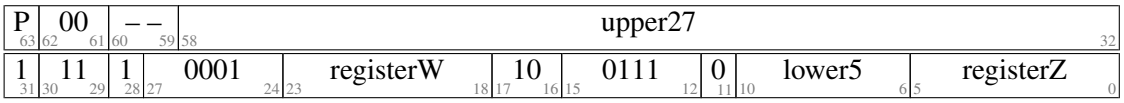
Description The *registerY* and the *registerZ* are converted to booleans, which are NOT AND-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = !(argument3.32[0] && argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0001



Syntax `lnandw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are converted to booleans, which are NOT AND-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE.X (Section 6.14.6)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = !(argument3.32[0] && argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.147 LNIORD — Logical Not Or of Double Words

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 0011

P	11	1	0011	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `lniord` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are converted to booleans, which are not OR-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = !(_ZX_64(argument3) || _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.148 LNIORW — Logical Not Or of Words

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 0011

P	11	1	0011	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `lniorw` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are converted to booleans, which are not OR-ed. The result is stored into the *registerW*.

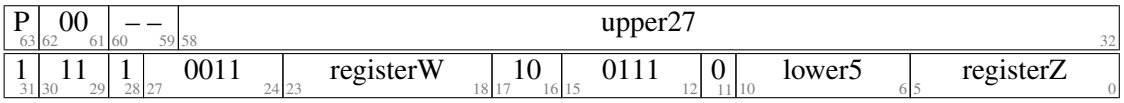
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = !(argument3.32[0] || argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WSWRR.W (Section 6.12.273), exucode2 = 0011



Syntax `lniorw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* and the *registerZ* are converted to booleans, which are not OR-ed. The result is stored into the *registerW*.

Execution Reservation table ALU_LITE.X (Section 6.14.6)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = !(argument3.32[0] || argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.149 MAKE — Make Word from Immediate

Encoding Format ALU_MAKE (Section 6.12.197), exucode2 = 0000

P	11	0	0000	registerW	00	signed16
31	30 29	28	27 24	23 18	17 16	15 0

Syntax `make` *registerW* = *signed16*

Description The *signed16* is stored into the *registerW*.

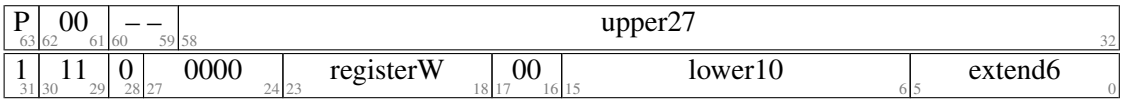
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = _SX_16(signed16);
  new result1 = _SX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_MAKE.X (Section 6.12.198), exucode2 = 0000



Syntax `make` *registerW* = *extend6_upper27_lower10*

Description The *extend6_upper27_lower10* is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument2 = _SX_43(extend6_upper27_lower10);
  new result1 = _SX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_MAKE.Y (Section 6.12.199), exucode2 = 0000

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	11 30 29	0 28	0000 27 24	registerW 23 18	00 17 16	lower10 15 6	– – – – – – 5 0

Syntax `make` *registerW = extend27_upper27_lower10*

Description The *extend27_upper27_lower10* is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument2 = _WX_64(extend27_upper27_lower10);
  new result1 = _SX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.150 MAXBO — Maximum of Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0101

P	11	1	0101	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxbo` *registerW* = *registerZ*, *registerY*

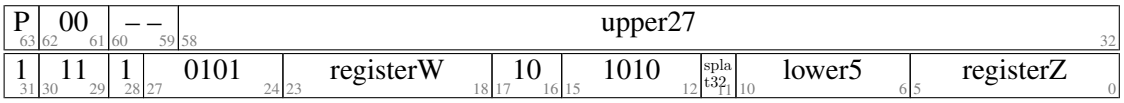
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The maximum of the *registerY* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MAX(_SX_8(argument3.8[I]), _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0101



Syntax `maxbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The maximum of the *upper27_lower5* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MAX(_SX_8(argument3.8[I]), _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.151 MAXD — Maximum of Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0101

P	11	1	0101	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxd` *registerW* = *registerZ*, *registerY*

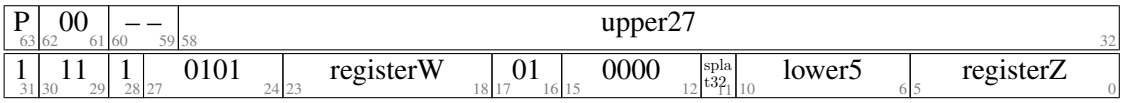
Description The maximum of the *registerY* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0101



Syntax `maxd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The maximum of the *upper27_lower5* and the *registerZ* is computed. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 0101

P	11	0	0101	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `maxd` *registerW* = *registerZ*, *signed10*

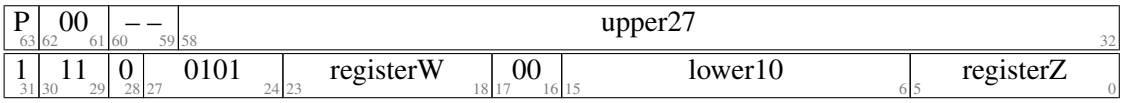
Description The maximum of the *signed10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 0101



Syntax `maxd` *registerW* = *registerZ*, *upper27_lower10*

Description The maximum of the *upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 0101

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	11 30 29	0 28	0101 27 24	registerW 23 18	00 17 16	lower10 15 6	registerZ 5 0

Syntax `maxd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The maximum of the *extend27_upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.152 MAXHQ — Maximum of Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0101

P	11	1	0101	registerW	01	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxhq` *registerW* = *registerZ*, *registerY*

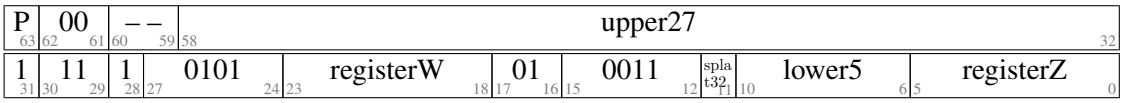
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The maximum of the *registerY* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MAX(_SX_16(argument3.16[I]), _SX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0101



Syntax `maxhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The maximum of the *upper27_lower5* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

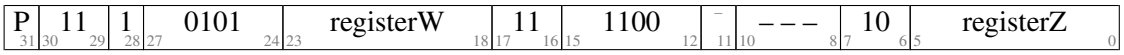
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MAX(_SX_16(argument3.16[I]), _SX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.153 MAXRBOD Maximum Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 0101



Syntax `maxrbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The maximum of these eight sign-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = _MAX(_SX_8(argument2.8[0]), _SX_8(argument2.8[1]));
  new result1_y = _MAX(_SX_8(argument2.8[2]), _SX_8(argument2.8[3]));
  new result1_z = _MAX(_SX_8(argument2.8[4]), _SX_8(argument2.8[5]));
  new result1_t = _MAX(_SX_8(argument2.8[6]), _SX_8(argument2.8[7]));
  new result1_l = _MAX(result1_x, result1_y);
  new result1_m = _MAX(result1_z, result1_t);
  new result1 = _MAX(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.154 MAXRHQD Maximum Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 0101

P	11	1	0101	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `maxrhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The maximum of these four sign-extended half words is computed and stored into the *registerW*.

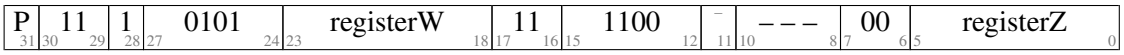
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = _MAX(_SX_16(argument2.16[0]), _SX_16(argument2.16[1]));
  new result1_m = _MAX(_SX_16(argument2.16[2]), _SX_16(argument2.16[3]));
  new result1 = _MAX(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.155 MAXRWPD Maximum Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 0101



Syntax `maxrwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The maximum of these two sign-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_32(argument2.32[0]), _SX_32(argument2.32[1]));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.156 MAXUBO Maximum of Unsigned Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0111

P	11	1	0111	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxubo` *registerW* = *registerZ*, *registerY*

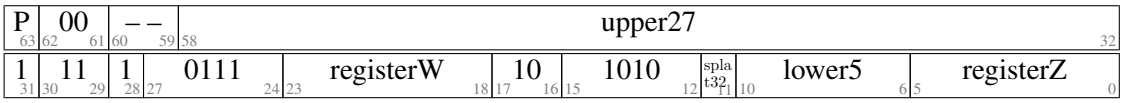
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The unsigned maximum of the *registerY* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MAX(_ZX_8(argument3.8[I]), _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0111



Syntax `maxubo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The unsigned maximum of the *upper27_lower5* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MAX(_ZX_8(argument3.8[I]), _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.157 MAXUD — Maximum of Unsigned Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0111

P	11	1	0111	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxud` *registerW* = *registerZ*, *registerY*

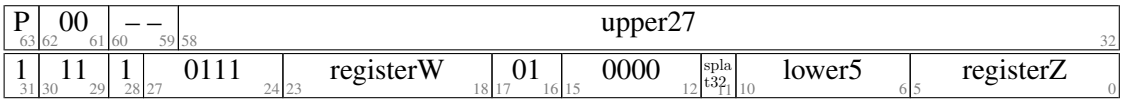
Description The unsigned maximum of the *registerY* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0111



Syntax `maxud` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The unsigned maximum of the *upper27_lower5* and the *registerZ* is computed. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 0111

P	11	0	0111	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `maxud` *registerW* = *registerZ*, *signed10*

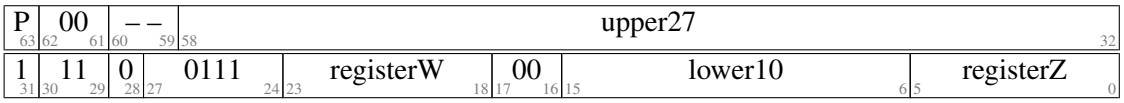
Description The unsigned maximum of the *signed10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 0111



Syntax `maxud` *registerW* = *registerZ*, *upper27_lower10*

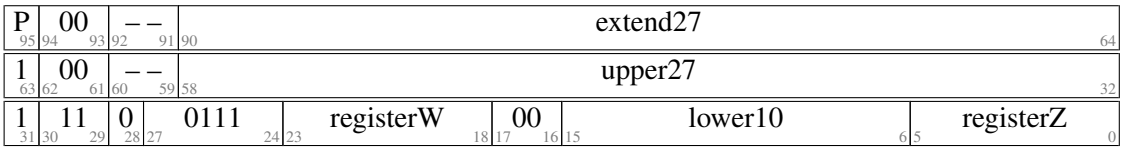
Description The unsigned maximum of the *upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 0111



Syntax `maxud` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The unsigned maximum of the *extend27_upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

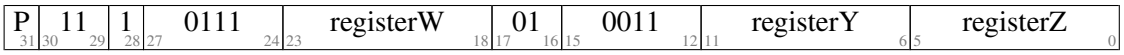
Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.158 MAXUHQ Maximum of Unsigned Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0111



Syntax `maxuhq` *registerW* = *registerZ*, *registerY*

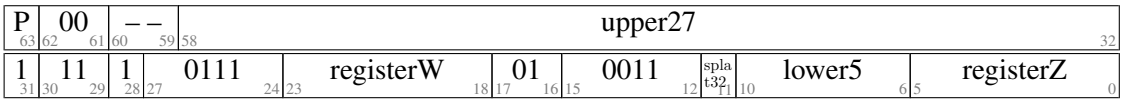
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The unsigned maximum of the *registerY* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MAX(_ZX_16(argument3.16[I]), _ZX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0111



Syntax `maxuhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The unsigned maximum of the *upper27_lower5* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

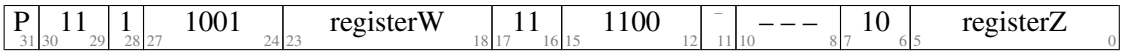
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MAX(_ZX_16(argument3.16[I]), _ZX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.159 MAXURBOD Maximum Unsigned Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 1001



Syntax `maxurbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The maximum of these eight zero-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = _MAX(argument2.8[0], argument2.8[1]);
  new result1_y = _MAX(argument2.8[2], argument2.8[3]);
  new result1_z = _MAX(argument2.8[4], argument2.8[5]);
  new result1_t = _MAX(argument2.8[6], argument2.8[7]);
  new result1_l = _MAX(result1_x, result1_y);
  new result1_m = _MAX(result1_z, result1_t);
  new result1 = _MAX(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.160 MAXURHQD Maximum Unsigned Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 1001

Syntax `maxurhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The maximum of these four zero-extended half words is computed and stored into the *registerW*.

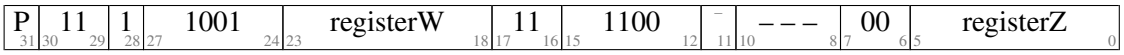
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = _MAX(_ZX_16(argument2.16[0]), _ZX_16(argument2.16[1]));
  new result1_m = _MAX(_ZX_16(argument2.16[2]), _ZX_16(argument2.16[3]));
  new result1 = _MAX(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.161 MAXURWPD Maximum Unsigned Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 1001



Syntax `maxurwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The maximum of these two zero-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _MAX(argument2.32[0], argument2.32[1]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.162 MAXUW — Maximum of Unsigned Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 0111

P	11	1	0111	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `maxuw` *registerW* = *registerZ*, *signed10*

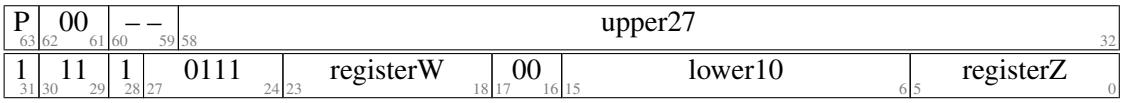
Description The unsigned maximum of the *signed10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(argument3.32[0], argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 0111



Syntax `maxuw` *registerW* = *registerZ*, *upper27_lower10*

Description The unsigned maximum of the *upper27_lower10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(argument3.32[0], argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0111

P	11	1	0111	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxuw` *registerW* = *registerZ*, *registerY*

Description The unsigned maximum of the *registerY* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MAX(argument3.32[0], argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.163 MAXUWP — Maximum of Unsigned Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0111

P	11	1	0111	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxuwp` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The unsigned maximum of the *registerY* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MAX(_ZX_32(argument3.32[I]), _ZX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0111

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0111	registerW	01	0010	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	f1	10 6	5 0

Syntax `maxuwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The unsigned maximum of the *upper27_lower5* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MAX(_ZX_32(argument3.32[I]), _ZX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.164 MAXW — Maximum of Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 0101

P	11	1	0101	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `maxw` *registerW* = *registerZ*, *signed10*

Description The maximum of the *signed10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

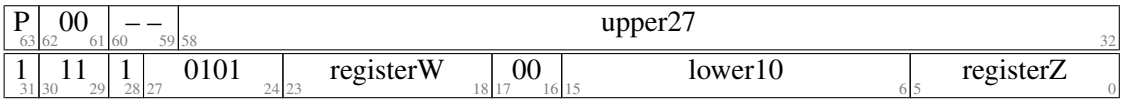
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_32(argument3), _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 0101



Syntax `maxw` *registerW* = *registerZ*, *upper27_lower10*

Description The maximum of the *upper27_lower10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_32(argument3), _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0101

P	11	1	0101	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxw` *registerW* = *registerZ*, *registerY*

Description The maximum of the *registerY* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MAX(_SX_32(argument3), _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.165 MAXWP Maximum of Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0101

P	11	1	0101	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maxwp` *registerW* = *registerZ*, *registerY*

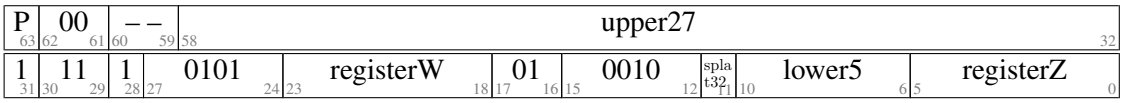
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The maximum of the *registerY* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MAX(_SX_32(argument3.32[I]), _SX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0101



Syntax `maxwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The maximum of the *upper27_lower5* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MAX(_SX_32(argument3.32[I]), _SX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.166 MINBO — Minimum of Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0100

P	11	1	0100	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minbo` *registerW* = *registerZ*, *registerY*

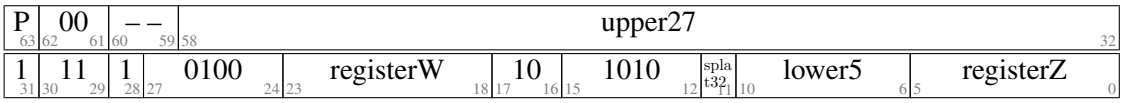
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The minimum of the *registerY* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MIN(_SX_8(argument3.8[I]), _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0100



Syntax `minbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The minimum of the *upper27_lower5* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MIN(_SX_8(argument3.8[I]), _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.167 MIND — Minimum of Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0100

P	11	1	0100	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `mind` *registerW* = *registerZ*, *registerY*

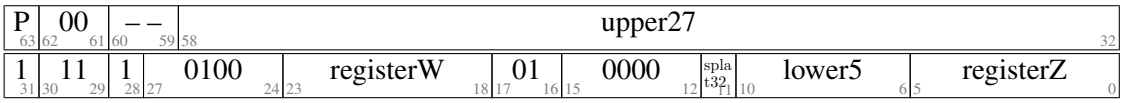
Description The minimum of the *registerY* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0100



Syntax `mind` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The minimum of the *upper27_lower5* and the *registerZ* is computed. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 0100

P	11	0	0100	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `mind` *registerW* = *registerZ*, *signed10*

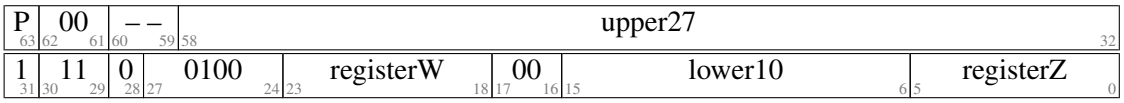
Description The minimum of the *signed10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 0100



Syntax `mind` *registerW* = *registerZ*, *upper27_lower10*

Description The minimum of the *upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 0100

P [95]	00 [94:93]	– – [92:91]	extend27 [90:64]				
1 [63]	00 [62:61]	– – [60:59]	upper27 [58:32]				
1 [31]	11 [30:29]	0 [28]	0100 [27:24]	registerW [23:18]	00 [17:16]	lower10 [15:6]	registerZ [5:0]

Syntax `mind` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The minimum of the *extend27_upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_64(argument3), _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.168 MINHQ — Minimum of Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0100

P	11	1	0100	registerW	01	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minhq` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The minimum of the *registerY* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MIN(_SX_16(argument3.16[I]), _SX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0100

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0100	registerW	01	0011	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `minhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The minimum of the *upper27_lower5* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

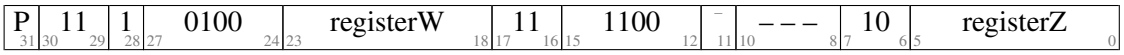
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MIN(_SX_16(argument3.16[I]), _SX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.169 MINRBOD Minimum Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 0100



Syntax `minrbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The minimum of these eight sign-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = _MIN(_SX_8(argument2.8[0]), _SX_8(argument2.8[1]));
  new result1_y = _MIN(_SX_8(argument2.8[2]), _SX_8(argument2.8[3]));
  new result1_z = _MIN(_SX_8(argument2.8[4]), _SX_8(argument2.8[5]));
  new result1_t = _MIN(_SX_8(argument2.8[6]), _SX_8(argument2.8[7]));
  new result1_l = _MIN(result1_x, result1_y);
  new result1_m = _MIN(result1_z, result1_t);
  new result1 = _MIN(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.170 MINRHQD Minimum Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 0100

P	11	1	0100	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `minrhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The minimum of these four sign-extended half words is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = _MIN(_SX_16(argument2.16[0]), _SX_16(argument2.16[1]));
  new result1_m = _MIN(_SX_16(argument2.16[2]), _SX_16(argument2.16[3]));
  new result1 = _MIN(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.171 MINRWPD Minimum Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 0100

P	11	1	0100	registerW	11	1100	–	– – –	00	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `minrwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The minimum of these two sign-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_32(argument2.32[0]), _SX_32(argument2.32[1]));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.172 MINUBO Minimum of Unsigned Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0110

P	11	1	0110	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minubo` *registerW* = *registerZ*, *registerY*

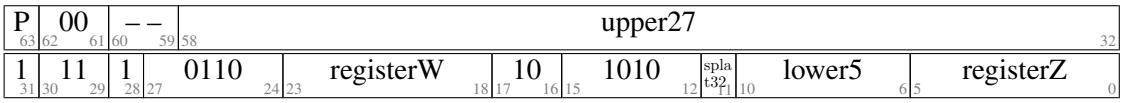
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The unsigned minimum of the *registerY* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MIN(_ZX_8(argument3.8[I]), _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0110



Syntax `minubo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The unsigned minimum of the *upper27_lower5* bytes and the *registerZ* bytes are computed. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _MIN(_ZX_8(argument3.8[I]), _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.173 MINUD — Minimum of Unsigned Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0110

P	11	1	0110	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minud` *registerW* = *registerZ*, *registerY*

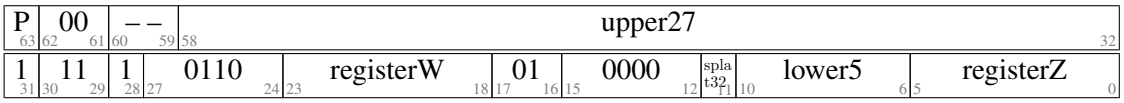
Description The unsigned minimum of the *registerY* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0110



Syntax `minud` *registerW = registerZ, upper27_lower5splat32*

Description The unsigned minimum of the *upper27_lower5* and the *registerZ* is computed. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

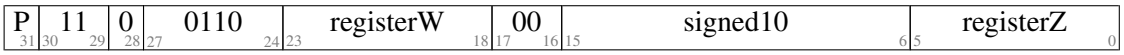
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 0110



Syntax `minud` *registerW* = *registerZ*, *signed10*

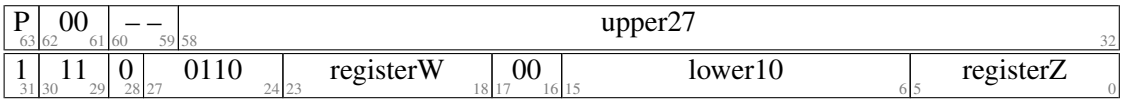
Description The unsigned minimum of the *signed10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 0110



Syntax `minud` *registerW* = *registerZ*, *upper27_lower10*

Description The unsigned minimum of the *upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 0110

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	11 30 29	0 28	0110 27 24	registerW 23 18	00 17 16	lower10 15 6	registerZ 5 0

Syntax `minud` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The unsigned minimum of the *extend27_upper27_lower10* and the *registerZ* is computed. The result is stored into the *registerW*.

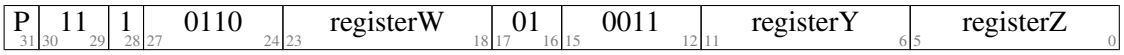
Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.174 MINUHQ — Minimum of Unsigned Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0110



Syntax `minuhq` *registerW* = *registerZ*, *registerY*

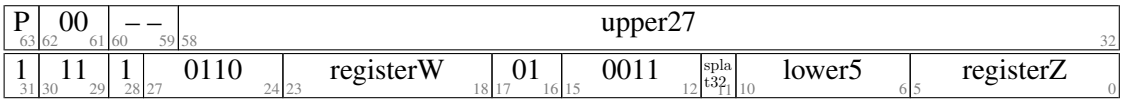
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The unsigned minimum of the *registerY* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MIN(_ZX_16(argument3.16[I]), _ZX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0110



Syntax `minuhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The unsigned minimum of the *upper27_lower5* half words and the *registerZ* half words are computed. The four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _MIN(_ZX_16(argument3.16[I]), _ZX_16(argument2.16[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.175 MINURBOD Minimum Unsigned Reduce Byte Octuple to Double Word

Encoding Format ALU_BORWRR (Section 6.12.311), exucode2 = 1000

P	11	1	1000	registerW	11	1100	–	– – –	10	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `minurbod` *registerW* = *registerZ*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The minimum of these eight zero-extended bytes is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_x = _MIN(argument2.8[0], argument2.8[1]);
  new result1_y = _MIN(argument2.8[2], argument2.8[3]);
  new result1_z = _MIN(argument2.8[4], argument2.8[5]);
  new result1_t = _MIN(argument2.8[6], argument2.8[7]);
  new result1_l = _MIN(result1_x, result1_y);
  new result1_m = _MIN(result1_z, result1_t);
  new result1 = _MIN(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.176 MINURHQDMinimum Unsigned Reduce Half Word Quadruple to Double Word

Encoding Format ALU_HQRWRR (Section 6.12.310), exucode2 = 1000

P	11	1	1000	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `minurhqd` *registerW* = *registerZ*

Description The *registerZ* is considered as four half words packed into 64 bits. The minimum of these four zero-extended half words is computed and stored into the *registerW*.

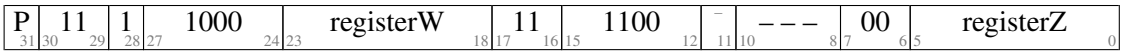
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1_l = _MIN(_ZX_16(argument2.16[0]), _ZX_16(argument2.16[1]));
  new result1_m = _MIN(_ZX_16(argument2.16[2]), _ZX_16(argument2.16[3]));
  new result1 = _MIN(result1_l, result1_m);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.177 MINURWPD Minimum Unsigned Reduce Word Pair to Double Word

Encoding Format ALU_WPRWRR (Section 6.12.309), exucode2 = 1000



Syntax `minurwpd` *registerW* = *registerZ*

Description The *registerZ* is considered as two words packed into 64 bits. The minimum of these two zero-extended words is computed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _MIN(argument2.32[0], argument2.32[1]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.178 MINUW — Minimum of Unsigned Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 0110

P	11	1	0110	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `minuw` *registerW* = *registerZ*, *signed10*

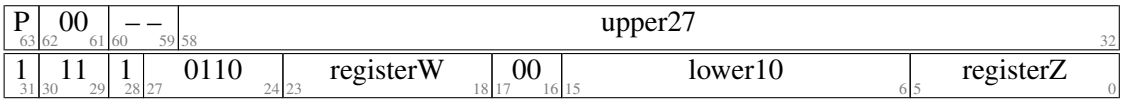
Description The unsigned minimum of the *signed10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(argument3.32[0], argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 0110



Syntax `minuw` *registerW* = *registerZ*, *upper27_lower10*

Description The unsigned minimum of the *upper27_lower10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(argument3.32[0], argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0110

P	11	1	0110	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minuw` *registerW* = *registerZ*, *registerY*

Description The unsigned minimum of the *registerY* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MIN(argument3.32[0], argument2.32[0]);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.179 MINUWP — Minimum of Unsigned Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0110

P	11	1	0110	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minuwp` *registerW* = *registerZ*, *registerY*

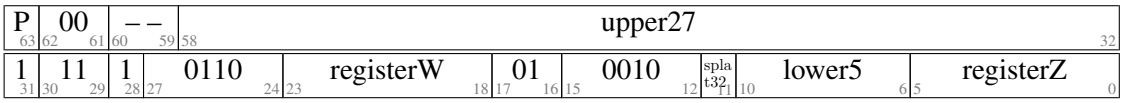
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The unsigned minimum of the *registerY* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MIN(_ZX_32(argument3.32[I]), _ZX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0110



Syntax `minuwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The unsigned minimum of the *upper27_lower5* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MIN(_ZX_32(argument3.32[I]), _ZX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.180 MINW — Minimum of Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 0100

P	11	1	0100	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `minw` *registerW* = *registerZ*, *signed10*

Description The minimum of the *signed10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

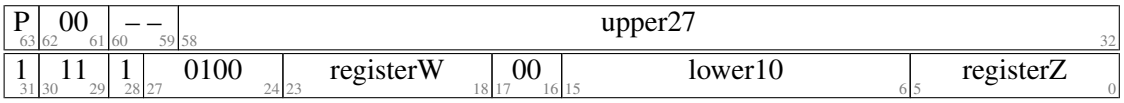
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_32(argument3), _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 0100



Syntax `minw` *registerW* = *registerZ*, *upper27_lower10*

Description The minimum of the *upper27_lower10* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_32(argument3), _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0100

P	11	1	0100	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minw` *registerW* = *registerZ*, *registerY*

Description The minimum of the *registerY* and the *registerZ* is computed. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _MIN(_SX_32(argument3), _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.181 MINWP Minimum of Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0100

P	11	1	0100	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `minwp` *registerW* = *registerZ*, *registerY*

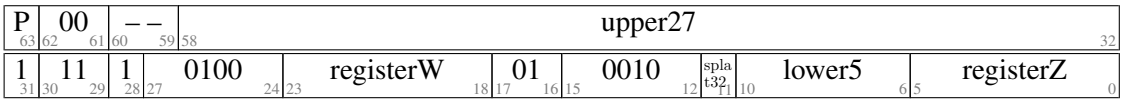
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The minimum of the *registerY* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MIN(_SX_32(argument3.32[I]), _SX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0100



Syntax `minwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The minimum of the *upper27_lower5* words and the *registerZ* words are computed. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _MIN(_SX_32(argument3.32[I]), _SX_32(argument2.32[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.182 NANDD — Bitwise Not And Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1001

P	11	1	1001	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `nandd` *registerW* = *registerZ*, *registerY*

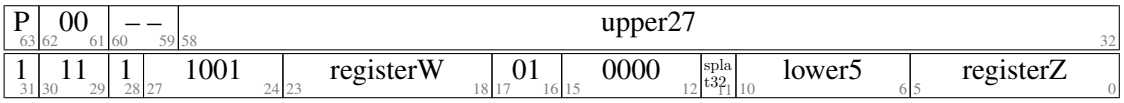
Description Bitwise not of bitwise and of the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1001



Syntax `nandd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise not of bitwise and of the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

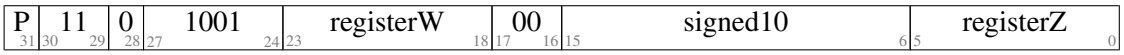
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1001



Syntax `nandd` *registerW* = *registerZ*, *signed10*

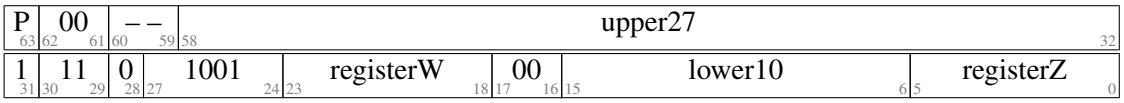
Description Bitwise not of bitwise and of the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1001



Syntax `nandd` *registerW* = *registerZ*, *upper27_lower10*

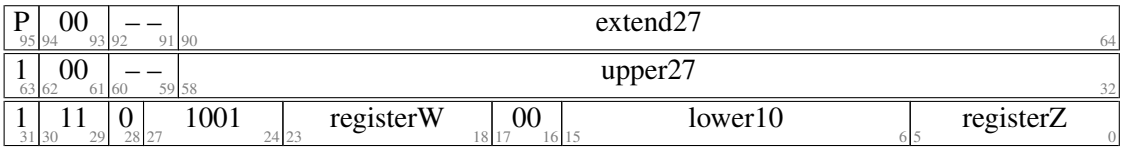
Description Bitwise not of bitwise and of the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1001



Syntax `nandd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description Bitwise not of bitwise and of the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.183 NANDW — Bitwise Not And Between Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1001

P	11	1	1001	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `nandw` *registerW* = *registerZ*, *signed10*

Description Bitwise not of bitwise and of the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

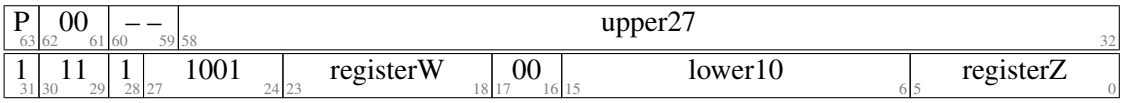
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1001



Syntax `nandw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise not of bitwise and of the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1001

Syntax `nandw` *registerW* = *registerZ*, *registerY*

Description Bitwise not of bitwise and of the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 & argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.184 NEORD Bitwise Not Exclusive Or Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1101

P	11	1	1101	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `neord` *registerW* = *registerZ*, *registerY*

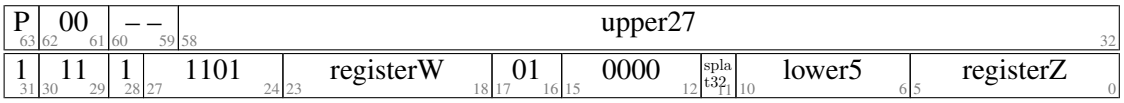
Description Bitwise not of bitwise exclusive or of the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1101



Syntax `neord` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise not of bitwise exclusive or of the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1101

P	11	0	1101	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `neord` *registerW* = *registerZ*, *signed10*

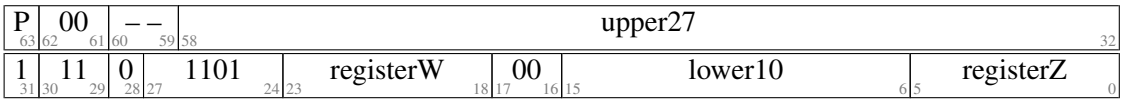
Description Bitwise not of bitwise exclusive or of the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1101



Syntax `neord` *registerW = registerZ, upper27_lower10*

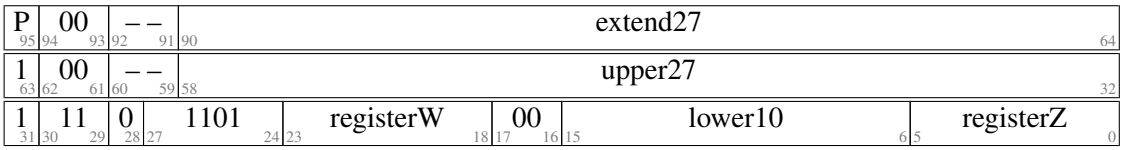
Description Bitwise not of bitwise exclusive or of the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1101



Syntax `neord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description Bitwise not of bitwise exclusive or of the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.185 NEORW Bitwise Not Exclusive Or Between Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1101

P	11	1	1101	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `neorw` *registerW* = *registerZ*, *signed10*

Description Bitwise not of bitwise exclusive or of the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1101

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1101	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `neorw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise not of bitwise exclusive or of the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1101

Syntax `neorw` *registerW* = *registerZ*, *registerY*

Description Bitwise not of bitwise exclusive or of the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.186 NIORD Bitwise Not Inclusive Or Between Double Words

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 1011

P	11	1	1011	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `niord` *registerW* = *registerZ*, *registerY*

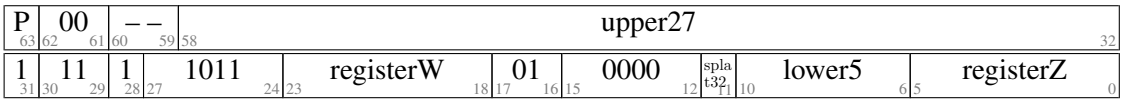
Description Bitwise not of bitwise inclusive or of the *registerZ* with the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 1011



Syntax `niord` *registerW* = *registerZ*, *upper27_lower5splat32*

Description Bitwise not of bitwise inclusive or of the *registerZ* with the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 1011

P	11	0	1011	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `niord` *registerW* = *registerZ*, *signed10*

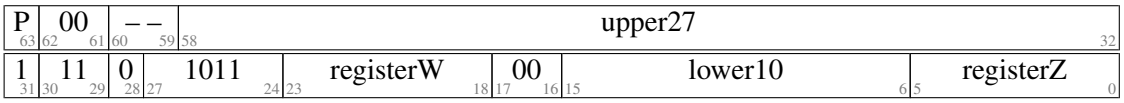
Description Bitwise not of bitwise inclusive or of the *registerZ* with the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 1011



Syntax `niord` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise not of bitwise inclusive or of the *registerZ* with the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 1011

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	11 30 29	0 28	1011 27 24	registerW 23 18	00 17 16	lower10 15 6	registerZ 5 0

Syntax `niord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description Bitwise not of bitwise inclusive or of the *registerZ* with the *extend27_upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.187 NIORW — Bitwise Not Inclusive Or Between Words

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 1011

P	11	1	1011	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `niorw` *registerW* = *registerZ*, *signed10*

Description Bitwise not of bitwise inclusive or of the *registerZ* with the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

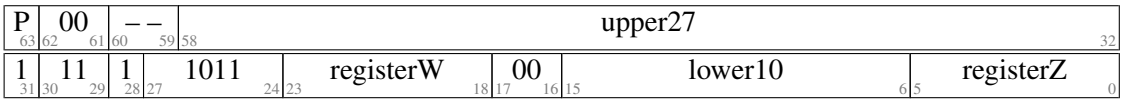
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 1011



Syntax `niorw` *registerW* = *registerZ*, *upper27_lower10*

Description Bitwise not of bitwise inclusive or of the *registerZ* with the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 1011

Syntax `niorw` *registerW* = *registerZ*, *registerY*

Description Bitwise not of bitwise inclusive or of the *registerZ* with the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

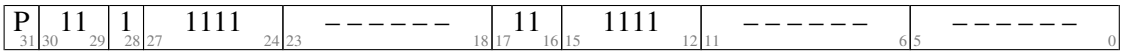
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = ~(argument2 | argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.188 NOP — No Operation

Encoding Format ALU_NOP (Section 6.12.321), exucode2 = 1111



Syntax `nop`

Description No effects except extending the current bundle by one syllable.

Execution Reservation table ALU_NOP (Section 6.14.10)

```
stage RR:
  ;
```


Kalray – all rights reserved

6.6.189 PCREL PC-Relative

Encoding Format ALU_PCREL (Section 6.12.216), exucode2 = 0000

P	11	1	0000	registerW	00	signed16
31	30 29	28	27 24	23 18	17 16	15 0

Syntax `pcrel` *registerW* = *signed16*

Description The *signed16* is is added to the bundle **PC** and the result is stored into the *registerW*.

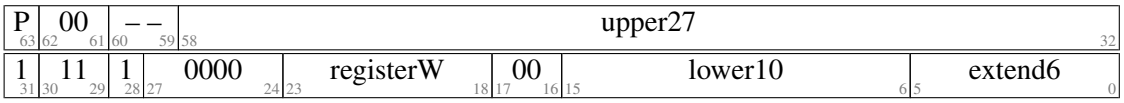
Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage RR:
  new argument2 = _SX_16(signed16);
  new result1 = PC + _SX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_PCREL.X (Section 6.12.217), exucode2 = 0000



Syntax `pcrel` *registerW* = *extend6_upper27_lower10*

Description The *extend6_upper27_lower10* is is added to the bundle **PC** and the result is stored into the *registerW*.

Execution Reservation table ALU_FULL.X (Section 6.14.3)

```
stage RR:
  new argument2 = _SX_43(extend6_upper27_lower10);
  new result1 = PC + _SX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_PCREL.Y (Section 6.12.218), exucode2 = 0000

P [95]	00 [94:93]	– – [92:91]	extend27 [90:64]				
1 [63]	00 [62:61]	– – [60:59]	upper27 [58:32]				
1 [31]	11 [30:29]	1 [28]	0000 [27:24]	registerW [23:18]	00 [17:16]	lower10 [15:6]	– – – – – – [5:0]

Syntax `pcrel` *registerW = extend27_upper27_lower10*

Description The *extend27_upper27_lower10* is is added to the bundle **PC** and the result is stored into the *registerW*.

Execution Reservation table ALU_FULL.Y (Section 6.14.4)

```
stage RR:
  new argument2 = _WX_64(extend27_upper27_lower10);
  new result1 = PC + _SX_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.190 ROLW Rotate Left Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1110

P 31	11 30 29	1 28	1110 27 24	registerW 23 18	10 17 16	0111 15 12	registerY 11 6	registerZ 5 0

Syntax `rolw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is rotated left by the *registerY* modulo 32. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ROL_32(argument2.32[0], _ZX_5(argument3));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1110

P	11	1	1110	registerW	10	0011	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `rolw` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is rotated left by the *unsigned6* modulo 32. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = _ROL_32(argument2.32[0], _ZX_5(argument3));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.191 ROLWPS Rotate Left Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1110

P	11	1	1110	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `rolwps` *registerW* = *registerZ*, *unsigned6*

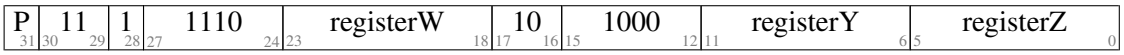
Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are rotated left by the *unsigned6* modulo 32. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ROL_32(argument2.32[I], shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1110



Syntax `rolwps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are rotated left by the *registerY* modulo 32. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ROL_32(argument2.32[I], shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.192 RORW — Rotate Right Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1111

P	11	1	1111	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `rorw` *registerW* = *registerZ*, *registerY*

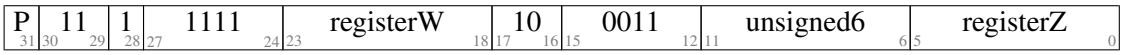
Description The *registerZ* is rotated right by the *registerY* modulo 32. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ROR_32(argument2.32[0], _ZX_5(argument3));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1111



Syntax `rorw` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is rotated right by the *unsigned6* modulo 32. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = _ROR_32(argument2.32[0], _ZX_5(argument3));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.193 RORWPS Rotate Right Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1111

P	11	1	1111	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `rorwps` *registerW* = *registerZ*, *unsigned6*

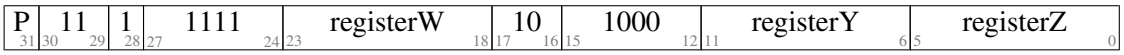
Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are rotated right by the *unsigned6* modulo 32. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ROR_32(argument2.32[I], shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1111



Syntax `rorwps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are rotated right by the *registerY* modulo 32. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ROR_32(argument2.32[I], shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.194 SBFBO Subtract Byte Octuple

Encoding Format ALU_BOWRR0 (Section 6.12.278), exucode2 = 0001

P	11	1	0001	registerW	10	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfbo` *registerW* = *registerZ*, *registerY*

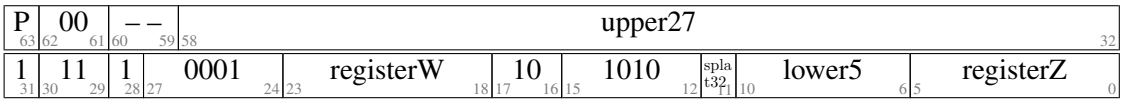
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = argument3.8[I] - argument2.8[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR0.M (Section 6.12.279), exucode2 = 0001



Syntax `sbfbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted from the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = argument3.8[I] - argument2.8[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.195 SBFCD Subtract Double Word from Double Word with Carry

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1111

P	11	1	1111	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfcd` *registerW* = *registerZ*, *registerY*

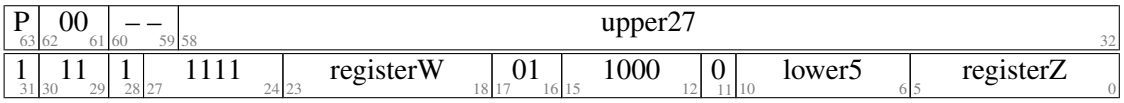
Description The complement of the *registerZ* is added to the *registerY* and the **IC** bit of the **CS** register. The result is stored into the *registerW* and the **IC** bit is updated.

Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_64(~argument2) + _ZX_64(argument3)) + CS.IC;
  new carry = (result1 >> 64) & 1;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  new counter = CS.CC;
  CS.CC = counter + carry;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1111



Syntax `sbfcd` *registerW* = *registerZ*, *upper27_lower5*

Description The complement of the *registerZ* is added to the *upper27_lower5* and the **IC** bit of the **CS** register. The result is stored into the *registerW* and the **IC** bit is updated.

Execution Reservation table ALU_FULL.X (Section 6.14.3)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_64(~argument2) + _ZX_64(argument3)) + CS.IC;
  new carry = (result1 >> 64) & 1;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  new counter = CS.CC;
  CS.CC = counter + carry;
```


Kalray – all rights reserved

©2025 Kalray

6.6.196 SBFCD.I Subtract Double Word from Double Word with Carry Initial

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1111

P	11	1	1111	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfcd.i` *registerW* = *registerZ*, *registerY*

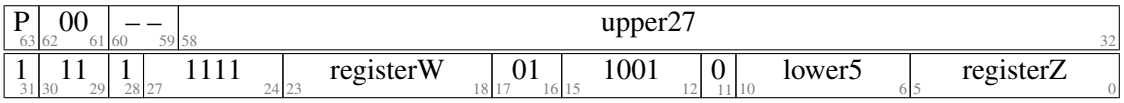
Description The complement of the *registerZ* plus one is added to the *registerY*. The result is stored into the *registerW* and the **IC** bit of the **CS** register is updated.

Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_64(~argument2) + _ZX_64(argument3)) + 1;
  new carry = (result1 >> 64) & 1;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  new counter = CS.CC;
  CS.CC = counter + carry;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1111



Syntax `sbfcd.i` *registerW* = *registerZ*, *upper27_lower5*

Description The complement of the *registerZ* plus one is added to the *upper27_lower5*. The result is stored into the *registerW* and the **IC** bit of the **CS** register is updated.

Execution Reservation table ALU_FULL.X (Section 6.14.3)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_64(~argument2) + _ZX_64(argument3)) + 1;
  new carry = (result1 >> 64) & 1;
stage E1:
  CS.IC = carry;
  GPR[registerW] = result1;
stage E2:
  new counter = CS.CC;
  CS.CC = counter + carry;
```


Kalray – all rights reserved

6.6.197 SBFD Subtract Double Word From Double Word

Encoding Format ALU_DWRR0 (Section 6.12.221), exucode2 = 0011

P	11	1	0011	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfd` *registerW* = *registerZ*, *registerY*

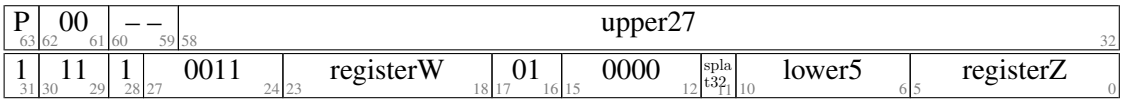
Description The *registerZ* is subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR0.M (Section 6.12.222), exucode2 = 0011



Syntax `sbfd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI (Section 6.12.200), exucode2 = 0011

P	11	0	0011	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `sbfd` *registerW* = *registerZ*, *signed10*

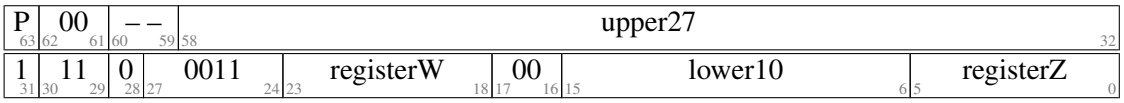
Description The *registerZ* is subtracted from the *signed10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.X (Section 6.12.201), exucode2 = 0011



Syntax `sbfd` *registerW* = *registerZ*, *upper27_lower10*

Description The *registerZ* is subtracted from the *upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRI.Y (Section 6.12.202), exucode2 = 0011

P 95	00 94 93	– – 92 91	extend27 90 64				
1 63	00 62 61	– – 60 59	upper27 58 32				
1 31	11 30 29	0 28	0011 27 24	registerW 23 18	00 17 16	lower10 15 6	registerZ 5 0

Syntax `sbfd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The *registerZ* is subtracted from the *extend27_upper27_lower10*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.198 SBFHQ — Subtract Half Words Quadruple

Encoding Format ALU_HQWRR0 (Section 6.12.227), exucode2 = 0011

P	11	1	0011	registerW	01	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfhq` *registerW* = *registerZ*, *registerY*

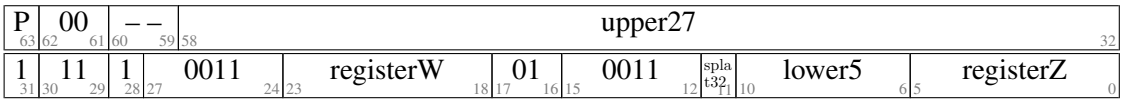
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted from the *registerY* half words. The four resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument3.16[I] - argument2.16[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR0.M (Section 6.12.228), exucode2 = 0011



Syntax `sbfhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted from the *upper27_lower5* half words. The four resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument3.16[I] - argument2.16[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.199 SBFSBO — Subtract Saturated Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 1101

P	11	1	1101	registerW	10	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfsbo` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted with saturation to 8 bits from the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

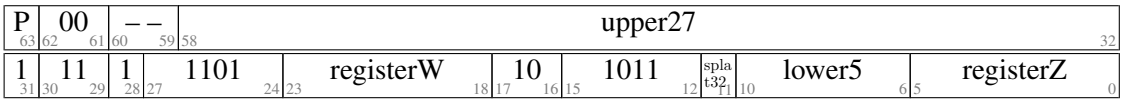
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_SX_8(argument3.8[I]) - _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 1101



Syntax `sbfsbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are subtracted with saturation to 8 bits from the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_SX_8(argument3.8[I]) - _SX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.200 SBFSD — Subtract Saturated Double Words

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1101

P	11	1	1101	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfsd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY* with 64-bit saturation.

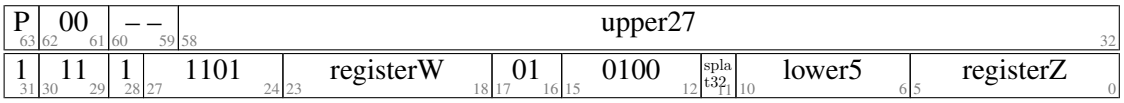
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SAT_64(_SX_64(argument3) - _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1101



Syntax `sbfsd` *registerW = registerZ, upper27_lower5splat32*

Description The *registerZ* is subtracted from the *upper27_lower5* with 64-bit saturation.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _SAT_64(_SX_64(argument3) - _SX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.201 SBFSHQ — Subtract Saturated Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 1101

P	11	1	1101	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfshq` *registerW* = *registerZ*, *registerY*

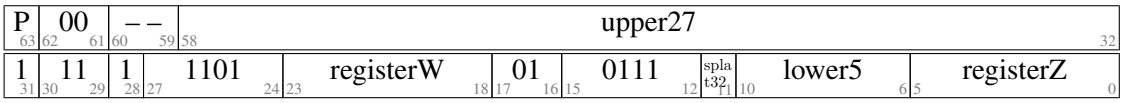
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted with saturation to 16 bits from the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_SX_16(argument3.16[I]) - _SX_16(argument2.16[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 1101



Syntax `sbfshq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are subtracted with saturation to 16 bits from the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_SX_16(argument3.16[I]) - _SX_16(argument2.16[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.202 SBFSW — Subtract Saturated Words

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1101

P	11	1	1101	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfsw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY* with 32-bit saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

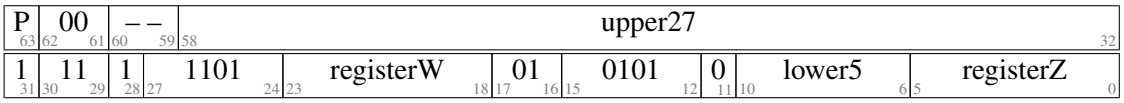
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SAT_32(_SX_32(argument3) - _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1101



Syntax `sbfsw` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is subtracted from the *upper27_lower5* with 32-bit saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SAT_32(_SX_32(argument3) - _SX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.203 SBFSWP Subtract Saturated Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 1101

P	11	1	1101	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfswp` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are subtracted with saturation to 32 bits from the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_SX_32(argument3.32[I]) - _SX_32(argument2.32[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 1101

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	11	1	1101				registerW					01	0110			splat32	lower5					registerZ										
31	30	29	28	27						24	23	18	17	16	15	12	11				10	6	5									0

Syntax `sbfswp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are subtracted with saturation to 32 bits from the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

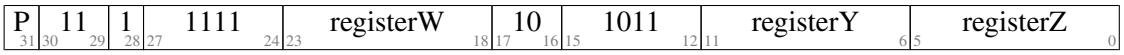
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_SX_32(argument3.32[I]) - _SX_32(argument2.32[I])
        )
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.204 SBFUSBO Subtract Unsigned Saturated Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 1111



Syntax `sbfusbo` *registerW* = *registerZ*, *registerY*

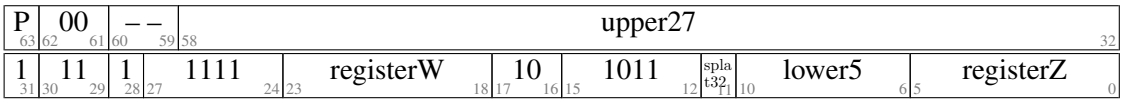
Description The *registerY* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *registerZ* bytes are subtracted with unsigned saturation to 8 bits from the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SATU_8(_ZX_8(argument3.8[I]) - _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 1111



Syntax `sbfusbo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight unsigned bytes packed into 64 bits. The *registerZ* bytes are subtracted with unsigned saturation to 8 bits from the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SATU_8(_ZX_8(argument3.8[I]) - _ZX_8(argument2.8[I]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.205 SBFUSD Subtract Unsigned Saturated Double Words

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1111

P	11	1	1111	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfusd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY* with 64-bit unsigned saturation.

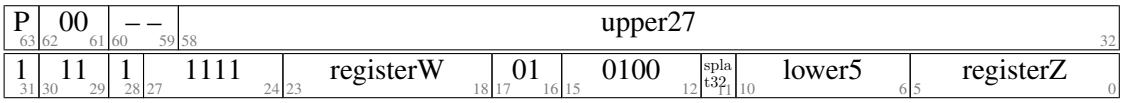
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SATU_64(_ZX_64(argument3) - _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1111



Syntax `sbfusd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is subtracted from the *upper27_lower5* with 64-bit unsigned saturation.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _SATU_64(_ZX_64(argument3) - _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.206 SBFUSHQ Subtract Unsigned Saturated Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 1111

P	11	1	1111	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfushq` *registerW* = *registerZ*, *registerY*

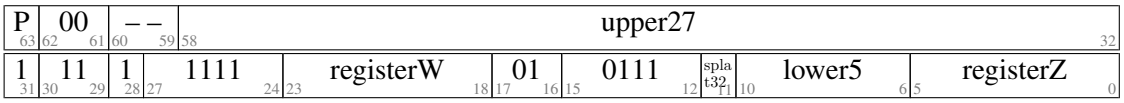
Description The *registerY* and the *registerZ* are considered as four unsigned half words packed into 64 bits. The *registerZ* half words are subtracted with unsigned saturation to 16 bits from the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SATU_16(_ZX_16(argument3.16[I]) - _ZX_16(argument2.16[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 1111



Syntax `sbfushq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four unsigned half words packed into 64 bits. The *registerZ* half words are subtracted with unsigned saturation to 16 bits from the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SATU_16(_ZX_16(argument3.16[I]) - _ZX_16(argument2.16[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.207 SBFUSW — Subtract Unsigned Saturated Words

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1111

P	11	1	1111	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfusw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY* with 32-bit unsigned saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

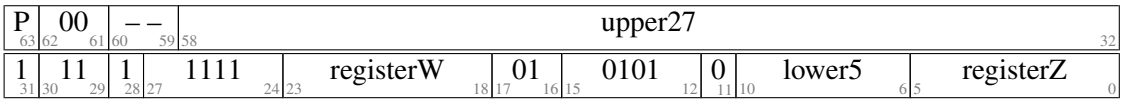
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SATU_32(_ZX_32(argument3) - _ZX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1111



Syntax `sbfusw` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is subtracted from the *upper27_lower5* with 32-bit unsigned saturation. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SATU_32(_ZX_32(argument3) - _ZX_32(argument2));
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.208 SBFUSWP Subtract Unsigned Saturated Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 1111

P	11	1	1111	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfuswp` *registerW* = *registerZ*, *registerY*

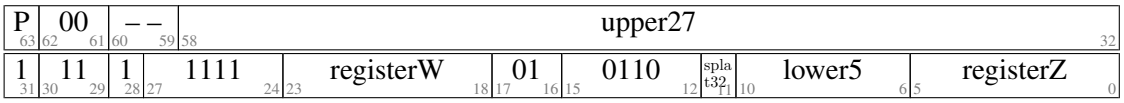
Description The *registerY* and the *registerZ* are considered as two unsigned words packed into 64 bits. The *registerZ* words are subtracted with unsigned saturation to 32 bits from the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SATU_32(_ZX_32(argument3.32[I]) - _ZX_32(argument2.32[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 1111



Syntax `sbfuswp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two unsigned words packed into 64 bits. The *registerZ* words are subtracted with unsigned saturation to 32 bits from the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SATU_32(_ZX_32(argument3.32[I]) - _ZX_32(argument2.32[I
        ]))
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.209 SBFUWD Subtract Unsigned Word from Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1101

P	11	1	1101	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfuwd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is zero-extended from 32 bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - argument2.32[0];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1101

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1101	registerW	01	1001	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `sbfuwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - argument2.32[0];
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.210 SBFW — Subtract Word from Word

Encoding Format ALU_WRIW (Section 6.12.219), exucode2 = 0011

P	11	1	0011	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `sbfw` *registerW* = *registerZ*, *signed10*

Description The *registerZ* is subtracted from the *signed10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRIW.X (Section 6.12.220), exucode2 = 0011

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0011	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `sbfw` *registerW* = *registerZ*, *upper27_lower10*

Description The *registerZ* is subtracted from the *upper27_lower10*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR0 (Section 6.12.223), exucode2 = 0011

P	11	1	0011	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - argument2;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.211 SBFWD — Subtract Word from Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1101

P	11	1	1101	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfwd` *registerW* = *registerZ*, *registerY*

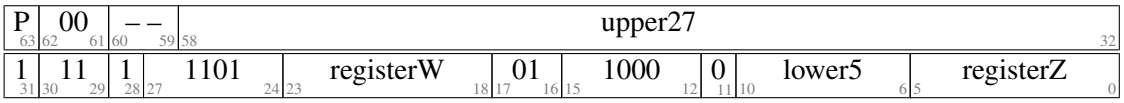
Description The *registerZ* is sign-extended from 32 bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - _SX_32(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1101



Syntax `sbfwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - _SX_32(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.212 SBFWP — Subtract Words Packed

Encoding Format ALU_WPWRR0 (Section 6.12.225), exucode2 = 0011

P	11	1	0011	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfwp` *registerW* = *registerZ*, *registerY*

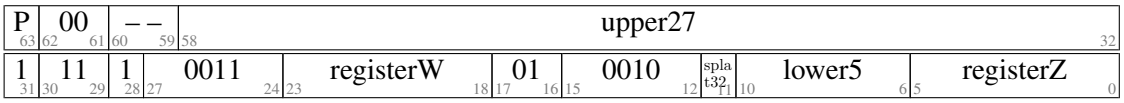
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are subtracted from the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument3.32[I] - argument2.32[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR0.M (Section 6.12.226), exucode2 = 0011



Syntax `sbfwp` *registerW = registerZ, upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are subtracted from the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument3.32[I] - argument2.32[I]
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.213 SBFX16BO Subtract Integer to Integer Times 16 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0111

P	11	1	0111	registerW	10	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx16bo` *registerW* = *registerZ*, *registerY*

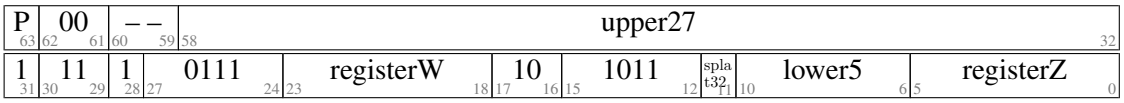
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by four bits and subtracted from the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0111



Syntax `sbfx16bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by four bits and subtracted from the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.214 SBFX16D Subtract Double Word Times 16 from Double Word

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0111

P	11	1	0111	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx16d` *registerW* = *registerZ*, *registerY*

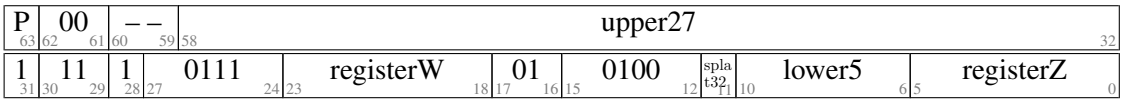
Description The *registerZ* is shifted left by four bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0111



Syntax `sbfx16d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by four bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.215 SBFX16HQ Subtract Integer to Integer Times 16 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0111

P	11	1	0111	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx16hq` *registerW* = *registerZ*, *registerY*

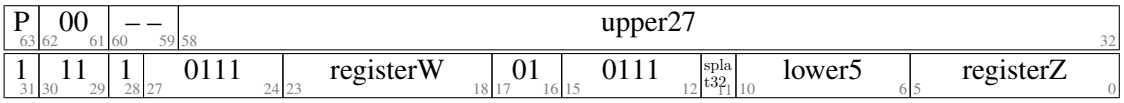
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by four bits and subtracted from the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0111



Syntax `sbfx16hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by four bits and subtracted from the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.216 SBFX16UWD Subtract Unsigned Word Times 16 from Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0111

P	11	1	0111	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx16uwd` *registerW* = *registerZ*, *registerY*

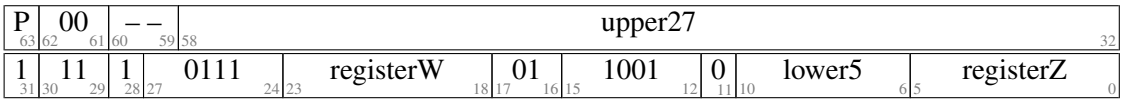
Description The *registerZ* is zero-extended from 32 bits, shifted left by four bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0111



Syntax `sbfx16uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by four bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.217 SBFX16W Subtract Word Times 16 from Word

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0111

P	11	1	0111	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx16w` *registerW* = *registerZ*, *registerY*

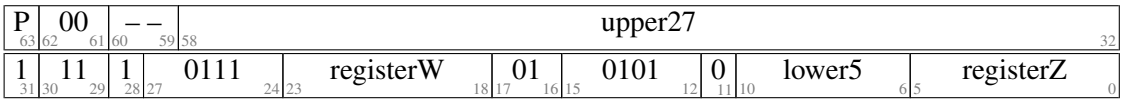
Description The *registerZ* is shifted left by four bits and subtracted from the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 4);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0111



Syntax `sbfx16w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by four bits and subtracted from the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 4);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.218 SBFX16WD Subtract Word Times 16 from Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0111

P	11	1	0111	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx16wd` *registerW* = *registerZ*, *registerY*

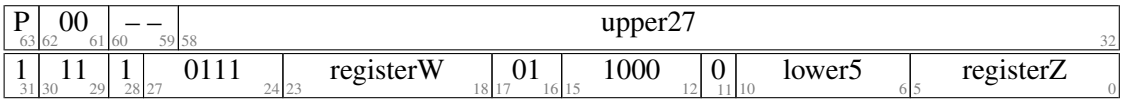
Description The *registerZ* is sign-extended from 32 bits, shifted left by four bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0111



Syntax `sbfx16wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by four bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 4);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.219 SBFX16WP Subtract Integer to Integer Times 16 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0111

P	11	1	0111	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx16wp` *registerW* = *registerZ*, *registerY*

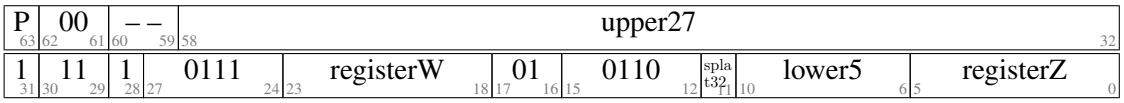
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by four bits and subtracted from the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0111



Syntax `sbfx16wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by four bits and subtracted from the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 4)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.220 SBFX2BO Subtract Integer to Integer Times 2 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0001

P	11	1	0001	registerW	10	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx2bo` *registerW* = *registerZ*, *registerY*

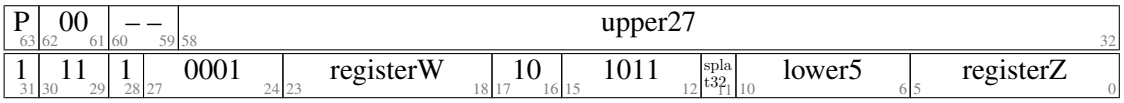
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by one bit and subtracted from the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0001



Syntax `sbfx2bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by one bit and subtracted from the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.221 SBFX2D Subtract Double Word Times 2 from Double Word

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0001

P	11	1	0001	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx2d` *registerW* = *registerZ*, *registerY*

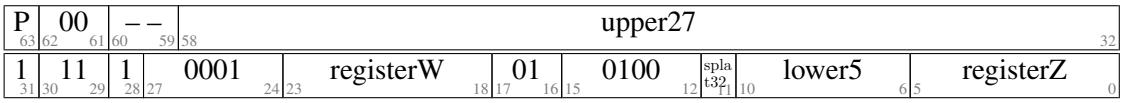
Description The *registerZ* is shifted left by one bit and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0001



Syntax `sbfx2d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by one bit and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.222 SBFX2HQ Subtract Integer to Integer Times 2 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0001

P	11	1	0001	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx2hq` *registerW* = *registerZ*, *registerY*

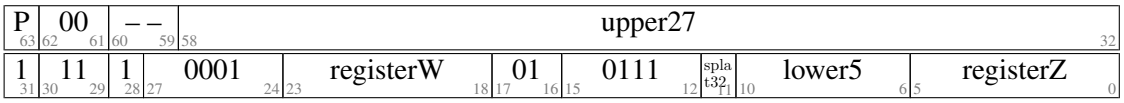
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by one bit and subtracted from the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0001



Syntax `sbfx2hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by one bit and subtracted from the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.223 SBFX2UWD Subtract Unsigned Word Times 2 from Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0001

P	11	1	0001	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx2uwd` *registerW* = *registerZ*, *registerY*

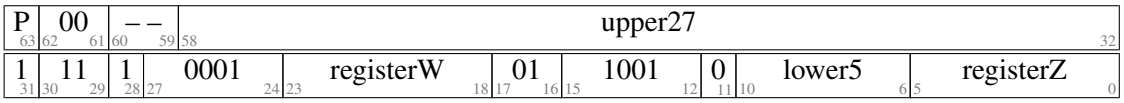
Description The *registerZ* is zero-extended from 32 bits, shifted left by one bit and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0001



Syntax `sbfx2uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by one bit and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.224 SBFX2W Subtract Word Times 2 from Word

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0001

P	11	1	0001	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx2w` *registerW* = *registerZ*, *registerY*

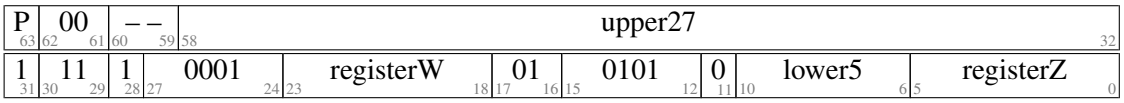
Description The *registerZ* is shifted left by one bit and subtracted from the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 1);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0001



Syntax `sbfx2w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by one bit and subtracted from the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 1);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.225 SBFX2WD Subtract Word Times 2 from Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0001

P	11	1	0001	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx2wd` *registerW* = *registerZ*, *registerY*

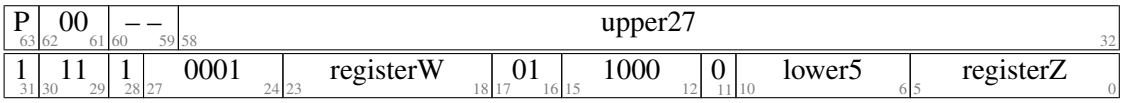
Description The *registerZ* is sign-extended from 32 bits, shifted left by one bit and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0001



Syntax `sbfx2wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by one bit and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 1);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.226 SBFX2WP Subtract Integer to Integer Times 2 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0001

P	11	1	0001	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx2wp` *registerW* = *registerZ*, *registerY*

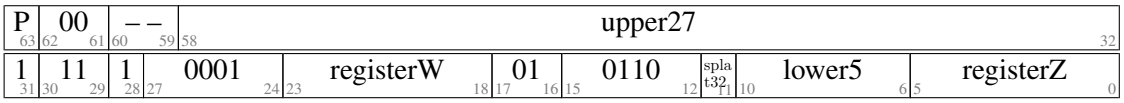
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by one bit and subtracted from the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0001



Syntax `sbfx2wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by one bit and subtracted from the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 1)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.227 SBFX32D Subtract Double Word Times 32 from Double Word

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1001

P	11	1	1001	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx32d` *registerW* = *registerZ*, *registerY*

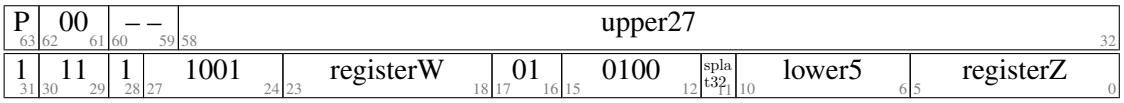
Description The *registerZ* is shifted left by five bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1001



Syntax `sbfx32d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by five bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.228 SBFX32UWD Subtract Unsigned Word Times 32 from Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1001

P	11	1	1001	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx32uwd` *registerW* = *registerZ*, *registerY*

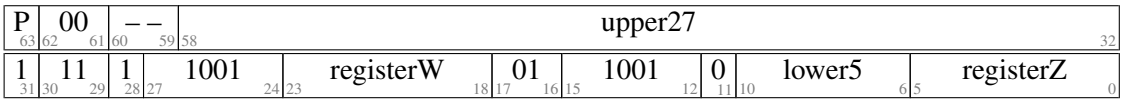
Description The *registerZ* is zero-extended from 32 bits, shifted left by five bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1001



Syntax `sbfx32uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by five bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.229 SBFX32W Subtract Word Times 32 from Word

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1001

P	11	1	1001	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx32w` *registerW* = *registerZ*, *registerY*

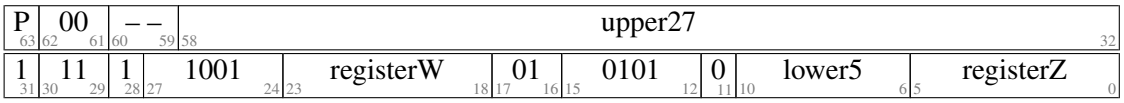
Description The *registerZ* is shifted left by five bits and subtracted from the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 5);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1001



Syntax `sbfx32w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by five bits and subtracted from the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 5);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.230 SBFX32WD Subtract Word Times 32 from Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1001

P	11	1	1001	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx32wd` *registerW* = *registerZ*, *registerY*

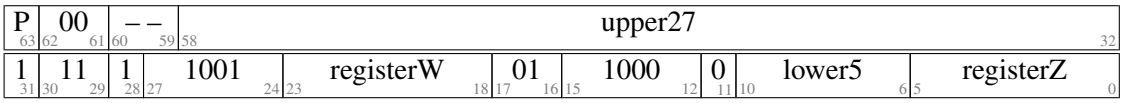
Description The *registerZ* is sign-extended from 32 bits, shifted left by five bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1001



Syntax `sbfx32wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by five bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 5);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.231 SBFX4BO Subtract Integer to Integer Times 4 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0011

P	11	1	0011	registerW	10	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx4bo` *registerW* = *registerZ*, *registerY*

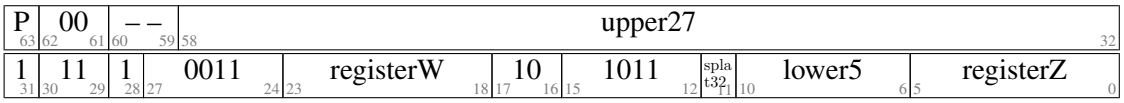
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by two bits and subtracted from the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0011



Syntax `sbfx4bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by two bits and subtracted from the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.232 SBFX4D Subtract Double Word Times 4 from Double Word

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0011

P	11	1	0011	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx4d` *registerW* = *registerZ*, *registerY*

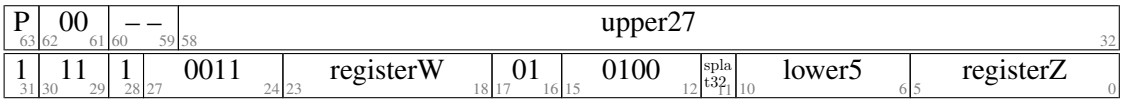
Description The *registerZ* is shifted left by two bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0011



Syntax `sbfx4d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by two bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.233 SBFX4HQ Subtract Integer to Integer Times 4 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0011

P	11	1	0011	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx4hq` *registerW* = *registerZ*, *registerY*

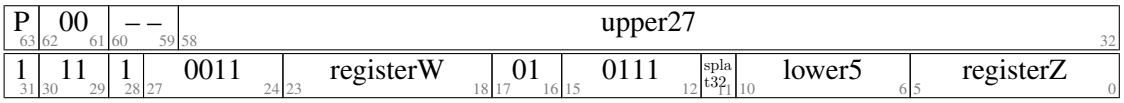
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by two bits and subtracted from the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0011



Syntax `sbfx4hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by two bits and subtracted from the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.234 SBFX4UWD Subtract Unsigned Word Times 4 from Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0011

P	11	1	0011	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx4uwd` *registerW* = *registerZ*, *registerY*

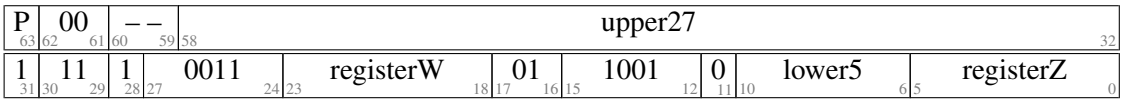
Description The *registerZ* is zero-extended from 32 bits, shifted left by two bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0011



Syntax `sbfx4uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by two bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.235 SBFX4W Subtract Word Times 4 from Word

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0011

P	11	1	0011	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx4w` *registerW* = *registerZ*, *registerY*

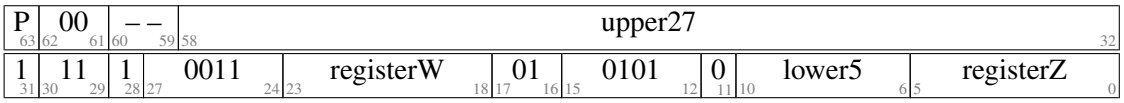
Description The *registerZ* is shifted left by two bits and subtracted from the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 2);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0011



Syntax `sbfx4w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by two bits and subtracted from the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 2);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.236 SBFX4WD Subtract Word Times 4 from Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0011

P	11	1	0011	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx4wd` *registerW* = *registerZ*, *registerY*

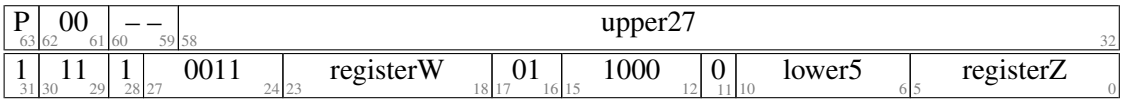
Description The *registerZ* is sign-extended from 32 bits, shifted left by two bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0011



Syntax `sbfx4wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by two bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.237 SBFX4WP Subtract Integer to Integer Times 4 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0011

P	11	1	0011	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx4wp` *registerW* = *registerZ*, *registerY*

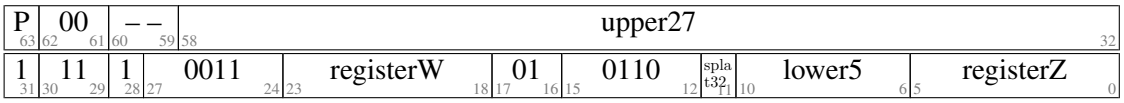
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by two bits and subtracted from the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0011



Syntax `sbfx4wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by two bits and subtracted from the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 2)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.238 SBFX64D Subtract Double Word Times 64 from Double Word

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 1011

P	11	1	1011	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx64d` *registerW* = *registerZ*, *registerY*

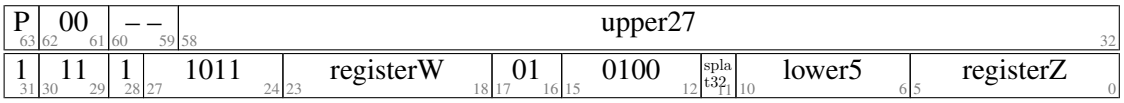
Description The *registerZ* is shifted left by six bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 1011



Syntax `sbfx64d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by six bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.239 SBFX64UWD Subtract Unsigned Word Times 64 from Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 1011

P	11	1	1011	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx64uwd` *registerW* = *registerZ*, *registerY*

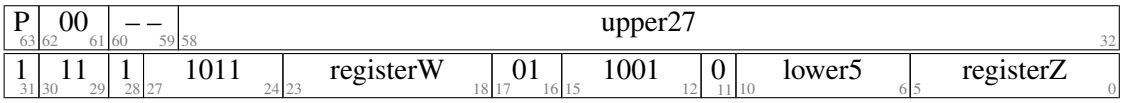
Description The *registerZ* is zero-extended from 32 bits, shifted left by six bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 1011



Syntax `sbfx64uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by six bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.240 SBFX64W — Subtract Word Times 64 from Word

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 1011

P	11	1	1011	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx64w` *registerW* = *registerZ*, *registerY*

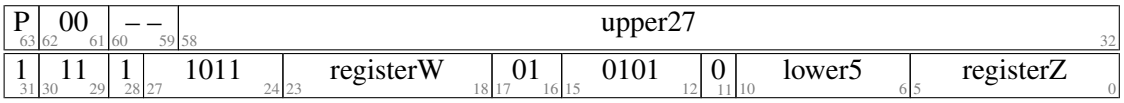
Description The *registerZ* is shifted left by six bits and subtracted from the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 6);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 1011



Syntax `sbfx64w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by six bits and subtracted from the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 6);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.241 SBFX64WD Subtract Word Times 64 from Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 1011

P	11	1	1011	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx64wd` *registerW* = *registerZ*, *registerY*

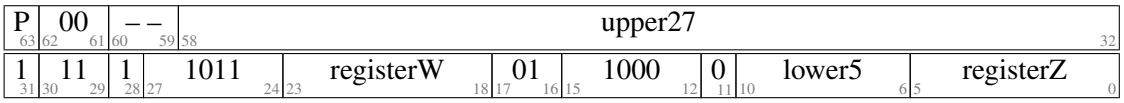
Description The *registerZ* is sign-extended from 32 bits, shifted left by six bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 1011



Syntax `sbfx64wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by six bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 6);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.242 SBFX8BO Subtract Integer to Integer Times 8 Byte Octuple

Encoding Format ALU_BOWRR1 (Section 6.12.280), exucode2 = 0101

P	11	1	0101	registerW	10	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx8bo` *registerW* = *registerZ*, *registerY*

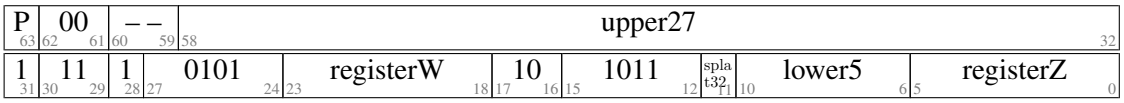
Description The *registerY* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by three bits and subtracted from the *registerY* bytes. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOWRR1.M (Section 6.12.281), exucode2 = 0101



Syntax `sbfx8bo` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by three bits and subtracted from the *upper27_lower5* bytes. The resulting bytes are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument3.8[I]) - (_SX_8(argument2.8[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.243 SBFX8D Subtract Double Word Times 8 from Double Word

Encoding Format ALU_DWRR1 (Section 6.12.229), exucode2 = 0101

P	11	1	0101	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx8d` *registerW* = *registerZ*, *registerY*

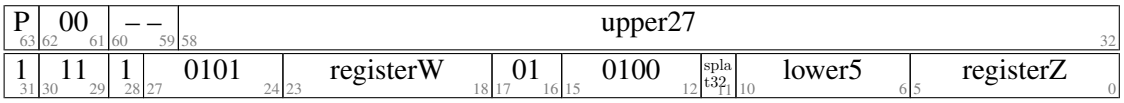
Description The *registerZ* is shifted left by three bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRR1.M (Section 6.12.230), exucode2 = 0101



Syntax `sbfx8d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is shifted left by three bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.244 SBFX8HQ Subtract Integer to Integer Times 8 Half Words Quadruple

Encoding Format ALU_HQWRR1 (Section 6.12.235), exucode2 = 0101

P	11	1	0101	registerW	01	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx8hq` *registerW* = *registerZ*, *registerY*

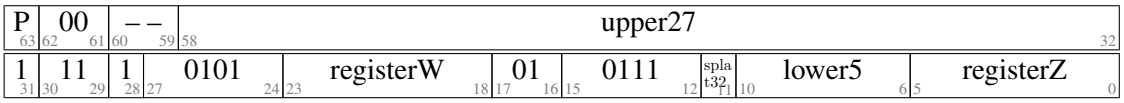
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by three bits and subtracted from the *registerY* half words. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRR1.M (Section 6.12.236), exucode2 = 0101



Syntax `sbfx8hq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by three bits and subtracted from the *upper27_lower5* half words. The resulting half words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) - (_SX_16(argument2.16[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.245 SBFX8UWD Subtract Unsigned Word Times 8 from Double Word

Encoding Format ALU_WDWRR3 (Section 6.12.239), exucode2 = 0101

P	11	1	0101	registerW	01	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx8uwd` *registerW* = *registerZ*, *registerY*

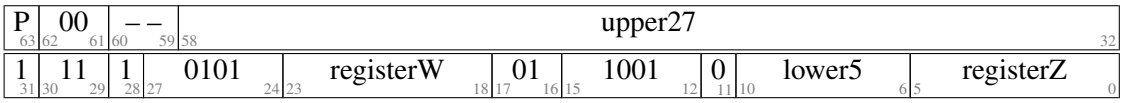
Description The *registerZ* is zero-extended from 32 bits, shifted left by three bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR3.W (Section 6.12.240), exucode2 = 0101



Syntax `sbfx8uwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is zero-extended from 32 bits, shifted left by three bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (argument2.32[0] << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.246 SBFX8W — Subtract Word Times 8 from Word

Encoding Format ALU_WWRR1 (Section 6.12.231), exucode2 = 0101

P	11	1	0101	registerW	01	0101	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx8w` *registerW* = *registerZ*, *registerY*

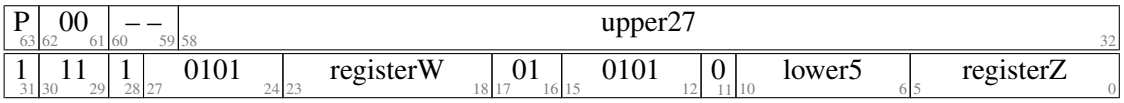
Description The *registerZ* is shifted left by three bits and subtracted from the *registerY*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WWRR1.W (Section 6.12.232), exucode2 = 0101



Syntax `sbfx8w` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is shifted left by three bits and subtracted from the *upper27_lower5*. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3 - (argument2 << 3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.247 SBFX8WD Subtract Word Times 8 from Double Word

Encoding Format ALU_WDWRR2 (Section 6.12.237), exucode2 = 0101

P	11	1	0101	registerW	01	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx8wd` *registerW* = *registerZ*, *registerY*

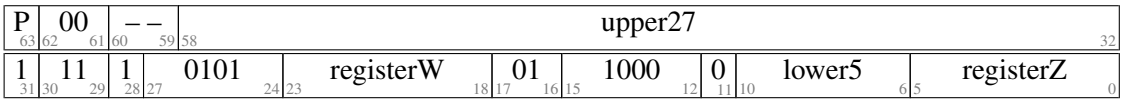
Description The *registerZ* is sign-extended from 32 bits, shifted left by three bits and subtracted from the *registerY*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WDWRR2.W (Section 6.12.238), exucode2 = 0101



Syntax `sbfx8wd` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is sign-extended from 32 bits, shifted left by three bits and subtracted from the *upper27_lower5*. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) - (_SX_32(argument2) << 3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.248 SBFX8WP Subtract Integer to Integer Times 8 Words Packed

Encoding Format ALU_WPWRR1 (Section 6.12.233), exucode2 = 0101

P	11	1	0101	registerW	01	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbfx8wp` *registerW* = *registerZ*, *registerY*

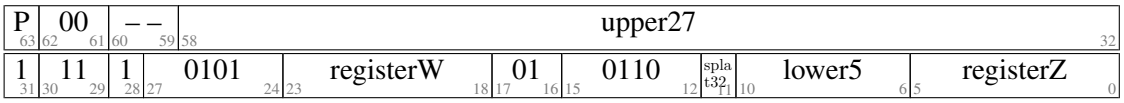
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by three bits and subtracted from the *registerY* words. The two resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRR1.M (Section 6.12.234), exucode2 = 0101



Syntax `sbfx8wp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *registerZ* words are shifted left by three bits and subtracted from the *upper27_lower5* words. The two resulting words are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) - (_SX_32(argument2.32[I]) << 3)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.249 SBMM8D Swapped Bit Matrix Multiplication 8×8

Encoding Format ALU_BWRR (Section 6.12.257), exucode2 = 1110

P	11	1	1110	registerW	10	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbmm8d` *registerW* = *registerZ*, *registerY*

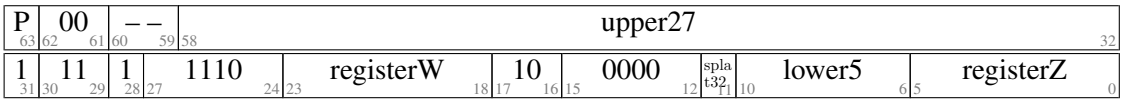
Description The *registerY* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _BMM_8(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BWRR.M (Section 6.12.258), exucode2 = 1110



Syntax `sbmm8d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _BMM_8(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DBMWRI (Section 6.12.212), exucode2 = 1110

P	11	0	1110	registerW	10	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Syntax `sbmm8d` *registerW* = *registerZ*, *signed10*

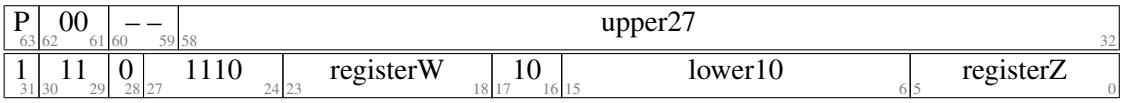
Description The *signed10* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _BMM_8(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DBMWRI.X (Section 6.12.213), exucode2 = 1110



Syntax `sbmm8d` *registerW* = *registerZ*, *upper27_lower10*

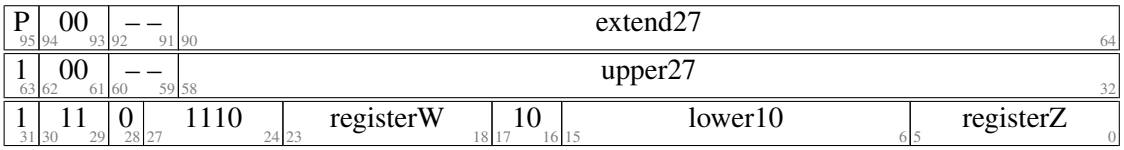
Description The *upper27_lower10* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _BMM_8(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DBMWRI.Y (Section 6.12.214), exucode2 = 1110



Syntax `sbmm8d` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The *extend27_upper27_lower10* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _BMM_8(_ZX_64(argument3), _ZX_64(argument2));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.250 SBMMT8D Swapped Bit Matrix Multiplication Transposed 8×8

Encoding Format ALU_BWRR (Section 6.12.257), exucode2 = 1111

P	11	1	1111	registerW	10	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sbmmt8d` *registerW* = *registerZ*, *registerY*

Description The *registerY* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is transposed, and stored into the *registerW*.

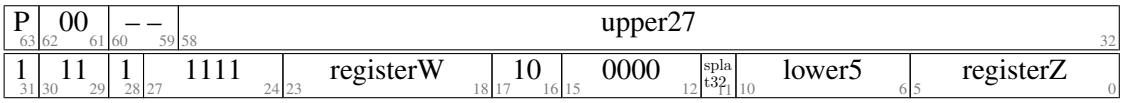
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _BMT_8(_BMM_8(_ZX_64(argument3), _ZX_64(argument2)));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_BWRR.M (Section 6.12.258), exucode2 = 1111



Syntax `sbmmt8d` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is transposed, and stored into the *registerW*.

Modifier(s)

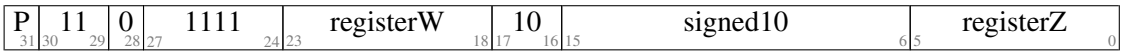
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _BMT_8(_BMM_8(_ZX_64(argument3), _ZX_64(argument2)));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DBMWRI (Section 6.12.212), exucode2 = 1111



Syntax `sbmmt8d` *registerW* = *registerZ*, *signed10*

Description The *signed10* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is transposed, and stored into the *registerW*.

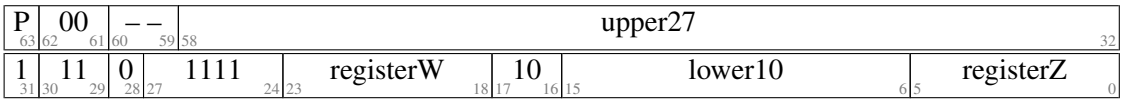
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _SX_10(signed10);
  new argument2 = GPR[registerZ];
  new result1 = _BMT_8(_BMM_8(_ZX_64(argument3), _ZX_64(argument2)));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_DBMWRI.X (Section 6.12.213), exucode2 = 1111



Syntax `sbmmt8d` *registerW* = *registerZ*, *upper27_lower10*

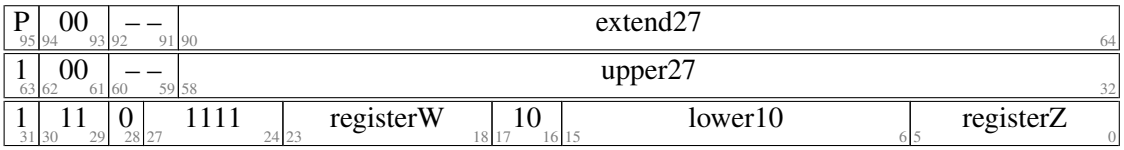
Description The *upper27_lower10* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is transposed, and stored into the *registerW*.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _SX_37(upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _BMT_8(_BMM_8(_ZX_64(argument3), _ZX_64(argument2)));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DBMWRI.Y (Section 6.12.214), exucode2 = 1111



Syntax `sbmmt8d` *registerW* = *registerZ*, *extend27_upper27_lower10*

Description The *extend27_upper27_lower10* interpreted as a 8×8 bit matrix is multiplied by the *registerZ* interpreted as a 8×8 bit matrix. The result is transposed, and stored into the *registerW*.

Execution Reservation table ALU_TINY.Y (Section 6.14.13)

```
stage RR:
  new argument3 = _WX_64(extend27_upper27_lower10);
  new argument2 = GPR[registerZ];
  new result1 = _BMT_8(_BMM_8(_ZX_64(argument3), _ZX_64(argument2)));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.251 SLLBOS Shift Left Logical Byte Octuple by Scalar

Encoding Format ALU_BOSWRI (Section 6.12.288), exucode2 = 1001

P	11	1	1001	registerW	10	1110	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sllbos` *registerW* = *registerZ*, *unsigned6*

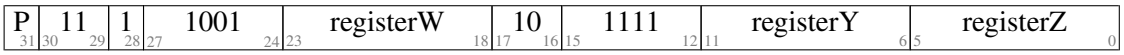
Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by the *unsigned6* modulo 8. The vacant positions are filled in with zeros. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = argument2.8[I] << shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 1001



Syntax `sllbos` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by the *registerY* modulo 8. The vacant positions are filled in with zeros. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = argument2.8[I] << shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.252 SLLD Shift Left Logical Double Word

Encoding Format ALU_DSWRI (Section 6.12.264), exucode2 = 1001

P	11	1	1001	registerW	10	0010	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slld` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted left by the *unsigned6* modulo 64. The vacant positions are filled in with zeros. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2) << _ZX_6(argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 1001

P	11	1	1001	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slld` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by the *registerY* modulo 64. The vacant positions are filled in with zeros. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2) << _ZX_6(argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.253 SLLHQS Shift Left Logical Half Word Quadruple by Scalar

Encoding Format ALU_HQSWRI (Section 6.12.270), exucode2 = 1001

P	11	1	1001	registerW	10	0101	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sllhqs` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by the *unsigned6* modulo 16. The vacant positions are filled in with zeros. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument2.16[I] << shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 1001

P	11	1	1001	registerW	10	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sllhqs` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by the *registerY* modulo 16. The vacant positions are filled in with zeros. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument2.16[I] << shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.254 SLLW — Shift Left Logical Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1001

P	11	1	1001	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sllw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by the *registerY* modulo 32. The vacant positions are filled in with zeros. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] << _ZX_5(argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1001

P	11	1	1001	registerW	10	0011	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sllw` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted left by the *unsigned6* modulo 32. The vacant positions are filled in with zeros. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] << _ZX_5(argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.255 SLLWPS Shift Left Logical Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1001

P	11	1	1001	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sllwps` *registerW* = *registerZ*, *unsigned6*

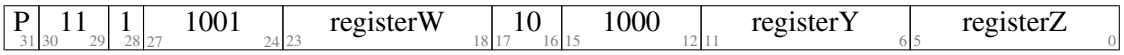
Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted left by the *unsigned6* modulo 32. The vacant positions are filled in with zeros. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument2.32[I] << shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1001



Syntax `sllwps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted left by the *registerY* modulo 32. The vacant positions are filled in with zeros. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument2.32[I] << shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.256 SLSBOS Shift Left Saturated Byte Octuple by Scalar

Encoding Format ALU_BOSWRI (Section 6.12.288), exucode2 = 1100

P	11	1	1100	registerW	10	1110	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slsbos` *registerW* = *registerZ*, *unsigned6*

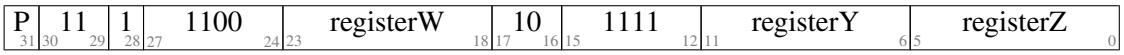
Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by the *unsigned6* modulo 8. The resulting bytes are saturated to 8 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_SX_8(argument2.8[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 1100



Syntax `slsbos` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted left by the *registerY* modulo 8. The resulting bytes are saturated to 8 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SAT_8(_SX_8(argument2.8[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.257 SLSD Shift Left Saturated Double Word

Encoding Format ALU_DSWRI (Section 6.12.264), exucode2 = 1100

P	11	1	1100	registerW	10	0010	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slsd` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted left by the *unsigned6* modulo 64. The result is saturated to 64 bits and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_6(argument3);
  new result1 = _SAT_64(_SX_64(argument2) << shift);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 1100

P	11	1	1100	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slsd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by the *registerY* modulo 64. The result is saturated to 64 bits and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_6(argument3);
  new result1 = _SAT_64(_SX_64(argument2) << shift);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.258 SLSHQS Shift Left Saturated Half Word Quadruple by Scalar

Encoding Format ALU_HQSWRI (Section 6.12.270), exucode2 = 1100

P	11	1	1100	registerW	10	0101	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slshqs` *registerW* = *registerZ*, *unsigned6*

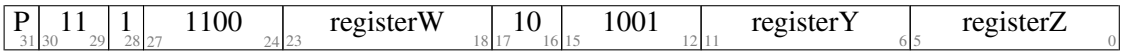
Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by the *unsigned6* modulo 16. The resulting half words are saturated to 16 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_SX_16(argument2.16[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 1100



Syntax `slshqs` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted left by the *registerY* modulo 16. The resulting half words are saturated to 16 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SAT_16(_SX_16(argument2.16[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.259 SLSW Shift Left Saturated Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1100

P	11	1	1100	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slsw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by the *registerY* modulo 32 then is saturated to 32 bits. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new result1 = _SAT_32(_SX_32(argument2) << shift);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1100

P	11	1	1100	registerW	10	0011	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slsw` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted left by the *unsigned6* modulo 32 then is saturated to 32 bits. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new result1 = _SAT_32(_SX_32(argument2) << shift);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.260 SLSWPS Shift Left Saturated Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1100

P	11	1	1100	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slswps` *registerW* = *registerZ*, *unsigned6*

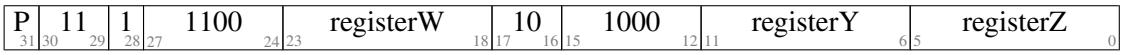
Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted left by the *unsigned6* modulo 32. The resulting words are saturated to 32 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_SX_32(argument2.32[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1100



Syntax `slswps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted left by the *registerY* modulo 32. The resulting words are saturated to 32 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SAT_32(_SX_32(argument2.32[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.261 SLUSBOS Shift Left Unsigned Saturated Byte Octuple by Scalar

Encoding Format ALU_BOSWRI (Section 6.12.288), exucode2 = 1101

P	11	1	1101	registerW	10	1110	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slusbos` *registerW* = *registerZ*, *unsigned6*

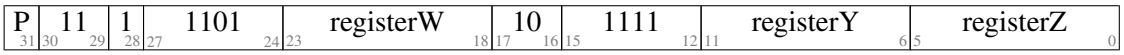
Description The *registerZ* is considered as eight unsigned bytes packed into 64 bits. The *registerZ* bytes are shifted left by the *unsigned6* modulo 8. The resulting bytes are unsigned saturated to 8 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SATU_8(_ZX_8(argument2.8[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 1101



Syntax `slusbos` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as eight unsigned bytes packed into 64 bits. The *registerZ* bytes are shifted left by the *registerY* modulo 8. The resulting bytes are unsigned saturated to 8 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SATU_8(_ZX_8(argument2.8[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.262 SLUSD Shift Left Unsigned Saturated Double Word

Encoding Format ALU_DSWRI (Section 6.12.264), exucode2 = 1101

P	11	1	1101	registerW	10	0010	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slusd` *registerW* = *registerZ*, *unsigned6*

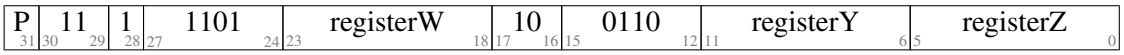
Description The *registerZ* is shifted left by the *unsigned6* modulo 64. The result is unsigned saturated to 64 bits and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_6(argument3);
  new result1 = _SATU_64(_ZX_64(argument2) << shift);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 1101



Syntax `slusd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by the *registerY* modulo 64. The result is unsigned saturated to 64 bits and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_6(argument3);
  new result1 = _SATU_64(_ZX_64(argument2) << shift);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.263 SLUSHQS Shift Left Unsigned Saturated Half Word Quadruple by Scalar

Encoding Format ALU_HQSWRI (Section 6.12.270), exucode2 = 1101

P 31	11 30 29	1 28	1101 27 24	registerW 23 18	10 17 16	0101 15 12	unsigned6 11 6	registerZ 5 0

Syntax `slushqs` *registerW* = *registerZ*, *unsigned6*

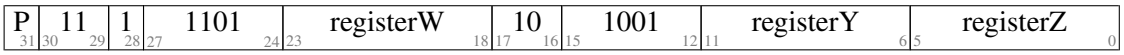
Description The *registerZ* is considered as four unsigned half words packed into 64 bits. The *registerZ* half words are shifted left by the *unsigned6* modulo 16. The resulting half words are unsigned saturated to 16 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SATU_16(_ZX_16(argument2.16[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 1101



Syntax `slushqs` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as four unsigned half words packed into 64 bits. The *registerZ* half words are shifted left by the *registerY* modulo 16. The resulting half words are unsigned saturated to 16 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SATU_16(_ZX_16(argument2.16[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.264 SLUSW Shift Left Unsigned Saturated Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1101

P	11	1	1101	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slusw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted left by the *registerY* modulo 32 then is unsigned saturated to 32 bits. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new result1 = _SATU_32(_ZX_32(argument2) << shift);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1101

P	11	1	1101	registerW	10	0011	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `slusw` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted left by the *unsigned6* modulo 32 then is unsigned saturated to 32 bits. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new result1 = _SATU_32(_ZX_32(argument2) << shift);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.265 SLUSWPS Shift Left Unsigned Saturated Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1101

P	11	1	1101	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sluswps` *registerW* = *registerZ*, *unsigned6*

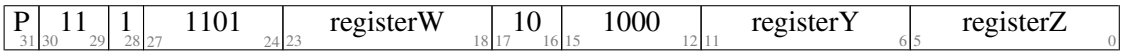
Description The *registerZ* is considered as two unsigned words packed into 64 bits. The *registerZ* words are shifted left by the *unsigned6* modulo 32. The resulting words are unsigned saturated to 32 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SATU_32(_ZX_32(argument2.32[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1101



Syntax `sluswps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two unsigned words packed into 64 bits. The *registerZ* words are shifted left by the *registerY* modulo 32. The resulting words are unsigned saturated to 32 bits, packed and stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SATU_32(_ZX_32(argument2.32[I]) << shift)
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.266 SRABOS Shift Right Arithmetic Byte Octuple by Scalar

Encoding Format ALU_BOSWRI (Section 6.12.288), exucode2 = 1010

P	11	1	1010	registerW	10	1110	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srabos` *registerW* = *registerZ*, *unsigned6*

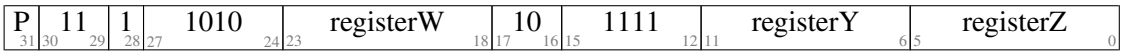
Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted right by the *unsigned6* modulo 8. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument2.8[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 1010



Syntax `srabos` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted right by the *registerY* modulo 8. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _SX_8(argument2.8[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.267 SRAD — Shift Right Arithmetic Double Word

Encoding Format ALU_DSWRI (Section 6.12.264), exucode2 = 1010

P	11	1	1010	registerW	10	0010	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srad` *registerW* = *registerZ*, *unsigned6*

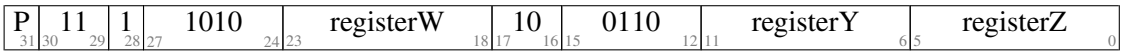
Description The *registerZ* is shifted right by the *unsigned6* modulo 64. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument2) >> _ZX_6(argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 1010



Syntax `srad` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted right by the *registerY* modulo 64. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument2) >> _ZX_6(argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.268 SRAHQS Shift Right Arithmetic Half Word Quadruple by Scalar

Encoding Format ALU_HQSWRI (Section 6.12.270), exucode2 = 1010

P	11	1	1010	registerW	10	0101	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srahqs` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted right by the *unsigned6* modulo 16. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument2.16[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 1010

P	11	1	1010	registerW	10	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srahqs` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted right by the *registerY* modulo 16. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument2.16[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.269 SRAW Shift Right Arithmetic Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1010

P	11	1	1010	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sraw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted right by the *registerY* modulo 32. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_32(argument2) >> _ZX_5(argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1010

P	11	1	1010	registerW	10	0011	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sraw` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted right by the *unsigned6* modulo 32. The leftmost bit is replicated to fill in the vacant positions, thereby extending the sign of the shifted operand. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = _SX_32(argument2) >> _ZX_5(argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.270 SRAWPS Shift Right Arithmetic Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1010

P	11	1	1010	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srawps` *registerW* = *registerZ*, *unsigned6*

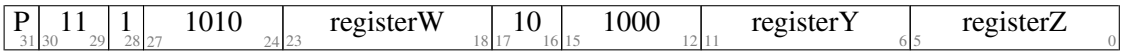
Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted right by the *unsigned6* modulo 32. The leftmost bit are replicated to fill in the vacant positions, thereby extending the signs of the shifted operands. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument2.32[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1010



Syntax `srawps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted right by the *registerY* modulo 32. The leftmost bit are replicated to fill in the vacant positions, thereby extending the signs of the shifted operands. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument2.32[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.271 SRLBOS Shift Right Logical Byte Octuple by Scalar

Encoding Format ALU_BOSWRI (Section 6.12.288), exucode2 = 1011

P	11	1	1011	registerW	10	1110	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srlbos` *registerW* = *registerZ*, *unsigned6*

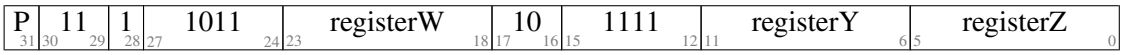
Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted right by the *unsigned6* modulo 8. The vacant positions are filled in with zeros. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _ZX_8(argument2.8[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 1011



Syntax `srlbos` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as eight bytes packed into 64 bits. The *registerZ* bytes are shifted right by the *registerY* modulo 8. The vacant positions are filled in with zeros. The resulting bytes are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = _ZX_8(argument2.8[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.272 SRLD Shift Right Logical Double Word

Encoding Format ALU_DSWRI (Section 6.12.264), exucode2 = 1011

P	11	1	1011	registerW	10	0010	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srld` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted right by the *unsigned6* modulo 64. The vacant positions are filled in with zeros. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2) >> _ZX_6(argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 1011

P	11	1	1011	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srld` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted right by the *registerY* modulo 64. The vacant positions are filled in with zeros. The result is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2) >> _ZX_6(argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.273 SRLHQS Shift Right Logical Half Word Quadruple by Scalar

Encoding Format ALU_HQSWRI (Section 6.12.270), exucode2 = 1011

P	11	1	1011	registerW	10	0101	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srlhqs` *registerW* = *registerZ*, *unsigned6*

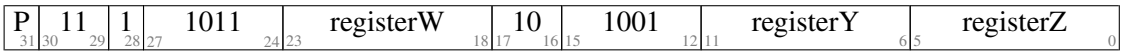
Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted right by the *unsigned6* modulo 16. The vacant positions are filled in with zeros. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ZX_16(argument2.16[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 1011



Syntax `srlhqs` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as four half words packed into 64 bits. The *registerZ* half words are shifted right by the *registerY* modulo 16. The vacant positions are filled in with zeros. The resulting half words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ZX_16(argument2.16[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.274 SRLW — Shift Right Logical Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1011

P	11	1	1011	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srlw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is shifted right by the *registerY* modulo 32. The vacant positions are filled in with zeros. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] >> _ZX_5(argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1011

P	11	1	1011	registerW	10	0011	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srlw` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is shifted right by the *unsigned6* modulo 32. The vacant positions are filled in with zeros. The result with the 32 upper bits cleared is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new result1 = argument2.32[0] >> _ZX_5(argument3);
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.275 SRLWPS Shift Right Logical Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1011

P	11	1	1011	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srlwps` *registerW* = *registerZ*, *unsigned6*

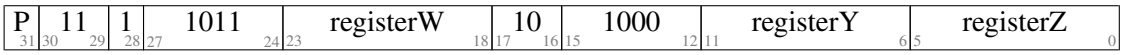
Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted right by the *unsigned6* modulo 32. The vacant positions are filled in with zeros. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(argument2.32[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1011



Syntax `srlwps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two words packed into 64 bits. The *registerZ* words are shifted right by the *registerY* modulo 32. The vacant positions are filled in with zeros. The resulting words are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(argument2.32[I]) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.276 SRSBOS Shift Right Symmetric Byte Octuple by Scalar

Encoding Format ALU_BOSWRI (Section 6.12.288), exucode2 = 1000

P	11	1	1000	registerW	10	1110	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srsbos` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is considered as eight bytes packed into 64 bits. If non-negative, each *registerZ* byte is shifted right by the *unsigned6* modulo 8. Else, each *registerZ* byte is biased by (2**(*unsigned6* modulo 8)) - 1 and shifted right by the *unsigned6* modulo 8. Each result is stored into the corresponding *registerW* byte. This instruction implements a signed division of the bytes of the *registerZ* by the power of two given by *unsigned6* modulo 8.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  new bias = (1 << shift) - 1;
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = (_SX_8(argument2.8[I]) + _SX_8(argument2.8[I]) < 0 ? bias
        : 0) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_BOSWRR (Section 6.12.289), exucode2 = 1000

P	11	1	1000	registerW	10	1111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srsbos` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as eight bytes packed into 64 bits. If non-negative, each *registerZ* byte is shifted right by the *registerY* modulo 8. Else, each *registerZ* byte is biased by (2**(*registerY* modulo 8)) - 1 and shifted right by the *registerY* modulo 8. Each result is stored into the corresponding *registerW* byte. This instruction implements a signed division of the bytes of the *registerZ* by the power of two given by *registerY* modulo 8.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_3(argument3);
  new bias = (1 << shift) - 1;
  for (I = 0; I < 8; I += 1) {
    result1.8[I] = (_SX_8(argument2.8[I]) + _SX_8(argument2.8[I]) < 0 ? bias
        : 0) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.277 SRSD Shift Right Symmetric Double Word

Encoding Format ALU_DSWRI (Section 6.12.264), exucode2 = 1000

P	11	1	1000	registerW	10	0010	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srsd` *registerW* = *registerZ*, *unsigned6*

Description If non-negative, the *registerZ* is shifted right by the *unsigned6* modulo 64. Else, the *registerZ* biased by (2**(*unsigned6* modulo 64)) - 1 and shifted right by the *unsigned6* modulo 64. The result is stored into the *registerW*. This instruction implements a signed division of the *registerZ* double word by a power of two.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_6(argument3);
  new bias = _SX_64(argument2) < 0 ? (1 << shift) - 1 : 0;
  new result1 = (_SX_64(argument2) + bias) >> shift;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DSWRR (Section 6.12.271), exucode2 = 1000

P	11	1	1000	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srsd` *registerW* = *registerZ*, *registerY*

Description If non-negative, the *registerZ* is shifted right by the *registerY* modulo 64. Else, the *registerZ* biased by (2**(*registerY* modulo 64)) - 1 and shifted right by the *registerY* modulo 64. The result is stored into the *registerW*. This instruction implements a signed division of the *registerZ* double word by a power of two.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_6(argument3);
  new bias = _SX_64(argument2) < 0 ? (1 << shift) - 1 : 0;
  new result1 = (_SX_64(argument2) + bias) >> shift;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.278 SRSHQS Shift Right Symmetric Half Word Quadruple by Scalar

Encoding Format ALU_HQSWRI (Section 6.12.270), exucode2 = 1000

P	11	1	1000	registerW	10	0101	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srshqs` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is considered as four half words packed into 64 bits. If non-negative, each *registerZ* half word is shifted right by the *unsigned6* modulo 16. Else, each *registerZ* half word is biased by (2**(*unsigned6* modulo 16)) - 1 and shifted right by the *unsigned6* modulo 16. Each result is stored into the corresponding *registerW* half word. This instruction implements a signed division of the half words of the *registerZ* by the power of two given by *unsigned6* modulo 16.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  new bias = (1 << shift) - 1;
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = (_SX_16(argument2.16[I]) + _SX_16(argument2.16[I]) < 0 ?
        bias : 0) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQSWRR (Section 6.12.276), exucode2 = 1000

P	11	1	1000	registerW	10	1001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srshqs` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as four half words packed into 64 bits. If non-negative, each *registerZ* half word is shifted right by the *registerY* modulo 16. Else, each *registerZ* half word is biased by (2**(*registerY* modulo 16)) - 1 and shifted right by the *registerY* modulo 16. Each result is stored into the corresponding *registerW* half word. This instruction implements a signed division of the half words of the *registerZ* by the power of two given by *registerY* modulo 16.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_4(argument3);
  new bias = (1 << shift) - 1;
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = (_SX_16(argument2.16[I]) + _SX_16(argument2.16[I]) < 0 ?
        bias : 0) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.279 SRSW Shift Right Symmetric Word

Encoding Format ALU_WSWRR (Section 6.12.272), exucode2 = 1000

P	11	1	1000	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srsw` *registerW* = *registerZ*, *registerY*

Description If non-negative, the *registerZ* is shifted right by the *registerY* modulo 32. Else, the *registerZ* biased by (2**(*registerY* modulo 32)) - 1 and shifted right by the *registerY* modulo 32. The result with the 32 upper bits cleared is stored into the *registerW*. This instruction implements a signed division of the *registerZ* least significant word by a power of two.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new bias = _SX_32(argument2) < 0 ? (1 << shift) - 1 : 0;
  new result1 = (_SX_32(argument2) + bias) >> shift;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WSWRI (Section 6.12.266), exucode2 = 1000

P	11	1	1000	registerW	10	0011	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srsw` *registerW* = *registerZ*, *unsigned6*

Description If non-negative, the *registerZ* is shifted right by the *unsigned6* modulo 32. Else, the *registerZ* biased by (2**(*unsigned6* modulo 32)) - 1 and shifted right by the *unsigned6* modulo 32. The result with the 32 upper bits cleared is stored into the *registerW*. This instruction implements a signed division of the *registerZ* least significant word by a power of two.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new bias = _SX_32(argument2) < 0 ? (1 << shift) - 1 : 0;
  new result1 = (_SX_32(argument2) + bias) >> shift;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

6.6.280 SRSWPS Shift Right Symmetric Word Pair by Scalar

Encoding Format ALU_WPSWRI (Section 6.12.268), exucode2 = 1000

P	11	1	1000	registerW	10	0100	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srswps` *registerW* = *registerZ*, *unsigned6*

Description The *registerZ* is considered as two words packed into 64 bits. If non-negative, each *registerZ* word is shifted right by the *unsigned6* modulo 32. Else, each *registerZ* word is biased by (2**(*unsigned6* modulo 32)) - 1 and shifted right by the *unsigned6* modulo 32. Each result is stored into the corresponding *registerW* word. This instruction implements a signed division of the *registerZ* words by the power of two given by the *unsigned6* modulo 32.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = _ZX_6(unsigned6);
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new bias = (1 << shift) - 1;
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = (_SX_32(argument2.32[I]) + _SX_32(argument2.32[I]) < 0 ?
        bias : 0) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPSWRR (Section 6.12.274), exucode2 = 1000

P	11	1	1000	registerW	10	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `srswps` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is considered as two words packed into 64 bits. If non-negative, each *registerZ* word is shifted right by the *registerY* modulo 32. Else, each *registerZ* word is biased by (2**(*registerY* modulo 32)) - 1 and shifted right by the *registerY* modulo 32. Each result is stored into the corresponding *registerW* word. This instruction implements a signed division of the *registerZ* words by the power of two given by the *registerY* modulo 32.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new shift = _ZX_5(argument3);
  new bias = (1 << shift) - 1;
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = (_SX_32(argument2.32[I]) + _SX_32(argument2.32[I]) < 0 ?
        bias : 0) >> shift
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.281 STSUD Subtract, Test and Shift Unsigned Double Words

Encoding Format ALU_DWRRS (Section 6.12.242), exucode2 = 1110

P	11	1	1110	registerW	01	1010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `stsud` *registerW* = *registerZ*, *registerY*

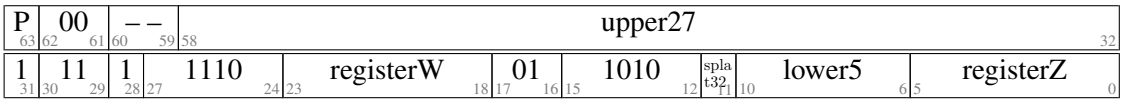
Description The *registerZ* is subtracted from the *registerY* If positive, the result is shifted left one bit and OR-ed with 0x1. Else, the *registerY* is shifted left one bit. The result is stored into the *registerW*. This instruction is provided to support double word unsigned integer division.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument3) >= _ZX_64(argument2) ? ((_ZX_64(argument3)
      - _ZX_64(argument2)) << 1) | 1 : _ZX_64(argument3) << 1;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_DWRRS.M (Section 6.12.243), exucode2 = 1110



Syntax `stsud` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* is subtracted from the *upper27_lower5* If positive, the result is shifted left one bit and OR-ed with 0x1. Else, the *upper27_lower5* is shifted left one bit. The result is stored into the *registerW*. This instruction is provided to support double word unsigned integer division.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument3) >= _ZX_64(argument2) ? ((_ZX_64(argument3)
      - _ZX_64(argument2)) << 1) | 1 : _ZX_64(argument3) << 1;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.282 STSUHQ Subtract, Test and Shift Unsigned Half Word Quadruple

Encoding Format ALU_HQWRRS (Section 6.12.254), exucode2 = 1110

P	11	1	1110	registerW	01	1111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `stsuhq` *registerW* = *registerZ*, *registerY*

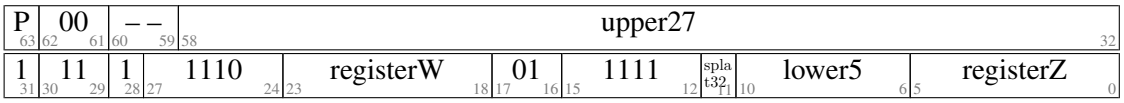
Description The *registerZ* interpreted as a half word quadruple is subtracted from the *registerY* interpreted as a half word quadruple . For each result, if the difference is positive it is shifted left one bit and OR-ed with 0x1. Else, the result is the corresponding half word of the *registerY* is shifted left one bit. The four resulting half word are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ZX_16(argument3.16[I]) >= _ZX_16(argument2.16[I]) ? ((
        _ZX_16(argument3.16[I]) - _ZX_16(argument2.16[I])) << 1) | 1 : _ZX_16(
        argument3.16[I]) << 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQWRRS.M (Section 6.12.255), exucode2 = 1110



Syntax `stsuhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* interpreted as a half word quadruple is subtracted from the *upper27_lower5* interpreted as a half word quadruple . For each result, if the difference is positive it is shifted left one bit and OR-ed with 0x1. Else, the result is the corresponding half word of the *upper27_lower5* is shifted left one bit. The four resulting half word are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _ZX_16(argument3.16[I]) >= _ZX_16(argument2.16[I]) ? ((
        _ZX_16(argument3.16[I]) - _ZX_16(argument2.16[I])) << 1) | 1 : _ZX_16(
        argument3.16[I]) << 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.283 STSUW Subtract, Test and Shift Unsigned Word

Encoding Format ALU_WRRSW (Section 6.12.246), exucode2 = 1110

P	11	1	1110	registerW	01	1011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `stsuw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY*. If positive, the result is shifted left one bit and OR-ed with 0x1. Else, the *registerY* is shifted left one bit. The result with the 32 upper bits cleared is stored into the *registerW*. This instruction is provided to support unsigned integer division.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = argument3.32[0] >= argument2.32[0] ? ((argument3.32[0] -
      argument2.32[0]) << 1) | 1 : argument3.32[0] << 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

Encoding Format ALU_WRRSW.W (Section 6.12.247), exucode2 = 1110

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1110	registerW	01	1011	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `stsuw` *registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is subtracted from the *upper27_lower5*. If positive, the result is shifted left one bit and OR-ed with 0x1. Else, the *upper27_lower5* is shifted left one bit. The result with the 32 upper bits cleared is stored into the *registerW*. This instruction is provided to support unsigned integer division.

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = argument3.32[0] >= argument2.32[0] ? ((argument3.32[0] -
      argument2.32[0]) << 1) | 1 : argument3.32[0] << 1;
stage E1:
  GPR[registerW] = _ZX_32(result1);
```


Kalray – all rights reserved

©2025 Kalray

6.6.284 STSUWP — Subtract, Test and Shift Unsigned Word Pair

Encoding Format ALU_WPWRRS (Section 6.12.250), exucode2 = 1110

P	11	1	1110	registerW	01	1110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `stsuwp` *registerW* = *registerZ*, *registerY*

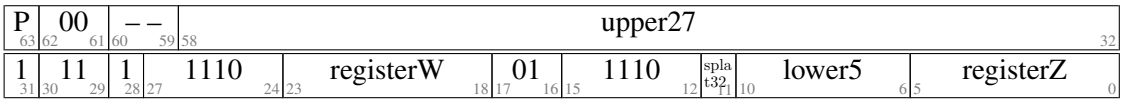
Description The *registerZ* interpreted as a word pair is subtracted from the *registerY* interpreted as a word pair . For each result, if the difference is positive it is shifted left one bit and OR-ed with 0x1. Else, the result is the corresponding word of the *registerY* is shifted left one bit. The two resulting word are packed and stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(argument3.32[I]) >= _ZX_32(argument2.32[I]) ? ((
        _ZX_32(argument3.32[I]) - _ZX_32(argument2.32[I])) << 1) | 1 : _ZX_32(
        argument3.32[I]) << 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPWRRS.M (Section 6.12.251), exucode2 = 1110



Syntax `stsuwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *registerZ* interpreted as a word pair is subtracted from the *upper27_lower5* interpreted as a word pair . For each result, if the difference is positive it is shifted left one bit and OR-ed with 0x1. Else, the result is the corresponding word of the *upper27_lower5* is shifted left one bit. The two resulting word are packed and stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _ZX_32(argument3.32[I]) >= _ZX_32(argument2.32[I]) ? ((
        _ZX_32(argument3.32[I]) - _ZX_32(argument2.32[I])) << 1) | 1 : _ZX_32(
        argument3.32[I]) << 1
  };
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.285 SXLBHQ Sign Extend Least Significant Bytes to Half Word Quadruple

Encoding Format ALU_LWRHQ (Section 6.12.269), exucode2 = 0110

P	11	1	0110	registerW	10	0101	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sxlbhq` *registerW* = *registerZ*

Description The *registerZ* is considered as four bytes packed into the least significant 32 bits. Each byte is sign-extended to 16-bit and stored into one of the half words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.16[0] = _SX_8(argument2.8[0]);
  result1.16[1] = _SX_8(argument2.8[1]);
  result1.16[2] = _SX_8(argument2.8[2]);
  result1.16[3] = _SX_8(argument2.8[3]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.286 SXLHWP Sign Extend Least Significant Half Words to Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0110

P	11	1	0110	registerW	10	0100	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sxlhwp` *registerW* = *registerZ*

Description The *registerZ* is considered as two half words packed into the least significant 32 bits. Each half word is sign-extended to 32 bits and stored into one of the words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.32[0] = _SX_16(argument2.16[0]);
  result1.32[1] = _SX_16(argument2.16[1]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.287 SXMBHQ Sign Extend Most Significant Bytes to Half Word Quadruple

Encoding Format ALU_LWRHQ (Section 6.12.269), exucode2 = 0111

P	11	1	0111	registerW	10	0101	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sxmbhq` *registerW* = *registerZ*

Description The *registerZ* is considered as four bytes packed into the most significant 32 bits. Each byte is sign-extended to 16 bits and stored into one of the half words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.16[0] = _SX_8(argument2.8[4]);
  result1.16[1] = _SX_8(argument2.8[5]);
  result1.16[2] = _SX_8(argument2.8[6]);
  result1.16[3] = _SX_8(argument2.8[7]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.288 SXMHWP Sign Extend Most Significant Half Words to Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0111

P	11	1	0111	registerW	10	0100	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `sxmhwp` *registerW* = *registerZ*

Description The *registerZ* is considered as two half words packed into the most significant 32 bits. Each half word is sign-extended to 32 bits and stored into one of the words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.32[0] = _SX_16(argument2.16[2]);
  result1.32[1] = _SX_16(argument2.16[3]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.289 XMOVETD Move To Extension Double Word

Encoding Format ALU_XMTD0 (Section 6.12.312), exucode5 = 1110

P 31	11 30 29	1 28	00 27 26	00 25 24	registerAx 23 18	11 17 16	1110 15 12	------ 11 6	registerZ 5 0

Syntax `xmovetd` *registerAx* = *registerZ*

Description The *registerAx* operand receives the *registerZ* value.

Execution Reservation table ALU_TINY_CRWL (Section 6.14.17)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2);
stage E1:
  CS.XMF = 1;
  XCR[registerAx<<2] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_XMTD1 (Section 6.12.313), exucode5 = 1110

Syntax `xmovetd` *registerAy* = *registerZ*

Description The *registerAy* operand receives the *registerZ* value.

Execution Reservation table ALU_TINY_CRWL (Section 6.14.17)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2);
stage E1:
  CS.XMF = 1;
  XCR[(registerAy<<2)+1] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_XMTD2 (Section 6.12.314), exucode5 = 1110

Syntax `xmovetd` *registerAz* = *registerZ*

Description The *registerAz* operand receives the *registerZ* value.

Execution Reservation table ALU_TINY_CRWH (Section 6.14.16)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2);
stage E1:
  CS.XMF = 1;
  XCR[(registerAz<<2)+2] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_XMTD3 (Section 6.12.315), exucode5 = 1110

P	11	1	00	11	registerAt	11	1110	------	registerZ
31	30 29	28	27 26	25 24	23 18	17 16	15 12	11 6	5 0

Syntax `xmovetd` *registerAt* = *registerZ*

Description The *registerAt* operand receives the *registerZ* value.

Execution Reservation table ALU_TINY_CRWH (Section 6.14.16)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2);
stage E1:
  CS.XMF = 1;
  XCR[(registerAt<<2)+3] = result1;
```


Kalray – all rights reserved

6.6.290 XMOVETQ Move To Extension Quadruple Word

Encoding Format ALU_XMTQ0 (Section 6.12.316), exucode5 = 1110

P	11	1	010	0	registerAE	11	1110	registerY	registerZ
31	30 29	28	27 25	24	23 18	17 16	15 12	11 6	5 0

Syntax `xmovetq` *registerAE* = *registerZ*, *registerY*

Description The *registerAE* operand receives the concatenation of the *registerZ* value and the *registerY* value.

Execution Reservation table ALU_TINY_CRWL (Section 6.14.17)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = 0;
  result1.64[0] = argument2;
  result1.64[1] = argument3;
stage E1:
  CS.XMF = 1;
  XBR[registerAE<<1] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_XMTQ1 (Section 6.12.317), exucode5 = 1110

P	11	1	010	1	registerAO	11	1110	registerY	registerZ
31	30 29	28	27 25	24	23 18	17 16	15 12	11 6	5 0

Syntax `xmovetq` *registerAO* = *registerZ*, *registerY*

Description The *registerAO* operand receives the concatenation of the *registerZ* value and the *registerY* value.

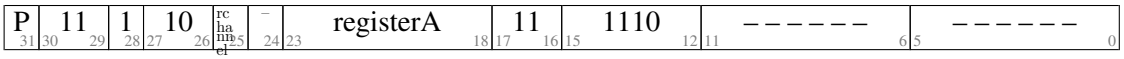
Execution Reservation table ALU_TINY_CRWH (Section 6.14.16)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = 0;
  result1.64[0] = argument2;
  result1.64[1] = argument3;
stage E1:
  CS.XMF = 1;
  XBR[(registerAO<<1)+1] = result1;
```


Kalray – all rights reserved

6.6.291 XRECVO Receive Extension Octuple Word from Channel

Encoding Format ALU_CRC (Section 6.12.319), exucode8 = 10



Syntax `xrecvo`*rchannel registerA*

Description The *registerA* operand is received from the channel designated by the *rchannel*.

Modifier(s)

- rchannel (cf. channel in section 6.10.2 on page 1415)

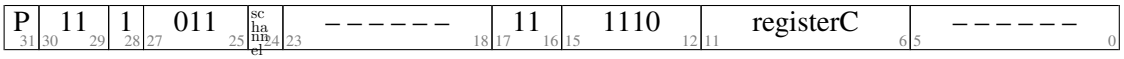
Execution Reservation table ALU_TINY_CRWL_CRWH (Section 6.14.18)

```
stage ID:
  new exchange = IXC.SFB;
  new rchannel = _ZX_1(rchannel);
stage RR:
  if (!comm_owner()) {
    _THROW(PRIVILEGE);
  } else {
    new result = recvv(rchannel ^ exchange);
  }
stage E1:
  CS.XMF = 1;
stage E2:
  XVR[registerA] = result;
```


Kalray – all rights reserved

6.6.292 XSENDO — Send Extension Octuple Word to Channel

Encoding Format ALU_CSC (Section 6.12.318), exucode9 = 011



Syntax `xsendo`*schannel registerC*

Description The *registerC* operand is sent to the channel designated by the *schannel*.

Modifier(s)

- schannel (cf. channel in section 6.10.2 on page 1415)

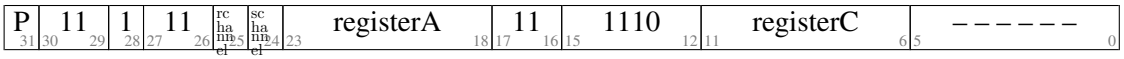
Execution Reservation table ALU_TINY_CRRP (Section 6.14.14)

```
stage ID:
  new exchange = IXC.SFB;
  new schannel = _ZX_1(schannel);
stage RR:
  new argument = XVR[registerC];
stage E1:
  CS.XMF = 1;
stage E4:
  if (!comm_owner()) {
    _THROW(PRIVILEGE);
  } else {
    sendv(argument, schannel ^ exchange);
  }
```


Kalray – all rights reserved

6.6.293 XSENDRECVO Send and Receive Octuple Word to/from Channels

Encoding Format ALU_CSRC (Section 6.12.320), exucode8 = 11



Syntax `xsendrecvo`*schannelrchannel registerC*, *registerA*

Description The *registerC* operand is sent to the channel designated by the *schannel*. The *registerA* operand is received from the channel designated by the *rchannel*.

Modifier(s)

- rchannel (cf. channel in section 6.10.2 on page 1415)
- schannel (cf. channel in section 6.10.2 on page 1415)

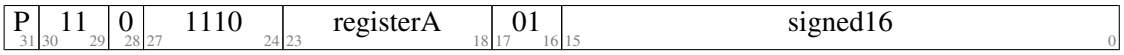
Execution Reservation table ALU_TINY_CRRP_CRWL_CRWH (Section 6.14.15)

```
stage ID:
  new exchange = IXC.SFB;
  new rchannel = _ZX_1(rchannel);
  new schannel = _ZX_1(schannel);
stage RR:
  new argument = XVR[registerC];
  if (!comm_owner()) {
    _THROW(PRIVILEGE);
  } else {
    new result = sendrecvv(argument, schannel ^ exchange, rchannel ^ exchange
        );
  }
stage E1:
  CS.XMF = 1;
stage E2:
  XVR[registerA] = result;
```


Kalray – all rights reserved

6.6.294 XSPLATDO — Move To Extension Double Word

Encoding Format ALU_XSDOI (Section 6.12.206), exucode2 = 1110



Syntax `xsplatdo` *registerA* = *signed16*

Description The *registerA* operand receives the *signed16* value replicated 4 times.

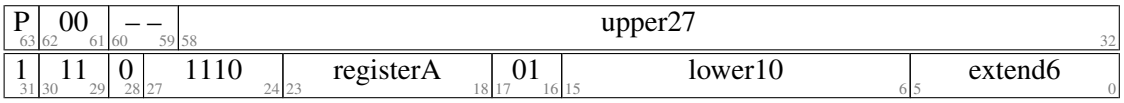
Execution Reservation table ALU_TINY_CRWL_CRWH (Section 6.14.18)

```
stage RR:
  new argument2 = _SX_16(signed16);
  new result1 = 0;
  result1.64[0] = argument2;
  result1.64[1] = argument2;
  result1.64[2] = argument2;
  result1.64[3] = argument2;
stage E1:
  CS.XMF = 1;
  XVR[registerA] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_XSDOI.X (Section 6.12.207), exucode2 = 1110



Syntax `xsplatdo` *registerA* = *extend6_upper27_lower10*

Description The *registerA* operand receives the *extend6_upper27_lower10* value replicated 4 times.

Execution Reservation table ALU_TINY_CRWL_CRWH.X (Section 6.14.19)

```
stage RR:
  new argument2 = _SX_43(extend6_upper27_lower10);
  new result1 = 0;
  result1.64[0] = argument2;
  result1.64[1] = argument2;
  result1.64[2] = argument2;
  result1.64[3] = argument2;
stage E1:
  CS.XMF = 1;
  XVR[registerA] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_XSDOI.Y (Section 6.12.208), exucode2 = 1110

P 95	00 94 93	– – 92 91	extend27 90 64												
1 63	00 62 61	– – 60 59	upper27 58 32												
1 31	11 30 29	0 28	1110 27 24	registerA 23 18				01 17 16	lower10 15 6					– – – – – – 5 0	

Syntax `xsplatdo` *registerA* = *extend27_upper27_lower10*

Description The *registerA* operand receives the *extend27_upper27_lower10* value replicated 4 times.

Execution Reservation table ALU_TINY_CRWL_CRWH.Y (Section 6.14.20)

```
stage RR:
  new argument2 = _WX_64(extend27_upper27_lower10);
  new result1 = 0;
  result1.64[0] = argument2;
  result1.64[1] = argument2;
  result1.64[2] = argument2;
  result1.64[3] = argument2;
stage E1:
  CS.XMF = 1;
  XVR[registerA] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.6.295 ZXLBHQ Zero Extend Least Significant Bytes to Half Word Quadruple

Encoding Format ALU_LWRHQ (Section 6.12.269), exucode2 = 0100

P	11	1	0100	registerW	10	0101	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `zxlbhq` *registerW* = *registerZ*

Description The *registerZ* is considered as four bytes packed into the least significant 32 bits. Each byte is zero-extended to 16-bit and stored into one of the half words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.16[0] = _ZX_8(argument2.8[0]);
  result1.16[1] = _ZX_8(argument2.8[1]);
  result1.16[2] = _ZX_8(argument2.8[2]);
  result1.16[3] = _ZX_8(argument2.8[3]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.296 ZXLHWP Zero Extend Least Significant Half Words to Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0100

P	11	1	0100	registerW	10	0100	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `zxlhwp` *registerW* = *registerZ*

Description The *registerZ* is considered as two half words packed into the least significant 32 bits. Each half word is zero-extended to 32 bits and stored into one of the words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.32[0] = _ZX_16(argument2.16[0]);
  result1.32[1] = _ZX_16(argument2.16[1]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.297 ZXMBHQ Zero Extend Most Significant Bytes to Half Word Quadruple

Encoding Format ALU_LWRHQ (Section 6.12.269), exucode2 = 0101

P	11	1	0101	registerW	10	0101	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `zxmbhq` *registerW* = *registerZ*

Description The *registerZ* is considered as four bytes packed into the most significant 32 bits. Each byte is zero-extended to 16 bits and stored into one of the half words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.16[0] = _ZX_8(argument2.8[4]);
  result1.16[1] = _ZX_8(argument2.8[5]);
  result1.16[2] = _ZX_8(argument2.8[6]);
  result1.16[3] = _ZX_8(argument2.8[7]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.6.298 ZXMHWP Zero Extend Most Significant Half Words to Word Pair

Encoding Format ALU_BWRWP (Section 6.12.267), exucode2 = 0101

P	11	1	0101	registerW	10	0100	------	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `zxmhwp` *registerW* = *registerZ*

Description The *registerZ* is considered as two half words packed into the most significant 32 bits. Each half word is zero-extended to 32 bits and stored into one of the words of the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  result1.32[0] = _ZX_16(argument2.16[2]);
  result1.32[1] = _ZX_16(argument2.16[3]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7 MAU Instructions

6.7.1 CMULDT Carryless Multiply Double Words to Tetra Word

Encoding Format MAU_QDD1 (Section 6.12.357), exucode2 = 1011

P	10	1	1011	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `cmuldt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming double integer is carryless multiplied by the *registerZ* assuming double integer. The 128 bits of the result are stored into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = clm_64_128(argument2, argument3);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.2 CMULGHXDT Carryless Multiply Galois High and Xor Double Words to Tetra Word

Encoding Format MAU_QQDD2 (Section 6.12.358), exucode2 = 1111

P	10	1	1111	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `cmulghxdt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming double integer is reflected and then carryless multiplied by the reflection of the *registerZ* assuming double integer. The 128 bits results is left-shifted by 128 and 64, shift results are XOR-ed together and then reduced modulo the GCM polynomial. The 128 bits of the result are reflected and XOR-ed with *registerM* and then stored into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 ^ gcm_tt_64_128(argument2, argument3);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.3 CMULGLXDT Carryless Multiply Galois Low and Xor Double Words to Tetra Word

Encoding Format MAU_QQDD2 (Section 6.12.358), exucode2 = 1101

P	10	1	1101	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `cmulglxdt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming double integer is reflected and then carryless multiplied by the reflection of the *registerZ* assuming double integer. The 128 bits results is XOR-ed to itself left-shifted by 64 and then reduced modulo the GCM polynomial. The 128 bits of the result are reflected and XOR-ed with *registerM* and then stored into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 ^ gcm_bb_64_128(argument2, argument3);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.4 CMULGMXDT Carryless Multiply Galois Middle and Xor Double Words to Tetra Word

Encoding Format MAU_QQDD2 (Section 6.12.358), exucode2 = 1110

P	10	1	1110	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `cmulgmxdt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming double integer is reflected and then carryless multiplied by the reflection of the *registerZ* assuming double integer. The 128 bits results is left-shifted by 64 and then reduced modulo the GCM polynomial. The 128 bits of the result are reflected and XOR-ed with *registerM* and then stored into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 ^ gcm_bt_64_128(argument2, argument3);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.5 CMULXDT Carryless Multiply and Xor Double Words to Tetra Word

Encoding Format MAU_QQDD2 (Section 6.12.358), exucode2 = 1100

P	10	1	1100	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `cmulxdt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming double integer is carryless multiplied by the *registerZ* assuming double integer. The final 128 bits of the result are XOR-ed with *registerM* and then stored into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 ^ clm_64_128(argument2, argument3);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.6 COPYQ — Copy Quadruple Word

Encoding Format MAU_COPYQ (Section 6.12.371), exucode2 = 1111

P	10	1	1111	registerM	–	00	1111	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `copyq` *registerM* = *registerZ*, *registerY*

Description The *registerM* operand receives the concatenation of the *registerZ* and *registerY* values.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument2) | (argument3 << 64);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.7 CRCBELLW Cyclic Redundancy Check Big-Endian Least Significant with Least Significant Words

Encoding Format MAU_CRC (Section 6.12.361), exucode2 = 1001

P	10	1	1001	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `crcbellw` *registerW* = *registerZ*, *registerY*

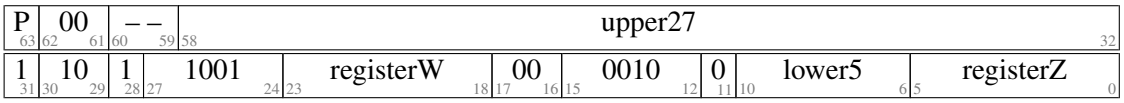
Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 least significant bytes of the *registerZ* considered in reverse order. The result is reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *registerY*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ _SWAP_16(_SWAP_8(argument2));
  new result1 = crc32_be_u32(argument1_data.32[0], argument3.32[0]);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_CRC.W (Section 6.12.362), exucode2 = 1001



Syntax `crcbellw` *registerW* = *registerZ*, *upper27_lower5*

Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 least significant bytes of the *registerZ* considered in reverse order. The result is reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *upper27_lower5*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ _SWAP_16(_SWAP_8(argument2));
  new result1 = crc32_be_u32(argument1_data.32[0], argument3.32[0]);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.8 CRCBELMW Cyclic Redundancy Check Big-Endian Least Significant with Most Significant Words

Encoding Format MAU_CRC (Section 6.12.361), exucode2 = 1000

P	10	1	1000	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `crcbelmw` *registerW* = *registerZ*, *registerY*

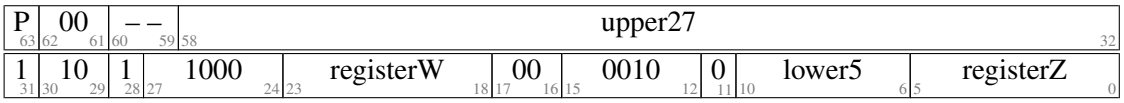
Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 most significant bytes of the *registerZ* considered in reverse order. The result is reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *registerY*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ _SWAP_16(_SWAP_8(argument2 >> 32));
  new result1 = crc32_be_u32(argument1_data.32[0], argument3.32[0]);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_CRC.W (Section 6.12.362), exucode2 = 1000



Syntax `crcbelmw` *registerW* = *registerZ*, *upper27_lower5*

Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 most significant bytes of the *registerZ* considered in reverse order. The result is reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *upper27_lower5*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ _SWAP_16(_SWAP_8(argument2 >> 32));
  new result1 = crc32_be_u32(argument1_data.32[0], argument3.32[0]);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.9 CRCLELLW Cyclic Redundancy Check Little-Endian Least Significant with Least Significant Words

Encoding Format MAU_CRC (Section 6.12.361), exucode2 = 1011

P	10	1	1011	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `crclellw` *registerW* = *registerZ*, *registerY*

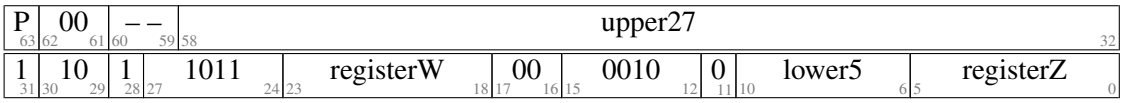
Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 least significant bytes of the *registerZ*. The result is bit reversed and then reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *registerY* bit reversed beforehand. The result is bit reversed and stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ argument2.32[0];
  new result1 = reflect_32(crc32_be_u32(reflect_32(argument1_data.32[0]),
      reflect_32(argument3.32[0])));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_CRC.W (Section 6.12.362), exucode2 = 1011



Syntax `crclellw` *registerW* = *registerZ*, *upper27_lower5*

Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 least significant bytes of the *registerZ*. The result is bit reversed and then reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *upper27_lower5* bit reversed beforehand. The result is bit reversed and stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ argument2.32[0];
  new result1 = reflect_32(crc32_be_u32(reflect_32(argument1_data.32[0]),
      reflect_32(argument3.32[0])));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.10 CRCLELMW Cyclic Redundancy Check Little-Endian Least Significant with Most Significant Words

Encoding Format MAU_CRC (Section 6.12.361), exucode2 = 1010

P	10	1	1010	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `crclelmw` *registerW* = *registerZ*, *registerY*

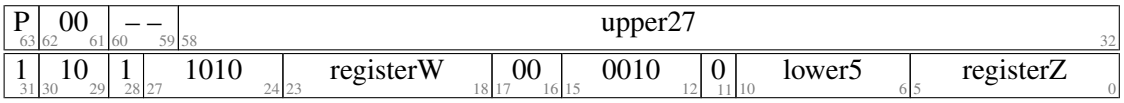
Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 most significant bytes of the *registerZ*. The result is bit reversed and then reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *registerY* bit reversed beforehand. The result is bit reversed and stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ argument2.32[1];
  new result1 = reflect_32(crc32_be_u32(reflect_32(argument1_data.32[0]),
      reflect_32(argument3.32[0])));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_CRC.W (Section 6.12.362), exucode2 = 1010



Syntax `crclelmw` *registerW* = *registerZ*, *upper27_lower5*

Description The 4 least significant bytes of the *registerW* are XOR-ed with the 4 most significant bytes of the *registerZ*. The result is bit reversed and then reduced modulo the 4-byte polynomial stored in the 4 least significant bytes of the *upper27_lower5* bit reversed beforehand. The result is bit reversed and stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new argument1_data = argument1.32[0] ^ argument2.32[1];
  new result1 = reflect_32(crc32_be_u32(reflect_32(argument1_data.32[0]),
      reflect_32(argument3.32[0])));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.11 DOT2SUWD Dot product of 2x Signed by Unsigned Word Vectors into Double Word

Encoding Format MAU_DPP (Section 6.12.363), exucode2 = 1110

P	10	1	1110	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `dot2suwd` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The dot-product of the *registerY* word pair and the *registerZ* word pair is computed, assuming unsigned and signed integers. The resulting double word is stored back into the *registerW*.

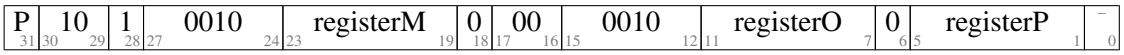
Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_32(argument3.32[0]) * _SX_32(argument2.32[0])) + (_ZX_32
      (argument3.32[1]) * _SX_32(argument2.32[1]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.12 DOT2SUWDP Dot products of 2x Signed by Unsigned Word Vectors into Double Word Pair

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0010



Syntax `dot2suwdp` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered each of four words packed into 128 bits. The dot-product of the *registerO* upper word pair and the *registerP* upper word pair is computed, assuming unsigned and signed integers. The dot-product of the *registerO* lower word pair and the *registerP* lower word pair is computed, assuming unsigned and signed integers. The resulting pair of double words is packed and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  result1.64[0] = (_ZX_32(argument3.32[0]) * _SX_32(argument2.32[0])) + (
      _ZX_32(argument3.32[1]) * _SX_32(argument2.32[1]));
  result1.64[1] = (_ZX_32(argument3.32[2]) * _SX_32(argument2.32[2])) + (
      _ZX_32(argument3.32[3]) * _SX_32(argument2.32[3]));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.13 DOT2UWD Dot product of 2x Unsigned Word Vectors into Double Word

Encoding Format MAU_DPP (Section 6.12.363), exucode2 = 1101

P	10	1	1101	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `dot2uwd` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The dot-product of the *registerY* word pair and the *registerZ* word pair is computed, assuming unsigned integers. The resulting double word is stored back into the *registerW*.

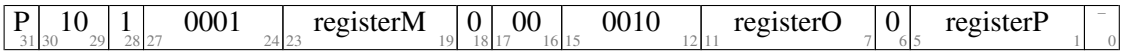
Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (_ZX_32(argument3.32[0]) * _ZX_32(argument2.32[0])) + (_ZX_32
      (argument3.32[1]) * _ZX_32(argument2.32[1]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.14 DOT2UWDP Dot products of 2x Unsigned Word Vectors into Double Word Pair

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0001



Syntax `dot2uwdp` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered each of four words packed into 128 bits. The dot-product of the *registerO* upper word pair and the *registerP* upper word pair is computed, assuming unsigned integers. The dot-product of the *registerO* lower word pair and the *registerP* lower word pair is computed, assuming unsigned integers. The resulting pair of double words is packed and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  result1.64[0] = (_ZX_32(argument3.32[0]) * _ZX_32(argument2.32[0])) + (
      _ZX_32(argument3.32[1]) * _ZX_32(argument2.32[1]));
  result1.64[1] = (_ZX_32(argument3.32[2]) * _ZX_32(argument2.32[2])) + (
      _ZX_32(argument3.32[3]) * _ZX_32(argument2.32[3]));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.15 DOT2W — Dot product of 2x Word Vectors into Word

Encoding Format MAU_DPP (Section 6.12.363), exucode2 = 1111

P	10	1	1111	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `dot2w` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The dot-product of the *registerY* word pair and the *registerZ* word pair is computed, assuming integers. The result is zero-extended and stored back into the *registerW*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32((argument3.32[0] * argument2.32[0]) + (argument3.32[1]
       * argument2.32[1]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.16 DOT2WD Dot product of 2x Word Vectors into Double Word

Encoding Format MAU_DPP (Section 6.12.363), exucode2 = 1100

P	10	1	1100	registerW	00	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `dot2wd` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The dot-product of the *registerY* word pair and the *registerZ* word pair is computed, assuming signed integers. The resulting double word is stored back into the *registerW*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = (_SX_32(argument3.32[0]) * _SX_32(argument2.32[0])) + (_SX_32
      (argument3.32[1]) * _SX_32(argument2.32[1]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.17 DOT2WDP Dot products of 2x Word Vectors into Double Word Pair

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0000

P	10	1	0000	registerM	0	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `dot2wdp` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered each of four words packed into 128 bits. The dot-product of the *registerO* upper word pair and the *registerP* upper word pair is computed, assuming signed integers. The dot-product of the *registerO* lower word pair and the *registerP* lower word pair is computed, assuming signed integers. The resulting pair of double words is packed and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  result1.64[0] = (_SX_32(argument3.32[0]) * _SX_32(argument2.32[0])) + (
      _SX_32(argument3.32[1]) * _SX_32(argument2.32[1]));
  result1.64[1] = (_SX_32(argument3.32[2]) * _SX_32(argument2.32[2])) + (
      _SX_32(argument3.32[3]) * _SX_32(argument2.32[3]));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.18 DOT2WZP Dot products of 2x Word Vectors into Word Zero Extended Pair

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0011

P	10	1	0011	registerM	0	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `dot2wzp` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered each of four words packed into 128 bits. The dot-product of the *registerO* upper word pair and the *registerP* upper word pair is computed, assuming integers. The dot-product of the *registerO* lower word pair and the *registerP* lower word pair is computed, assuming integers. The resulting pair of words are zero-extended to 64-bit, packed and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  result1.64[0] = _ZX_32((argument3.32[0] * argument2.32[0]) + (argument3
      .32[1] * argument2.32[1]));
  result1.64[1] = _ZX_32((argument3.32[2] * argument2.32[2]) + (argument3
      .32[3] * argument2.32[3]));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.19 MADDD Multiply-Add Double Word

Encoding Format MAU_DDDD0 (Section 6.12.348), exucode2 = 0000

P	10	1	0000	registerW	00	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maddd` *registerW* = *registerZ*, *registerY*

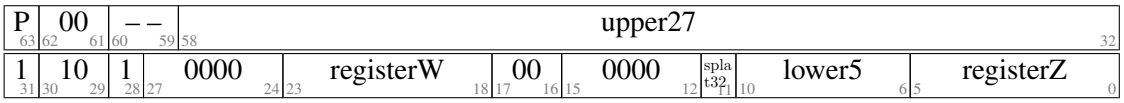
Description The *registerY* is multiplied by the *registerZ*, assuming 64-bit integers, and the product is added to the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_SX_64(argument3) * _SX_64(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDDD0.M (Section 6.12.349), exucode2 = 0000



Syntax `maddd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming 64-bit integers, and the product is added to the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_SX_64(argument3) * _SX_64(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.20 MADDDT — Multiply-Add Double to Tetra Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1000

P	10	1	1000	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `madddt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming 64-bit integers, and the product is added to the 128 bits of the *registerM*. The result is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 + (_SX_64(argument3) * _SX_64(argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.21 MADDHQ — Multiply-Add Half Word Quadruple

Encoding Format MAU_DDDD0 (Section 6.12.348), exucode2 = 0010

P	10	1	0010	registerW	00	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maddhq` *registerW* = *registerZ*, *registerY*

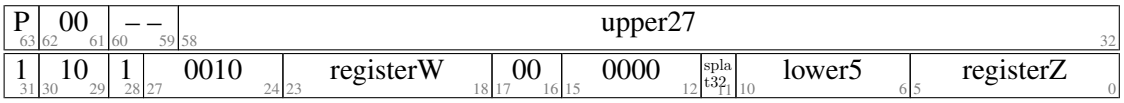
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting half words are added to the respective half words of the *registerW*. The resulting half words are packed and stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument1.16[I] + (_SX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDDD0.M (Section 6.12.349), exucode2 = 0010



Syntax `maddhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting half words are added to the respective half words of the *registerW*. The resulting half words are packed and stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument1.16[I] + (_SX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.22 MADDHWQ Multiply-Add Half Word to Word Quadruple

Encoding Format MAU_QQDD3 (Section 6.12.368), exucode2 = 0000

P	10	1	0000	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `maddhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting words are added to the respective words of the *registerM*. The resulting words are packed and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] + (_SX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.23 MADDMWQ Multiply-Add Most Significant Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0001

P	10	1	0001	registerM	1	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `maddmwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming signed integers. The four most significant words of the 64-bit results are added to the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] + ((_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I])) >> 32)
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.24 MADDSUDT Multiply-Add Signed by Unsigned Double to Tetra Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1010

P	10	1	1010	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `maddsudt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming unsigned 64-bit integer and the *registerZ* assuming signed 64-bit integer are multiplied. The product is added to the 128 bits of the *registerM* and the result is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 + (_ZX_64(argument3) * _SX_64(argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.25 MADDSUHWQMultiply-Add Signed by Unsigned Half Word to Word Quadruple

Encoding Format MAU_QQDD3 (Section 6.12.368), exucode2 = 0010

P	10	1	0010	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `maddsuhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words assuming unsigned integers and the *registerZ* half words assuming signed integers are multiplied. The four resulting words are added to the respective words of the *registerM*. The resulting words are packed and stored back into the *registerM*.

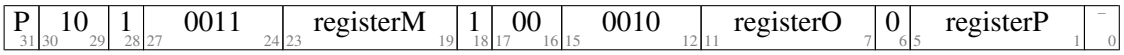
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] + (_ZX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.26 MADDSUMWQ Multiply-Add Signed by Unsigned Most Significant Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0011



Syntax `maddsumwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words assuming unsigned integers and the *registerP* words assuming signed integers are multiplied. The four most significant words of the 64-bit results are added to the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] + ((_ZX_32(argument3.32[I]) * _SX_32(
        argument2.32[I])) >> 32)
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.27 MADDSUWD Multiply-Add Signed by Unsigned Words to Double Word

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0010

P	10	1	0010	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maddsuwd` *registerW* = *registerZ*, *registerY*

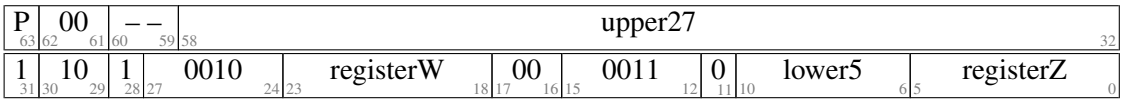
Description The *registerY* assuming unsigned integer and the *registerZ* assuming signed integer are multiplied. The product is added to the 64 bits of the *registerW* and the result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_ZX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0010



Syntax `maddsuwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* assuming unsigned integer and the *registerZ* assuming signed integer are multiplied. The product is added to the 64 bits of the *registerW* and the result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_ZX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.28 MADDSUWDP Multiply-Add Signed by Unsigned Words to Double Word Pair

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1010

P	10	1	1010	registerM	–	00	0000	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `maddsuwdp` *registerM* = *registerZ*, *registerY*

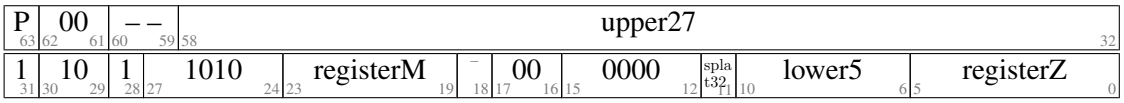
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words assuming unsigned integer and the *registerZ* words assuming signed integer are multiplied. The product pair is added to the 64-bit pair of the *registerM* and the resulting pair is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] + (_ZX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1010



Syntax `maddsuwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words assuming unsigned integer and the *registerZ* words assuming signed integer are multiplied. The product pair is added to the 64-bit pair of the *registerM* and the resulting pair is stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] + (_ZX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.29 MADDUDT Multiply-Add Unsigned Double to Tetra Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1001

P	10	1	1001	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `maddudt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming unsigned 64-bit integers, and the product is added to the 128 bits of the *registerM*. The result is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 + (_ZX_64(argument3) * _ZX_64(argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.30 MADDUHWQ Multiply-Add Unsigned Half Word to Word Quadruple

Encoding Format MAU_QQDD3 (Section 6.12.368), exucode2 = 0001

P	10	1	0001	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `madduhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming unsigned integers. The four resulting words are added to the respective words of the *registerM*. The resulting words are packed and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] + (_ZX_16(argument3.16[I]) * _ZX_16(
        argument2.16[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.31 MADDUMWQ Multiply-Add Unsigned Most Significant Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0010

P	10	1	0010	registerM	1	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `maddumwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming unsigned integers. The four most significant words of the 64-bit results are added to the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] + ((_ZX_32(argument3.32[I]) * _ZX_32(
        argument2.32[I])) >> 32)
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.32 MADDUWD Multiply-Add Unsigned Words to Double Word

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0001

P	10	1	0001	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `madduwd` *registerW* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming unsigned integers, and the product is added to the 64 bits of the *registerW*. The result is stored back into the *registerW*.

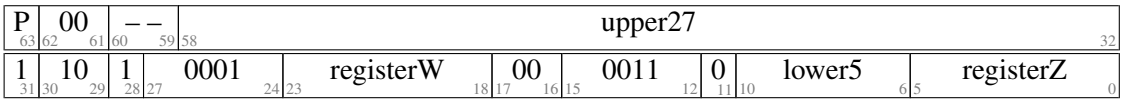
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_ZX_32(argument3) * _ZX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0001



Syntax `madduwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming unsigned integers, and the product is added to the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_ZX_32(argument3) * _ZX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.33 MADDUWDP Multiply-Add Unsigned Words to Double Word Pair

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1001

P	10	1	1001	registerM	–	00	0000	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `madduwdp` *registerM* = *registerZ*, *registerY*

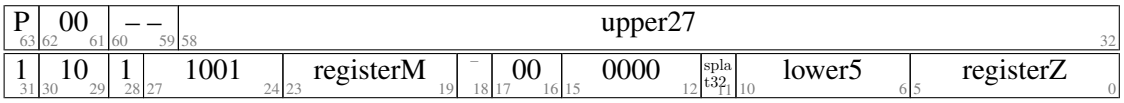
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words are multiplied by the *registerZ* words, assuming unsigned integers, and the product pair is added to the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] + (_ZX_32(argument3.32[I]) * _ZX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1001



Syntax `madduwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words are multiplied by the *registerZ* words, assuming unsigned integers, and the product pair is added to the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] + (_ZX_32(argument3.32[I]) * _ZX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.34 MADDUZDT Multiply-Add Unsigned Double to Tetra Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1011

P	10	1	1011	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `madduzdt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming unsigned 64-bit integers, and the product is added to the upper 64 bits of the *registerM*. The result is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = _ZX_64(argument1 >> 64) + (_ZX_64(argument3) * _ZX_64(
      argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.35 MADDW Multiply-Add Words

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0011

P	10	1	0011	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maddw` *registerW* = *registerZ*, *registerY*

Description The *registerY* least significant word and the *registerZ* least significant word are multiplied. The multiplication result is added to the value of the *registerW*. The resulting word is zero-extended and stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = _ZX_32(argument1 + (argument3.32[0] * argument2.32[0]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0011

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	10	1	0011				registerW						00		0011			0	lower5					registerZ									
31	30	29	28	27							24	23					18	17	16	15			12	11	10			6	5				0

Syntax `maddw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* least significant word and the *registerZ* least significant word are multiplied. The multiplication result is added to the value of the *registerW*. The resulting word is zero-extended and stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = _ZX_32(argument1 + (argument3.32[0] * argument2.32[0]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.36 MADDWD Multiply-Add Words to Double Word

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0000

P	10	1	0000	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maddwd` *registerW* = *registerZ*, *registerY*

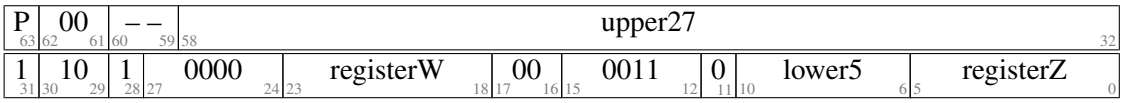
Description The *registerY* is multiplied by the *registerZ*, assuming signed integers, and the product is added to the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_SX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0000



Syntax `maddwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming signed integers, and the product is added to the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 + (_SX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.37 MADDWDP Multiply-Add Words to Double Word Pair

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1000

P	10	1	1000	registerM	–	00	0000	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `maddwdp` *registerM* = *registerZ*, *registerY*

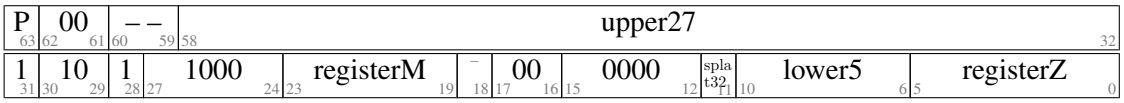
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words are multiplied by the *registerZ* words, assuming signed integers, and the product pair is added to the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] + (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1000



Syntax `maddwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words are multiplied by the *registerZ* words, assuming signed integers, and the product pair is added to the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] + (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.38 MADDWP — Multiply-Add Word Pair

Encoding Format MAU_DDDD0 (Section 6.12.348), exucode2 = 0001

P	10	1	0001	registerW	00	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `maddwp` *registerW* = *registerZ*, *registerY*

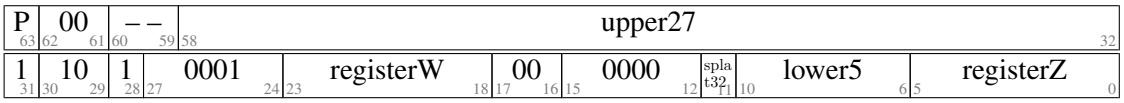
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are multiplied, assuming signed integers. The two resulting words are added to the respective words of the *registerW*. The resulting paired word is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument1.32[I] + (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDDD0.M (Section 6.12.349), exucode2 = 0001



Syntax `maddwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are multiplied, assuming signed integers. The two resulting words are added to the respective words of the *registerW*. The resulting paired word is stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

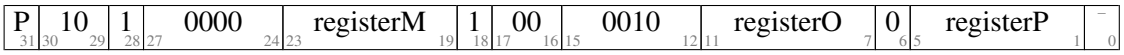
Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument1.32[I] + (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.39 MADDWQ Multiply-Add Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0000



Syntax `maddwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming signed integers. The four resulting words are added to the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] + (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.40 MM212W Matrix Multiply 2x1x2 Word Quadruple

Encoding Format MAU_QDD0 (Section 6.12.352), exucode2 = 0011

P	10	1	0011	registerM	–	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mm212w` *registerM* = *registerZ*, *registerY*

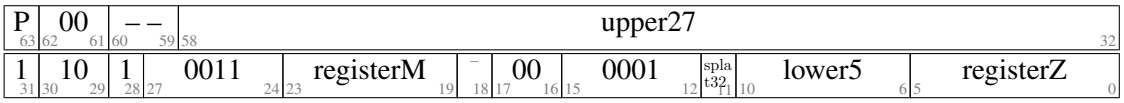
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The matrix multiplication between the *registerZ* words and the *registerY* words is computed, assuming integers. The four resulting words are stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1_x = _ZX_32(argument2.32[0] * argument3.32[0]);
  new result1_y = _ZX_32(argument2.32[0] * argument3.32[1]);
  new result1_z = _ZX_32(argument2.32[1] * argument3.32[0]);
  new result1_t = _ZX_32(argument2.32[1] * argument3.32[1]);
  result1.64[0] = result1_x | (result1_y << 32);
  result1.64[1] = result1_z | (result1_t << 32);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QDD0.M (Section 6.12.353), exucode2 = 0011



Syntax `mm212w` *registerM = registerZ, upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The matrix multiplication between the *registerZ* words and the *upper27_lower5* words is computed, assuming integers. The four resulting words are stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU.X (Section 6.14.43)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1_x = _ZX_32(argument2.32[0] * argument3.32[0]);
  new result1_y = _ZX_32(argument2.32[0] * argument3.32[1]);
  new result1_z = _ZX_32(argument2.32[1] * argument3.32[0]);
  new result1_t = _ZX_32(argument2.32[1] * argument3.32[1]);
  result1.64[0] = result1_x | (result1_y << 32);
  result1.64[1] = result1_z | (result1_t << 32);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.41 MMA212W Matrix Multiply-Add 2x1x2 Word

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1011

P	10	1	1011	registerM	–	00	0000	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mma212w` *registerM* = *registerZ*, *registerY*

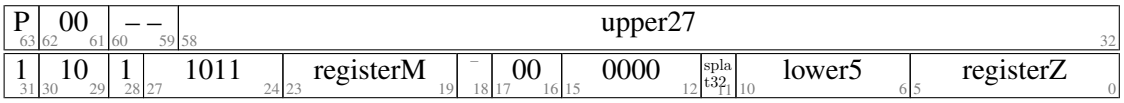
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The matrix multiplication between the *registerZ* words and the *registerY* words is computed, assuming integers. The four resulting words are added to the four words of the *registerM*, and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new argument1_x = argument1.32[0];
  new argument1_y = argument1.32[1];
  new argument1_z = argument1.32[2];
  new argument1_t = argument1.32[3];
  new result1_x = _ZX_32(argument1_x + (argument2.32[0] * argument3.32[0]));
  new result1_y = _ZX_32(argument1_y + (argument2.32[0] * argument3.32[1]));
  new result1_z = _ZX_32(argument1_z + (argument2.32[1] * argument3.32[0]));
  new result1_t = _ZX_32(argument1_t + (argument2.32[1] * argument3.32[1]));
  result1.64[0] = result1_x | (result1_y << 32);
  result1.64[1] = result1_z | (result1_t << 32);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1011



Syntax `mma212w` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The matrix multiplication between the *registerZ* words and the *upper27_lower5* words is computed, assuming integers. The four resulting words are added to the four words of the *registerM*, and stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new argument1_x = argument1.32[0];
  new argument1_y = argument1.32[1];
  new argument1_z = argument1.32[2];
  new argument1_t = argument1.32[3];
  new result1_x = _ZX_32(argument1_x + (argument2.32[0] * argument3.32[0]));
  new result1_y = _ZX_32(argument1_y + (argument2.32[0] * argument3.32[1]));
  new result1_z = _ZX_32(argument1_z + (argument2.32[1] * argument3.32[0]));
  new result1_t = _ZX_32(argument1_t + (argument2.32[1] * argument3.32[1]));
  result1.64[0] = result1_x | (result1_y << 32);
  result1.64[1] = result1_z | (result1_t << 32);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.42 MMS212W Matrix Multiply-Subtract 2x1x2 Word

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1111

P	10	1	1111	registerM	–	00	0000	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mms212w` *registerM* = *registerZ*, *registerY*

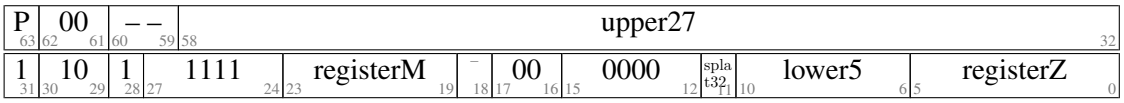
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The matrix multiplication between the *registerZ* words and the *registerY* words is computed, assuming integers. The four resulting words are subtracted from the four words of the *registerM*, and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new argument1_x = argument1.32[0];
  new argument1_y = argument1.32[1];
  new argument1_z = argument1.32[2];
  new argument1_t = argument1.32[3];
  new result1_x = _ZX_32(argument1_x - (argument2.32[0] * argument3.32[0]));
  new result1_y = _ZX_32(argument1_y - (argument2.32[0] * argument3.32[1]));
  new result1_z = _ZX_32(argument1_z - (argument2.32[1] * argument3.32[0]));
  new result1_t = _ZX_32(argument1_t - (argument2.32[1] * argument3.32[1]));
  result1.64[0] = result1_x | (result1_y << 32);
  result1.64[1] = result1_z | (result1_t << 32);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1111



Syntax `mms212w` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The matrix multiplication between the *registerZ* words and the *upper27_lower5* words is computed, assuming integers. The four resulting words are subtracted from the four words of the *registerM*, and stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new argument1_x = argument1.32[0];
  new argument1_y = argument1.32[1];
  new argument1_z = argument1.32[2];
  new argument1_t = argument1.32[3];
  new result1_x = _ZX_32(argument1_x - (argument2.32[0] * argument3.32[0]));
  new result1_y = _ZX_32(argument1_y - (argument2.32[0] * argument3.32[1]));
  new result1_z = _ZX_32(argument1_z - (argument2.32[1] * argument3.32[0]));
  new result1_t = _ZX_32(argument1_t - (argument2.32[1] * argument3.32[1]));
  result1.64[0] = result1_x | (result1_y << 32);
  result1.64[1] = result1_z | (result1_t << 32);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.43 MSBFD — Multiply-Subtract Double Word

Encoding Format MAU_DDDD0 (Section 6.12.348), exucode2 = 0100

P	10	1	0100	registerW	00	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `msbfd` *registerW* = *registerZ*, *registerY*

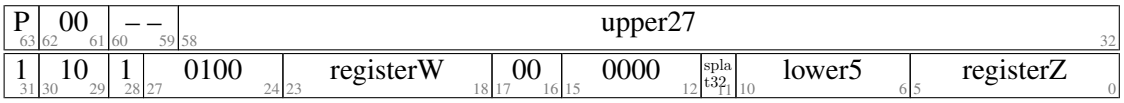
Description The *registerY* is multiplied by the *registerZ*, assuming 64-bit integers, and the product is subtracted from the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 - (_SX_64(argument3) * _SX_64(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDDD0.M (Section 6.12.349), exucode2 = 0100



Syntax `msbfd` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming 64-bit integers, and the product is subtracted from the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 - (_SX_64(argument3) * _SX_64(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.44 MSBFDT Multiply-Subtract Double to Tetra Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1100

P	10	1	1100	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfdt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming 64-bit integers, and the product is subtracted from the 128 bits of the *registerM*. The result is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 - (_SX_64(argument3) * _SX_64(argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.45 MSBFHQ Multiply-Subtract Half Word Quadruple

Encoding Format MAU_DDDD0 (Section 6.12.348), exucode2 = 0110

P	10	1	0110	registerW	00	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `msbfhq` *registerW* = *registerZ*, *registerY*

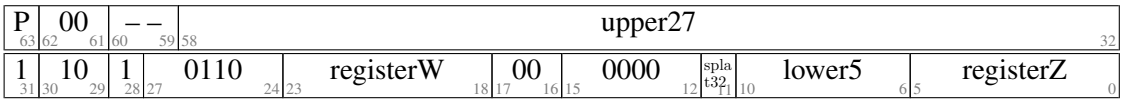
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting half words are subtracted from to the respective half words of the *registerW*. The resulting half words are packed and stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument1.16[I] - (_SX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDDD0.M (Section 6.12.349), exucode2 = 0110



Syntax `msbfhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting half words are subtracted from to the respective half words of the *registerW*. The resulting half words are packed and stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = argument1.16[I] - (_SX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.46 MSBFHWQ Multiply-Subtract Half Word from Word Quadruple

Encoding Format MAU_QQDD3 (Section 6.12.368), exucode2 = 0100

P	10	1	0100	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting words are subtracted from the respective words of the *registerM*. The resulting words are packed and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] - (_SX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.47 MSBFMWQ Multiply-Subtract Most Significant Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0101

P	10	1	0101	registerM	1	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `msbfmwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming signed integers. The four most significant words of the 64-bit results are subtracted from the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] - ((_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I])) >> 32)
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.48 MSBFSUDT Multiply-Subtract Signed by Unsigned Double to Tetra Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1110

P	10	1	1110	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfsudt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming unsigned 64-bit integer and the *registerZ* assuming signed 64-bit integer are multiplied. The product is subtracted from the 128 bits of the *registerM* and the result is stored back into the *registerM*.

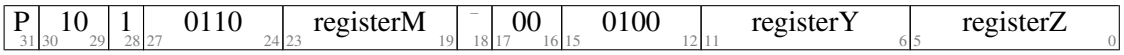
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 - (_ZX_64(argument3) * _SX_64(argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.49 MSBFSUHWQ Multiply-Subtract Signed by Unsigned Half Word from Word Quadruple

Encoding Format MAU_QQDD3 (Section 6.12.368), exucode2 = 0110



Syntax `msbfsuhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words assuming unsigned integers and the *registerZ* half words assuming signed integers are multiplied. The four resulting words are subtracted from the respective words of the *registerM*. The resulting words are packed and stored back into the *registerM*.

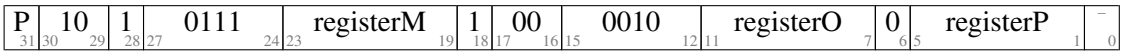
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] - (_ZX_16(argument3.16[I]) * _SX_16(
        argument2.16[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.50 MSBFSUMWQ Multiply-Subtract Signed by Unsigned Most Significant Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0111



Syntax `msbfsumwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words assuming unsigned integers and the *registerP* words assuming signed integers are multiplied. The four most significant words of the 64-bit results are subtracted from the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] - ((_ZX_32(argument3.32[I]) * _SX_32(
        argument2.32[I])) >> 32)
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.51 MSBFSUWD Multiply-Subtract Signed by Unsigned Words to Double Word

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0110

P	10	1	0110	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `msbfsuwd` *registerW* = *registerZ*, *registerY*

Description The *registerY* assuming unsigned integer and the *registerZ* assuming signed integer are multiplied. The product is subtracted from the 64 bits of the *registerW* and the result is stored back into the *registerW*.

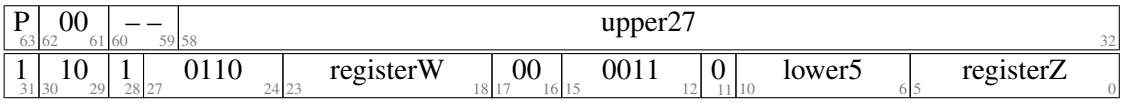
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = _ZX_64(argument1) - (_ZX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0110



Syntax `msbfsuwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* assuming unsigned integer and the *registerZ* assuming signed integer are multiplied. The product is subtracted from the 64 bits of the *registerW* and the result is stored back into the *registerW*.

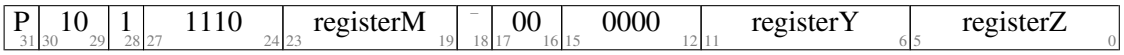
Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = _ZX_64(argument1) - (_ZX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.52 MSBFSUWDP Multiply-Subtract Signed by Unsigned Words to Double Word Pair

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1110



Syntax `msbfsuwdp` *registerM* = *registerZ*, *registerY*

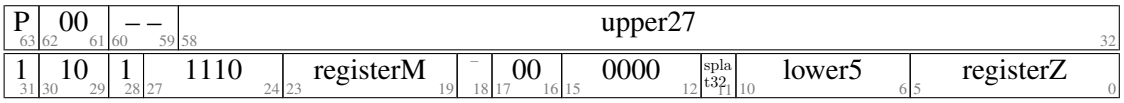
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words assuming unsigned integer and the *registerZ* words assuming signed integer are multiplied. The product pair is subtracted from the 64-bit pair of the *registerM* and the resulting pair is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] - (_ZX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1110



Syntax `msbfsuwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words assuming unsigned integer and the *registerZ* words assuming signed integer are multiplied. The product pair is subtracted from the 64-bit pair of the *registerM* and the resulting pair is stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] - (_ZX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.53 MSBFUDT Multiply-Subtract Unsigned Double to Tetra Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1101

P	10	1	1101	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfudt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming unsigned 64-bit integers, and the product is subtracted from the 128 bits of the *registerM*. The result is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = argument1 - (_ZX_64(argument3) * _ZX_64(argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.54 MSBFUHWQ Multiply-Subtract Unsigned Half Word from Word Quadruple

Encoding Format MAU_QQDD3 (Section 6.12.368), exucode2 = 0101

P	10	1	0101	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfuhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming unsigned integers. The four resulting words are subtracted from the respective words of the *registerM*. The resulting words are packed and stored back into the *registerM*.

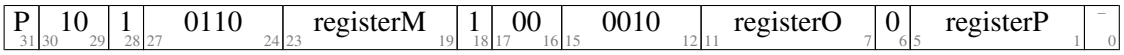
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] - (_ZX_16(argument3.16[I]) * _ZX_16(
        argument2.16[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.55 MSBFUMWQ Multiply-Subtract Unsigned Most Significant Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0110



Syntax `msbfumwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming unsigned integers. The four most significant words of the 64-bit results are subtracted from the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] - ((_ZX_32(argument3.32[I]) * _ZX_32(
        argument2.32[I])) >> 32)
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.56 MSBFUWD Multiply-Subtract Unsigned Words to Double Word

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0101

P	10	1	0101	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `msbfuwd` *registerW* = *registerZ*, *registerY*

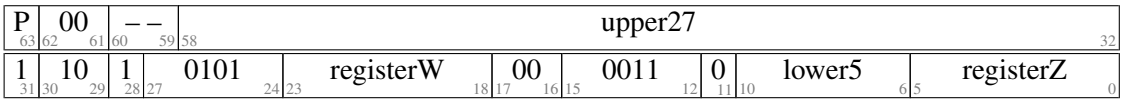
Description The *registerY* is multiplied by the *registerZ*, assuming unsigned integers, and the product is subtracted from the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 - (_ZX_32(argument3) * _ZX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0101



Syntax `msbfuwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming unsigned integers, and the product is subtracted from the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 - (_ZX_32(argument3) * _ZX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.57 MSBFUWDP Multiply-Subtract Unsigned Words to Double Word Pair

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1101

P	10	1	1101	registerM	–	00	0000	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfuwdp` *registerM* = *registerZ*, *registerY*

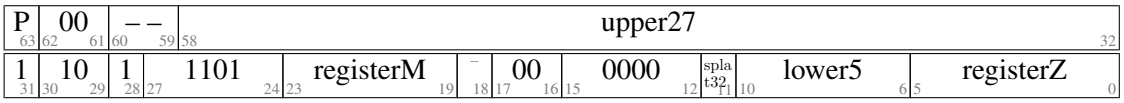
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words are multiplied by the *registerZ*, assuming unsigned integers, and the product pair is subtracted from the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] - (_ZX_32(argument3.32[I]) * _ZX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1101



Syntax `msbfuwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words are multiplied by the *registerZ*, assuming unsigned integers, and the product pair is subtracted from the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] - (_ZX_32(argument3.32[I]) * _ZX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.58 MSBFUZDT Multiply-Subtract Unsigned Double to Quadruple Word

Encoding Format MAU_QQDD1 (Section 6.12.356), exucode2 = 1111

P	10	1	1111	registerM	0	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfuzdt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming unsigned 64-bit integers, and the product is subtracted from the upper 64 bits of the *registerM*. The result is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  new result1 = _ZX_64(argument1 >> 64) - (_ZX_64(argument3) * _ZX_64(
      argument2));
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.59 MSBFW — Multiply-Subtract Words

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0111

P	10	1	0111	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `msbfw` *registerW* = *registerZ*, *registerY*

Description The *registerY* least significant word and the *registerZ* least significant word are multiplied. The multiplication result is subtracted from the value of the *registerW*. The resulting word is zero-extended and stored back into the *registerW*.

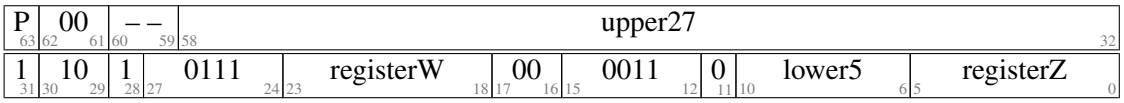
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = _ZX_32(argument1 - (argument3.32[0] * argument2.32[0]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0111



Syntax `msbfw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* least significant word and the *registerZ* least significant word are multiplied. The multiplication result is subtracted from the value of the *registerW*. The resulting word is zero-extended and stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = _ZX_32(argument1 - (argument3.32[0] * argument2.32[0]));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.60 MSBFWD Multiply-Subtract Words to Double Word

Encoding Format MAU_SSSS (Section 6.12.364), exucode2 = 0100

P	10	1	0100	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `msbfwd` *registerW* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming signed integers, and the product is subtracted from the 64 bits of the *registerW*. The result is stored back into the *registerW*.

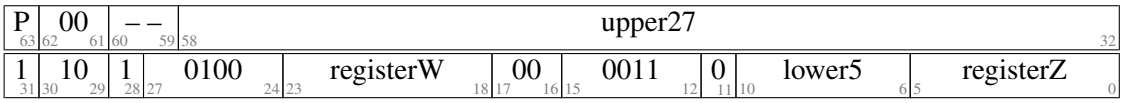
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 - (_SX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_SSSS.W (Section 6.12.365), exucode2 = 0100



Syntax `msbfwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming signed integers, and the product is subtracted from the 64 bits of the *registerW*. The result is stored back into the *registerW*.

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  new result1 = argument1 - (_SX_32(argument3) * _SX_32(argument2));
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.61 MSBFWDP Multiply-Subtract Words to Double Word Pair

Encoding Format MAU_QQDD (Section 6.12.350), exucode2 = 1100

P	10	1	1100	registerM	–	00	0000	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `msbfwdp` *registerM* = *registerZ*, *registerY*

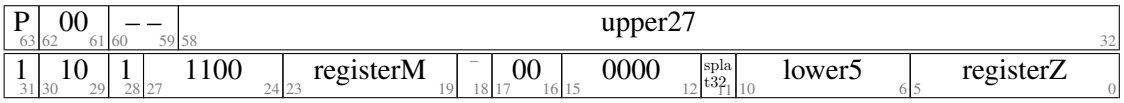
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words are multiplied by the *registerZ* words, assuming signed integers, and the product pair is subtracted from the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] - (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QQDD.M (Section 6.12.351), exucode2 = 1100



Syntax `msbfwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words are multiplied by the *registerZ* words, assuming signed integers, and the product pair is subtracted from the 64-bit pair of the *registerM*. The resulting pair is stored back into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = argument1.64[I] - (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.62 MSBFWP Multiply-Subtract Word Pair

Encoding Format MAU_DDDD0 (Section 6.12.348), exucode2 = 0101

P	10	1	0101	registerW	00	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `msbfwp` *registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are multiplied, assuming signed integers. The two resulting words are subtracted from the respective words of the *registerW*. The resulting paired word is stored back into the *registerW*.

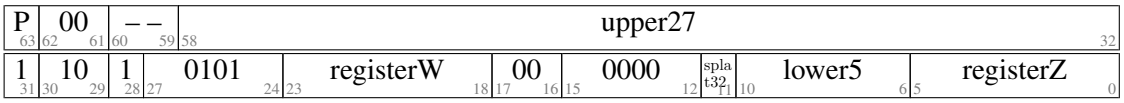
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument1.32[I] - (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_DDDD0.M (Section 6.12.349), exucode2 = 0101



Syntax `msbfwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are multiplied, assuming signed integers. The two resulting words are subtracted from the respective words of the *registerW*. The resulting paired word is stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

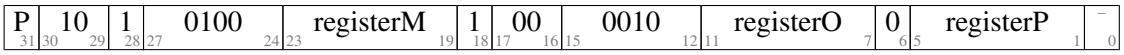
Execution Reservation table MAU_AUXR.X (Section 6.14.46)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
stage E1:
  new argument1 = GPR[registerW];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = argument1.32[I] - (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.63 MSBFWQ Multiply-Subtract Word Quadruple

Encoding Format MAU_QQQQ (Section 6.12.360), exucode2 = 0100



Syntax `msbfwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming signed integers. The four resulting words are subtracted from the respective words of the *registerM*, packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
stage E1:
  new argument1 = PGR[registerM];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = argument1.32[I] - (_SX_32(argument3.32[I]) * _SX_32(
        argument2.32[I]))
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.64 MULD — Multiply Double Word

Encoding Format MAU_DDD0 (Section 6.12.354), exucode2 = 0100

P	10	1	0100	registerW	00	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `muld` *registerW* = *registerZ*, *registerY*

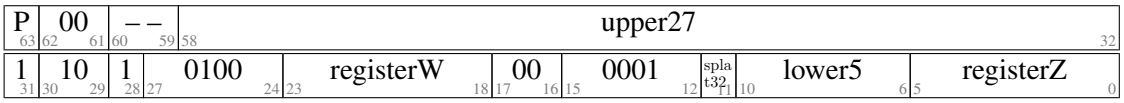
Description The *registerY* is multiplied by the *registerZ*, assuming 64-bit integers. The 64 bits of the result are stored into the *registerW*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) * _SX_64(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDD0.M (Section 6.12.355), exucode2 = 0100



Syntax `muld` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming 64-bit integers. The 64 bits of the result are stored into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU.X (Section 6.14.43)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) * _SX_64(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.65 MULDT — Multiply Double Word to Tetra Word

Encoding Format MAU_QDD1 (Section 6.12.357), exucode2 = 1000

P	10	1	1000	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `muldt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming 64-bit integers. The 128 bits of the result are stored into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_64(argument3) * _SX_64(argument2);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.66 MULHQ — Multiply Half Word Quadruple

Encoding Format MAU_DDD0 (Section 6.12.354), exucode2 = 0110

P	10	1	0110	registerW	00	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `mulhq` *registerW* = *registerZ*, *registerY*

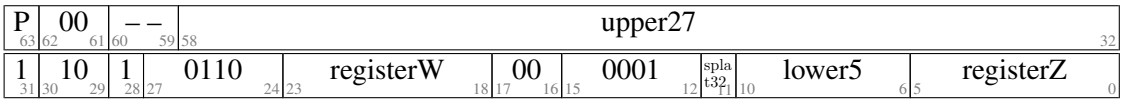
Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting half words are packed and stored back into the *registerW*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) * _SX_16(argument2.16[I])
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDD0.M (Section 6.12.355), exucode2 = 0110



Syntax `mulhq` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as four half words packed into 64 bits. The *upper27_lower5* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting half words are packed and stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU.X (Section 6.14.43)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.16[I] = _SX_16(argument3.16[I]) * _SX_16(argument2.16[I])
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.67 MULHWQ — Multiply Half Word to Word Quadruple

Encoding Format MAU_QDD3 (Section 6.12.369), exucode2 = 1000

P	10	1	1000	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mulhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming signed integers. The four resulting words are packed and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = _SX_16(argument3.16[I]) * _SX_16(argument2.16[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.68 MULMWQ Multiply Most Significant Word Quadruple

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0101

P	10	1	0101	registerM	0	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `mulmwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming signed integers. The four most significant words of the 64-bit results are packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = (_SX_32(argument3.32[I]) * _SX_32(argument2.32[I])) >> 32
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.69 MULSUDT Multiply Signed by Unsigned Double Words to Tetra Word

Encoding Format MAU_QDD1 (Section 6.12.357), exucode2 = 1010

P	10	1	1010	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mulsudt` *registerM* = *registerZ*, *registerY*

Description The *registerY* assuming unsigned double integer and the *registerZ* assuming signed double integer are multiplied. The 128 bits of the result are stored into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument3) * _SX_64(argument2);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.70 MULSUHWQ Multiply Signed by Unsigned Half Word to Word Quadruple

Encoding Format MAU_QDD3 (Section 6.12.369), exucode2 = 1010

P	10	1	1010	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mulsuhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words assuming unsigned integers and the *registerZ* half words assuming signed integers are multiplied. The four resulting words are packed and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = _ZX_16(argument3.16[I]) * _SX_16(argument2.16[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.71 MULSUMWQ Multiply Signed by Unsigned Most Significant Word Quadruple

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0111

P	10	1	0111	registerM	0	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `mulsumwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words assuming unsigned integers and the *registerP* words assuming signed integers are multiplied. The four most significant words of the 64-bit results are packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = (_ZX_32(argument3.32[I]) * _SX_32(argument2.32[I])) >> 32
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.72 MULSUWD Multiply Signed by Unsigned Words to Double Word

Encoding Format MAU_SSS (Section 6.12.366), exucode2 = 1010

P	10	1	1010	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `mulsuwd` *registerW* = *registerZ*, *registerY*

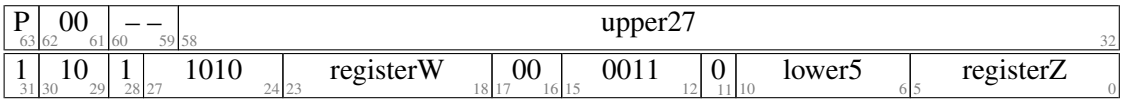
Description The *registerY* assuming unsigned integer and the *registerZ* assuming signed integer are multiplied. The 64 bits of the result are stored into the *registerW*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument3) * _SX_32(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_SSS.W (Section 6.12.367), exucode2 = 1010



Syntax `mulsuwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* assuming unsigned integer and the *registerZ* assuming signed integer are multiplied. The 64 bits of the result are stored into the *registerW*.

Execution Reservation table MAU.X (Section 6.14.43)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument3) * _SX_32(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.73 MULSUWDP Multiply Signed by Unsigned Words to Double Word Pair

Encoding Format MAU_QDD0 (Section 6.12.352), exucode2 = 0010

P	10	1	0010	registerM	–	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mulsuwdp` *registerM* = *registerZ*, *registerY*

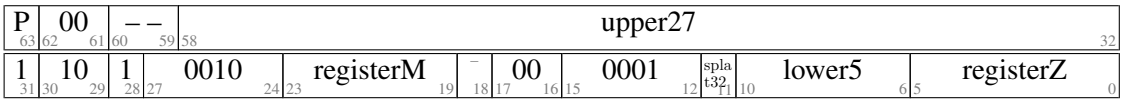
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words assuming unsigned integers and the *registerZ* words assuming signed integers are multiplied. The resulting 64-bit pair is stored into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = _ZX_32(argument3.32[I]) * _SX_32(argument2.32[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QDD0.M (Section 6.12.353), exucode2 = 0010



Syntax `mulsuwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words assuming unsigned integers and the *registerZ* words assuming signed integers are multiplied. The resulting 64-bit pair is stored into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU.X (Section 6.14.43)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = _ZX_32(argument3.32[I]) * _SX_32(argument2.32[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.7.74 MULUDT Multiply Unsigned Double Words to Tetra Word

Encoding Format MAU_QDD1 (Section 6.12.357), exucode2 = 1001

P	10	1	1001	registerM	1	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `muludt` *registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ* assuming unsigned double integers The 128 bits of the result are stored into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_64(argument3) * _ZX_64(argument2);
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.75 MULUHWQ Multiply Unsigned Half Word to Word Quadruple

Encoding Format MAU_QDD3 (Section 6.12.369), exucode2 = 1001

P	10	1	1001	registerM	–	00	0100	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `muluhwq` *registerM* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as four half words packed into 64 bits. The *registerY* half words and the *registerZ* half words are multiplied, assuming unsigned integers. The four resulting words are packed and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = _ZX_16(argument3.16[I]) * _ZX_16(argument2.16[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.76 MULUMWQ Multiply Unsigned Most Significant Word Quadruple

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0110

P	10	1	0110	registerM	0	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `mulumwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming unsigned integers. The four most significant words of the 64-bit results are packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = (_ZX_32(argument3.32[I]) * _ZX_32(argument2.32[I])) >> 32
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.77 MULUWD — Multiply Unsigned Words to Double Word

Encoding Format MAU_SSS (Section 6.12.366), exucode2 = 1001

P	10	1	1001	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `muluwd` *registerW* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming unsigned integers. The 64 bits of the result are stored into the *registerW*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument3) * _ZX_32(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_SSS.W (Section 6.12.367), exucode2 = 1001

P	00	– –	upper27
63	62 61	60 59	58 32

1	10	1	1001	registerW	00	0011	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `muluwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming unsigned integers. The 64 bits of the result are stored into the *registerW*.

Execution Reservation table MAU.X (Section 6.14.43)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument3) * _ZX_32(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.78 MULUWDP Multiply Unsigned Words to Double Word Pair

Encoding Format MAU_QDD0 (Section 6.12.352), exucode2 = 0001

P	10	1	0001	registerM	–	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `muluwdp` *registerM* = *registerZ*, *registerY*

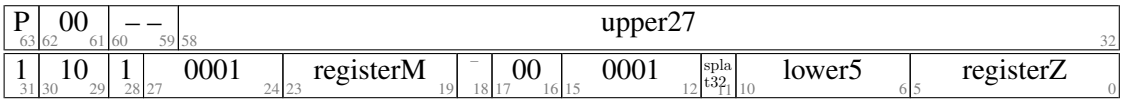
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words are multiplied by the *registerZ* words, assuming unsigned integers. The resulting 64-bit pair is stored into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = _ZX_32(argument3.32[I]) * _ZX_32(argument2.32[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QDD0.M (Section 6.12.353), exucode2 = 0001



Syntax `muluwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words are multiplied by the *registerZ* words, assuming unsigned integers. The resulting 64-bit pair is stored into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU.X (Section 6.14.43)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = _ZX_32(argument3.32[I]) * _ZX_32(argument2.32[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.79 MULW — Multiply Words

Encoding Format MAU_SSS (Section 6.12.366), exucode2 = 1011

P	10	1	1011	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `mulw` *registerW* = *registerZ*, *registerY*

Description The *registerY* least significant word and the *registerZ* least significant word are multiplied, assuming signed integers. The resulting word is zero-extended and stored back into the *registerW*.

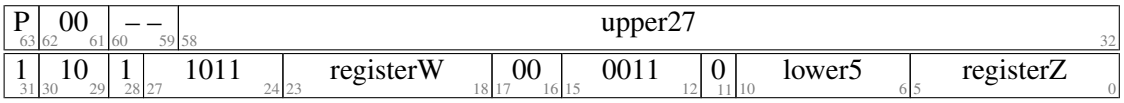
Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument3.32[0] * argument2.32[0]);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_SSS.W (Section 6.12.367), exucode2 = 1011



Syntax `mulw` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* least significant word and the *registerZ* least significant word are multiplied, assuming signed integers. The resulting word is zero-extended and stored back into the *registerW*.

Execution Reservation table MAU.X (Section 6.14.43)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument3.32[0] * argument2.32[0]);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.80 MULWD — Multiply Words to Double Word

Encoding Format MAU_SSS (Section 6.12.366), exucode2 = 1000

P	10	1	1000	registerW	00	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `mulwd` *registerW* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming signed integers. The 64 bits of the result are stored into the *registerW*.

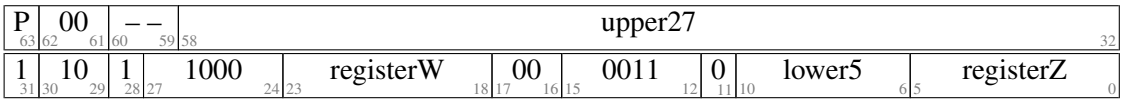
Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _SX_32(argument3) * _SX_32(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format MAU_SSS.W (Section 6.12.367), exucode2 = 1000



Syntax `mulwd` *registerW* = *registerZ*, *upper27_lower5*

Description The *upper27_lower5* is multiplied by the *registerZ*, assuming signed integers. The 64 bits of the result are stored into the *registerW*.

Execution Reservation table MAU.X (Section 6.14.43)

```
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _SX_32(argument3) * _SX_32(argument2);
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.81 MULWDP — Multiply Words to Double Word Pair

Encoding Format MAU_QDD0 (Section 6.12.352), exucode2 = 0000

P	10	1	0000	registerM	–	00	0001	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `mulwdp` *registerM* = *registerZ*, *registerY*

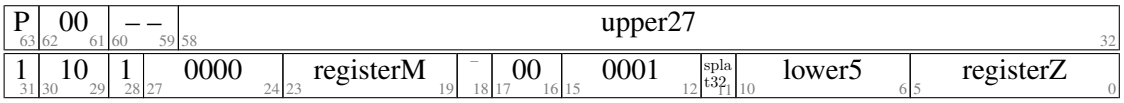
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words are multiplied by the *registerZ* words , assuming signed integers. The resulting 64-bit pair is stored into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = _SX_32(argument3.32[I]) * _SX_32(argument2.32[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_QDD0.M (Section 6.12.353), exucode2 = 0000



Syntax `mulwdp` *registerM* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words are multiplied by the *registerZ* words , assuming signed integers. The resulting 64-bit pair is stored into the *registerM*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU.X (Section 6.14.43)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.64[I] = _SX_32(argument3.32[I]) * _SX_32(argument2.32[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.7.82 MULWP — Multiply Word Pair

Encoding Format MAU_DDD0 (Section 6.12.354), exucode2 = 0101

P	10	1	0101	registerW	00	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `mulwp` *registerW* = *registerZ*, *registerY*

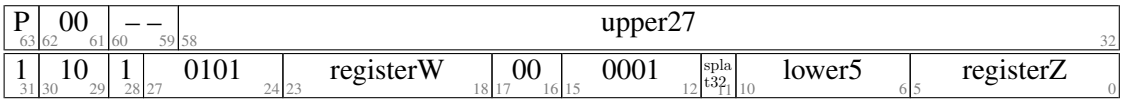
Description The *registerY* and the *registerZ* are considered as two words packed into 64 bits. The *registerY* words and the *registerZ* words are multiplied, assuming signed integers. The two resulting words are packed and stored back into the *registerW*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) * _SX_32(argument2.32[I])
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format MAU_DDD0.M (Section 6.12.355), exucode2 = 0101



Syntax `mulwp` *registerW* = *registerZ*, *upper27_lower5splat32*

Description The *upper27_lower5* and the *registerZ* are considered as two words packed into 64 bits. The *upper27_lower5* words and the *registerZ* words are multiplied, assuming signed integers. The two resulting words are packed and stored back into the *registerW*.

Modifier(s)

- splat32 (cf. splat32 in section 6.10.2 on page 1430)

Execution Reservation table MAU.X (Section 6.14.43)

```
stage ID:
  new splat32 = _ZX_1(splat32);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  for (I = 0; I < 2; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) * _SX_32(argument2.32[I])
  };
stage E2:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.7.83 MULWQ — Multiply Word Quadruple

Encoding Format MAU_QQQ (Section 6.12.359), exucode2 = 0100

P	10	1	0100	registerM	0	00	0010	registerO	0	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `mulwq` *registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as four words packed into 128 bits. The *registerO* words and the *registerP* words are multiplied, assuming signed integers. The four resulting words are packed into 128 bits and stored back into the *registerM*.

Execution Reservation table MAU (Section 6.14.42)

```
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  for (I = 0; I < 4; I += 1) {
    result1.32[I] = _SX_32(argument3.32[I]) * _SX_32(argument2.32[I])
  };
stage E2:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8 FPU Instructions

6.8.1 FABSD — Floating-Point Absolute Value Double Word

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0001

P	11	1	0001	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fabsd` *registerW* = *registerZ*

Description The absolute value the *registerZ*, assuming binary 64 floating-point number, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

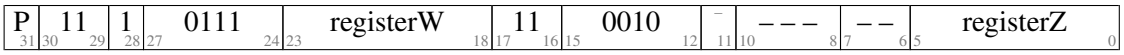
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2 & 0x7FFFFFFFFFFFFFFF;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.2 FABSHQ Floating-Point Absolute Value Half Word Quadruple

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0111



Syntax `fabshq` *registerW* = *registerZ*

Description The absolute values of the *registerZ*, assuming a quadruple of binary 16 floating-point numbers, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2 & 0x7FFF7FFF7FFF7FFF;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.3 FABSW — Floating-Point Absolute Value Word

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0011

P	11	1	0011	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fabsw` *registerW* = *registerZ*

Description The absolute value the *registerZ*, assuming binary 32 floating-point number, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

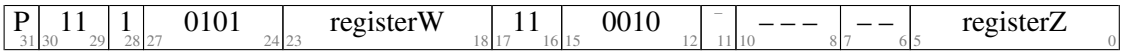
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument2 & 0x7FFFFFFF);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.4 FABSWP Floating-Point Absolute Value Word Pair

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0101



Syntax `fabswp` *registerW* = *registerZ*

Description The absolute values of the *registerZ*, assuming a pair of binary 32 floating-point numbers, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2 & 0x7FFFFFFF7FFFFFFF;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.5 FADDD — Floating-Point Add Double Word

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0000

P	10	1	0000	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `faddd`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is added to the *registerY*, assuming binary 64 floating-point numbers. The result is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fadd_64_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.6 FADDDC.C Floating-Point Add Double Word Complex Conjugate

Encoding Format MAU_FQQQ0 (Section 6.12.381), exucode2 = 1101

P	10	1	1101	registerM	0	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fadddc.c`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as binary 64 floating-point complex numbers. The conjugate of the *registerP* is added to the *registerO*, according to the rounding mode set in the CS register. The resulting binary 64 floating-point complex number is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = faddcc_64_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.7 FADDDP — Floating-Point Add Double Word Pair

Encoding Format MAU_FQQQ0 (Section 6.12.381), exucode2 = 1100

P	10	1	1100	registerM	0	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fadddp`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerP* is added to the *registerO*, assuming binary 64 floating-point number pairs. The resulting pair is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

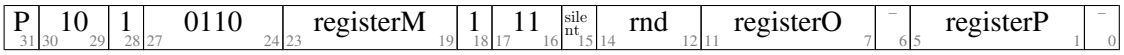
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fadd_64_64_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.8 FADDHO — Floating-Point Add Half Word Octuple

Encoding Format MAU_FQQQ2 (Section 6.12.379), exucode2 = 0110



Syntax `faddho`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerP* is added to the *registerO*, assuming binary 16 floating-point number octuples. The resulting octuple is rounded to the binary 16 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fadd_16_16_x8(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.9 FADDHQ Floating-Point Add Half Word Quadruple

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0010

P	10	1	0010	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `faddhq`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is added to the *registerY*, assuming binary 16 floating-point number quadruples. The results are rounded to the binary 16 floating-point format according to the rounding mode set in the CS register, packed and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fadd_16_16_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.10 FADDW — Floating-Point Add Words

Encoding Format MAU_FSSS (Section 6.12.376), exucode2 = 1100

P	10	1	1100	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `faddw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is added to the *registerY*, assuming binary 32 floating-point numbers. The result is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fadd_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.11 FADDWC.C Floating-Point Add Word Complex Conjugate

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0011

P	10	1	0011	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `faddwc.c`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as binary 32 floating-point complex numbers. The conjugate of the *registerZ* is added to the *registerY*, according to the rounding mode set in the CS register. The resulting binary 32 floating-point complex number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

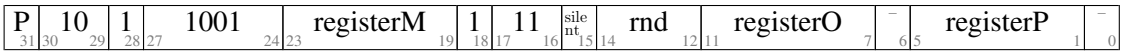
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = faddcc_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.12 FADDWCP.C Floating-Point Add Word Complex Pair Conjugate

Encoding Format MAU_FQQQ1 (Section 6.12.382), exucode2 = 1001



Syntax `faddwcp.c`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of binary 32 floating-point complex numbers. The conjugate of the *registerP* pair is added to the *registerO* pair, according to the rounding mode set in the CS register. The resulting pair of binary 32 floating-point complex numbers is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = faddcc_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.13 FADDWP Floating-Point Add Word Pair

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0001

P	10	1	0001	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `faddwp`*roundingsilent registerW = registerZ, registerY*

Description The *registerZ* is added to the *registerY*, assuming binary 32 floating-point number pairs. The resulting pair is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fadd_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.14 FADDWQ Floating-Point Add Word Quadruple

Encoding Format MAU_FQQQ1 (Section 6.12.382), exucode2 = 1000

P	10	1	1000	registerM	1	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `faddwq`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerP* is added to the *registerO*, assuming binary 32 floating-point number quadruples. The resulting quadruple is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fadd_32_32_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.15 FCDIVD Floating-Point Cases for Division Double Word

Encoding Format ALU_DFWRR (Section 6.12.300), exucode2 = 0001

P	11	1	0001	registerW	11	0101	sile nt2	– – –	– –	registerP	–
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 1	0

Syntax `fcdivd`*silent2 registerW = registerP*

Description The value of the special case for the division of the least significant half by the most significant half of the *registerP*, interpreted as binary 64 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = PGR[registerP];
  new argument2_0 = _ZX_64(argument2);
  new argument2_1 = _ZX_64(argument2 >> 64);
  new result1 = fcdiv_64(argument2_0, argument2_1);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.16 FCDIVW — Floating-Point Cases for Word Division

Encoding Format ALU_DFWRR (Section 6.12.300), exucode2 = 0011

P	11	1	0011	registerW	11	0101	silent2	– – –	– –	registerP	–
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 1	0

Syntax `fcdivw`*silent2 registerW = registerP*

Description The value of the special case for the division of the least significant half by the most significant half of the *registerP*, interpreted as binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = PGR[registerP];
  new argument2_0 = argument2.32[0];
  new argument2_1 = argument2.32[2];
  new result1 = fcdiv_32(argument2_0, argument2_1);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.17 FCDIVWP Floating-Point Cases for Division Word Pair

Encoding Format ALU_DFWRR (Section 6.12.300), exucode2 = 0101

P	11	1	0101	registerW	11	0101	silent2	– – –	– –	registerP	–
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 1	0

Syntax `fcdivwp`*silent2 registerW = registerP*

Description The pair of values of the special cases for the division of the least significant half by the most significant half of the *registerP*, interpreted as pairs of binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = PGR[registerP];
  new argument2_0 = _ZX_64(argument2);
  new argument2_1 = _ZX_64(argument2 >> 64);
  new result1 = fcdiv_32_x2(argument2_0, argument2_1);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.18 FCOMPD — Floating-Point Compare Double

Encoding Format ALU_FCWRRS (Section 6.12.291), exubit27 = 1

P	11	1	1	floatcomp	registerW	11	0000	registerY	registerZ
31	30 29	28	27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `fcompd`*floatcomp registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *floatcomp*, assuming binary 64 floating-point numbers. The boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = floatcomp_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_FCWRRS.W (Section 6.12.292), exubit27 = 1

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1	floatcomp	registerW	11	0000	0	lower5	registerZ
31	30 29	28	27	26 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `fcompd`*floatcomp registerW = registerZ, upper27_lower5*

Description The *registerZ* is compared to the *upper27_lower5* using condition *floatcomp*, assuming binary 64 floating-point numbers. The boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

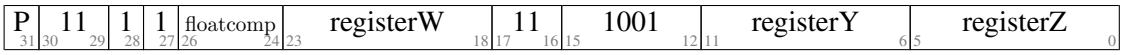
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = floatcomp_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.19 FCOMPND Floating-Point Compare Negate Double

Encoding Format ALU_FCNWRRS (Section 6.12.305), exubit27 = 1



Syntax `fcompnd`*floatcomp registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *floatcomp*, assuming binary 64 floating-point numbers. The negated boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = -floatcomp_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_FCNWRRS.W (Section 6.12.306), exubit27 = 1

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1	floatcomp	registerW	11	1001	0	lower5	registerZ
31	30 29	28	27	26 24	23 18	17 16	15 12	11	10 6	5 0

Syntax `fcompnd`*floatcomp registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is compared to the *upper27_lower5* using condition *floatcomp*, assuming binary 64 floating-point numbers. The negated boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

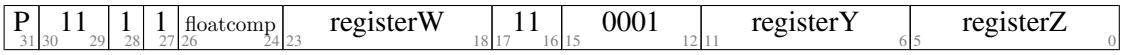
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = -floatcomp_64(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.20 FCOMPNHQ Floating-Point Compare Negate Half Word Quadruple

Encoding Format ALU_HQFCWRR (Section 6.12.295), exubit27 = 1



Syntax `fcompnhq`*floatcomp registerW = registerZ, registerY*

Description The *registerZ* and the *registerY* are considered as four half words packed into 64 bits. The *registerZ* half words are compared to the *registerY* half words using condition *floatcomp*, assuming binary 16 floating-point numbers. The resulting booleans are negated, packed and stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

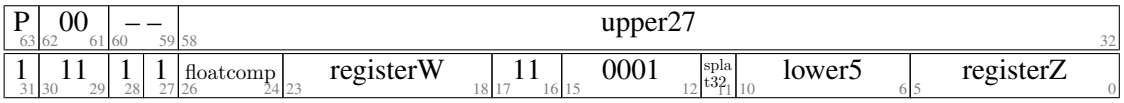
- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  result1.16[0] = -floatcomp_16(argument4, argument2.16[0], argument3.16[0]);
  result1.16[1] = -floatcomp_16(argument4, argument2.16[1], argument3.16[1]);
  result1.16[2] = -floatcomp_16(argument4, argument2.16[2], argument3.16[2]);
  result1.16[3] = -floatcomp_16(argument4, argument2.16[3], argument3.16[3]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_HQFCWRR.M (Section 6.12.296), exubit27 = 1



Syntax `fcompnhq`*floatcomp registerW = registerZ, upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as four half words packed into 64 bits. The *registerZ* half words are compared to the *upper27_lower5* half words using condition *floatcomp*, assuming binary 16 floating-point numbers. The resulting booleans are negated, packed and stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

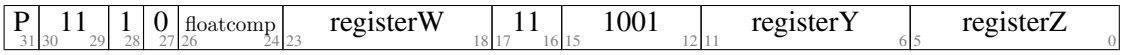
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  result1.16[0] = -floatcomp_16(argument4, argument2.16[0], argument3.16[0]);
  result1.16[1] = -floatcomp_16(argument4, argument2.16[1], argument3.16[1]);
  result1.16[2] = -floatcomp_16(argument4, argument2.16[2], argument3.16[2]);
  result1.16[3] = -floatcomp_16(argument4, argument2.16[3], argument3.16[3]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.21 FCOMPNW Floating-Point Compare Negate Single

Encoding Format ALU_FCNWRRS (Section 6.12.305), exubit27 = 0



Syntax `fcompnw`*floatcomp registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *floatcomp*, assuming binary 32 floating-point numbers. The negated boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

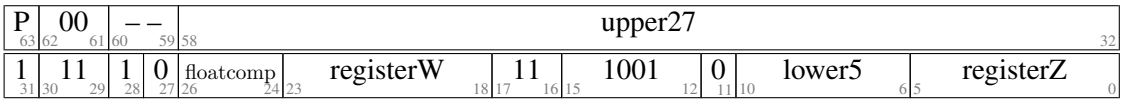
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(-floatcomp_32(argument4, argument2, argument3));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

Encoding Format ALU_FCNWRRS.W (Section 6.12.306), exubit27 = 0



Syntax `fcompnw`*floatcomp registerW* = *registerZ*, *upper27_lower5*

Description The *registerZ* is compared to the *upper27_lower5* using condition *floatcomp*, assuming binary 32 floating-point numbers. The negated boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(-floatcomp_32(argument4, argument2, argument3));
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.22 FCOMPNWP Floating-Point Compare Negate Word Pair

Encoding Format ALU_WPFCWRR (Section 6.12.293), exubit27 = 0

P	11	1	0	floatcomp	registerW	11	0001	registerY	registerZ
31	30 29	28	27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `fcompnwp`*floatcomp registerW* = *registerZ*, *registerY*

Description The *registerZ* and the *registerY* are considered as two words packed into 64 bits. The *registerZ* words are compared to the *registerY* words using condition *floatcomp*, assuming binary 32 floating-point numbers. The resulting booleans are negated, packed and stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

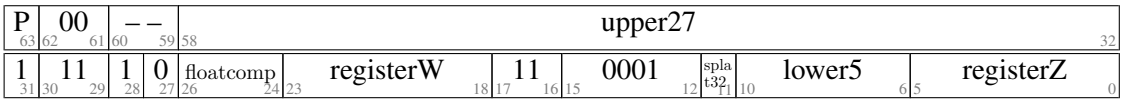
- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  result1.32[0] = -floatcomp_32(argument4, argument2.32[0], argument3.32[0]);
  result1.32[1] = -floatcomp_32(argument4, argument2.32[1], argument3.32[1]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_WPFCWRR.M (Section 6.12.294), exubit27 = 0



Syntax `fcompnwp`*floatcomp registerW = registerZ, upper27_lower5splat32*

Description The *registerZ* and the *upper27_lower5* are considered as two words packed into 64 bits. The *registerZ* words are compared to the *upper27_lower5* words using condition *floatcomp*, assuming binary 32 floating-point numbers. The resulting booleans are negated, packed and stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)
- splat32 (cf. splat32 in section 6.10.2 on page 1430)

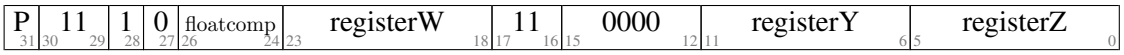
Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new splat32 = _ZX_1(splat32);
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = splat32 ? (_WX_32(upper27_lower5) << 32) | _ZX_32(_WX_32(
      upper27_lower5)) : _SX_32(_WX_32(upper27_lower5));
  new argument2 = GPR[registerZ];
  result1.32[0] = -floatcomp_32(argument4, argument2.32[0], argument3.32[0]);
  result1.32[1] = -floatcomp_32(argument4, argument2.32[1], argument3.32[1]);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.23 FCOMPW Floating-Point Compare Single

Encoding Format ALU_FCWRRS (Section 6.12.291), exubit27 = 0



Syntax `fcompw`*floatcomp registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY* using condition *floatcomp*, assuming binary 32 floating-point numbers. The boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

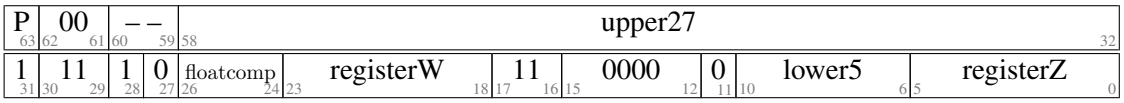
- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = floatcomp_32(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

Encoding Format ALU_FCWRRS.W (Section 6.12.292), exubit27 = 0



Syntax `fcompw`*floatcomp registerW = registerZ, upper27_lower5*

Description The *registerZ* is compared to the *upper27_lower5* using condition *floatcomp*, assuming binary 32 floating-point numbers. The boolean result is stored into the *registerW*. This instruction does not raise an invalid exception when handling a signaling NaN.

Modifier(s)

- floatcomp (cf. floatcomp in section 6.10.2 on page 1420)

Execution Reservation table ALU_TINY.X (Section 6.14.12)

```
stage ID:
  new argument4 = _ZX_3(floatcomp);
stage RR:
  new argument3 = _WX_32(upper27_lower5);
  new argument2 = GPR[registerZ];
  new result1 = floatcomp_32(argument4, argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.24 FDOT2W Floating-Point Dot Product x2 Word

Encoding Format MAU_FDDD (Section 6.12.373), exucode2 = 1100

P	10	1	1100	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fdot2w`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as two binary 32 floating-point numbers packed into 64 bits. The dot product between the *registerZ* words and the *registerY* words is computed. The result represented as binary 32 floating-point number is stored back into the *registerW*.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fdot2_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.25 FDOT2WD Floating-Point Dot Product x2 Word to Double Word

Encoding Format MAU_FDDD (Section 6.12.373), exucode2 = 1101

P	10	1	1101	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fdot2wd`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* and the *registerZ* are considered as two binary 32 floating-point numbers packed into 64 bits. The dot product between the *registerZ* words and the *registerY* words is computed. The result represented as binary 64 floating-point number is stored back into the *registerW*.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

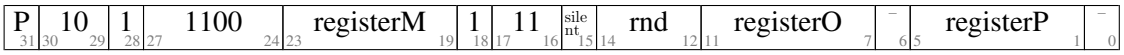
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fdot2_32_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.26 FDOT2WDP Floating-Point Dot Product x2 Word to Double Word Pair

Encoding Format MAU_FQQQ1 (Section 6.12.382), exucode2 = 1100



Syntax `fdot2wdp`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered each of four words packed into 128 bits. The dot-product of the *registerO* upper word pair and the *registerP* upper word pair is computed, assuming binary 32 floating-point numbers. The dot-product of the *registerO* lower word pair and the *registerP* lower word pair is computed, assuming binary 32 floating-point numbers. The resulting pair of binary 64 floating-point numbers is packed and stored back into the *registerM*.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fdot2_32_64_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.27 FDOT2WZP Floating-Point Dot Product x2 Word Zero Extended Pair

Encoding Format MAU_FQQQ1 (Section 6.12.382), exucode2 = 1101

P	10	1	1101	registerM	1	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fdot2wzp`*roundingsilent registerM = registerP, registerO*

Description The *registerO* and the *registerP* are considered as two binary 32 floating-point numbers packed into 64 bits. The dot product between the *registerP* words and the *registerO* words is computed. The result represented as binary 32 floating-point number is zero-extended to 64 bits and stored back into the *registerM*.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fdot2_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.28 FFDMASW Floating-Point Fused Dual Multiply Add Subtract Word

Encoding Format MAU_FDMDAW (Section 6.12.342), exucode2 = 0011

P	10	0	0011	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffdmasw`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* and the *registerZ* are considered as pairs of binary 32 floating-point numbers. The low half of the *registerZ* is multiplied by the low half of the *registerY* to produce a low product. The high half of the *registerZ* is multiplied by the high half of the *registerY* to produce a high product. The sum of the *registerW*, the low product and the negated high product is rounded according to the rounding mode. The resulting binary 32 floating-point number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmas_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.29 FFDMASWP Floating-Point Fused Dual Multiply Add Subtract Word pair

Encoding Format MAU_FDMDAWP (Section 6.12.343), exucode2 = 0111

P	10	0	0111	registerW	10	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 18	17 16	15	14 12	11 7	6	5 1	0

Syntax `ffdmaswp`*roundingsilent registerW* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of pairs of binary 32 floating-point numbers. The low half of each *registerP* inner pair is multiplied by the low half of each *registerO* inner pair to produce a low product pair. The high half of each *registerP* inner pair is multiplied by the high half of each *registerO* inner pair to produce a high product pair. The sums of the *registerW*, the low product pair and the negated high product pair are rounded according to the rounding mode. The resulting binary 32 floating-point number pair is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

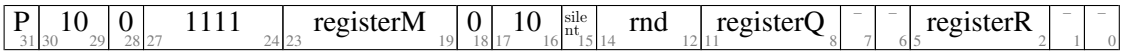
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmas_32_32_x2(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.30 FFDMASWQFloating-Point Fused Dual Multiply Add Subtract Word Quadruple

Encoding Format MAU_FDMDAWQ (Section 6.12.345), exucode2 = 1111



Syntax `ffdmaswq`*roundingsilent registerM = registerR, registerQ*

Description The *registerQ* and the *registerR* are considered as pairs of pairs of binary 32 floating-point numbers. The low half of each *registerR* inner pair is multiplied by the low half of each *registerQ* inner pair to produce a low product pair. The high half of each *registerR* inner pair is multiplied by the high half of each *registerQ* inner pair to produce a high product pair. The sums of the *registerM*, the low product pair and the negated high product pair are rounded according to the rounding mode. The resulting binary 32 floating-point number pair is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = QGR[registerQ];
  new argument2 = QGR[registerR];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmas_32_32_x4(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.31 FFDMAW Floating-Point Fused Dual Multiply Add Word

Encoding Format MAU_FDMAW (Section 6.12.334), exucode2 = 0000

P	10	0	0000	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffdmaw`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* and the *registerZ* are considered as pairs of binary 32 floating-point numbers. The low half of the *registerZ* is multiplied by the low half of the *registerY* to produce a low product. The high half of the *registerZ* is multiplied by the high half of the *registerY* to produce a high product. The sum of the low product and the high product is rounded according to the rounding mode. The resulting binary 32 floating-point number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

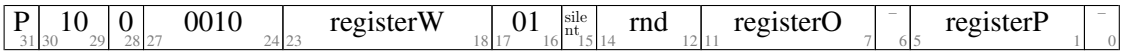
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdma_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.32 FFDMAWP Floating-Point Fused Dual Multiply Add Word Pair

Encoding Format MAU_FDMAWP (Section 6.12.335), exucode2 = 0010



Syntax `ffdmawp`*roundingsilent registerW* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of pairs of binary 32 floating-point numbers. The low half of each *registerP* inner pair is multiplied by the low half of each *registerO* inner pair to produce a low product pair. The high half of each *registerP* inner pair is multiplied by the high half of each *registerO* inner pair to produce a high product pair. The sums of the low product pair and the high product pair are rounded according to the rounding mode. The resulting binary 32 floating-point number pair is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdma_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.33 FFDMAWQ Floating-Point Fused Dual Multiply Add Word Quadruple

Encoding Format MAU_FDMAWQ (Section 6.12.337), exucode2 = 0110

P	10	0	0110	registerM	0	01	silent	rnd	registerQ	–	–	registerR	–	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 8	7	6	5 2	1	0

Syntax `ffdmawq`*roundingsilent registerM* = *registerR*, *registerQ*

Description The *registerQ* and the *registerR* are considered as quadruples of pairs of binary 32 floating-point numbers. The low half of each *registerR* inner pair is multiplied by the low half of each *registerQ* inner pair to produce a low product quadruple. The high half of each *registerR* inner pair is multiplied by the high half of each *registerQ* inner pair to produce a high product quadruple. The sums of the low product quadruple and the high product quadruple are rounded according to the rounding mode. The resulting binary 32 floating-point number quadruple is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = QGR[registerQ];
  new argument2 = QGR[registerR];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdma_32_32_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.34 FFDMDAW Floating-Point Fused Dual Multiply Dual Add Word

Encoding Format MAU_FDMDAW (Section 6.12.342), exucode2 = 0000

P	10	0	0000	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffdmdaw`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* and the *registerZ* are considered as pairs of binary 32 floating-point numbers. The low half of the *registerZ* is multiplied by the low half of the *registerY* to produce a low product. The high half of the *registerZ* is multiplied by the high half of the *registerY* to produce a high product. The sum of the *registerW*, the low product and the high product is rounded according to the rounding mode. The resulting binary 32 floating-point number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

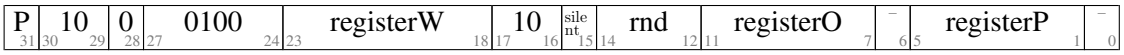
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmda_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.35 FFDMDAWP Floating-Point Fused Dual Multiply Dual Add Word Pair

Encoding Format MAU_FDMDAWP (Section 6.12.343), exucode2 = 0100



Syntax `ffdmdawp`*roundingsilent registerW* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of pairs of binary 32 floating-point numbers. The low half of each *registerP* inner pair is multiplied by the low half of each *registerO* inner pair to produce a low product pair. The high half of each *registerP* inner pair is multiplied by the high half of each *registerO* inner pair to produce a high product pair. The sums of the *registerW*, the low product pair and the high product pair are rounded according to the rounding mode. The resulting binary 32 floating-point number pair is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmda_32_32_x2(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.36 FFDMDAWQ Floating-Point Fused Dual Multiply Dual Add Word Quadruple

Encoding Format MAU_FDMDAWQ (Section 6.12.345), exucode2 = 1100

P	10	0	1100	registerM	0	10	silent	rnd	registerQ	–	–	registerR	–	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 8	7	6	5 2	1	0

Syntax `ffdmdawq`*roundingsilent registerM* = *registerR*, *registerQ*

Description The *registerQ* and the *registerR* are considered as quadruples of pairs of binary 32 floating-point numbers. The low half of each *registerR* inner pair is multiplied by the low half of each *registerQ* inner pair to produce a low product quadruple. The high half of each *registerR* inner pair is multiplied by the high half of each *registerQ* inner pair to produce a high product quadruple. The sums of the *registerM*, the low product quadruple and the high product quadruple are rounded according to the rounding mode. The resulting binary 32 floating-point number quadruple is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

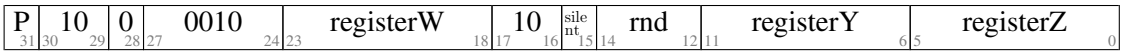
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = QGR[registerQ];
  new argument2 = QGR[registerR];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmda_32_32_x4(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.37 FFDMDSW Floating-Point Fused Dual Multiply Dual Subtract Word

Encoding Format MAU_FDMDAW (Section 6.12.342), exucode2 = 0010



Syntax `ffdmdsw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as pairs of binary 32 floating-point numbers. The low half of the *registerZ* is multiplied by the low half of the *registerY* to produce a low product. The high half of the *registerZ* is multiplied by the high half of the *registerY* to produce a high product. The sum of the *registerW*, the negated low product and the negated high product is rounded according to the rounding mode. The resulting binary 32 floating-point number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

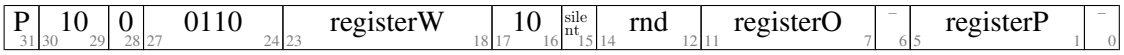
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmds_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.38 FFDMDSWP Floating-Point Fused Dual Multiply Dual Subtract Word Pair

Encoding Format MAU_FDMDAWP (Section 6.12.343), exucode2 = 0110



Syntax `ffdmdswp`*roundingsilent registerW = registerP, registerO*

Description The *registerO* and the *registerP* are considered as pairs of pairs of binary 32 floating-point numbers. The low half of each *registerP* inner pair is multiplied by the low half of each *registerO* inner pair to produce a low product pair. The high half of each *registerP* inner pair is multiplied by the high half of each *registerO* inner pair to produce a high product pair. The sums of the *registerW*, the negated low product pair and the negated high product pair are rounded according to the rounding mode. The resulting binary 32 floating-point number pair is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

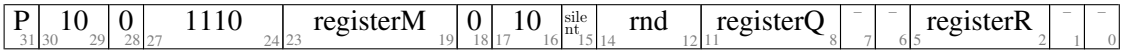
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmds_32_32_x2(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.39 FFDMDSWQ Floating-Point Fused Dual Multiply Dual Subtract Word Quadruple

Encoding Format MAU_FDMDAWQ (Section 6.12.345), exucode2 = 1110



Syntax `ffdmdswq`*roundingsilent registerM* = *registerR*, *registerQ*

Description The *registerQ* and the *registerR* are considered as quadruples of pairs of binary 32 floating-point numbers. The low half of each *registerR* inner pair is multiplied by the low half of each *registerQ* inner pair to produce a low product quadruple. The high half of each *registerR* inner pair is multiplied by the high half of each *registerQ* inner pair to produce a high product quadruple. The sums of the *registerM*, the negated low product quadruple and the negated high product quadruple are rounded according to the rounding mode. The resulting binary 32 floating-point number quadruple is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = QGR[registerQ];
  new argument2 = QGR[registerR];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmds_32_32_x4(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.40 FFDMSAW Floating-Point Fused Dual Multiply Subtract Add Word

Encoding Format MAU_FDMDAW (Section 6.12.342), exucode2 = 0001

P	10	0	0001	registerW	10	sile nt	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffdmsaw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as pairs of binary 32 floating-point numbers. The low half of the *registerZ* is multiplied by the low half of the *registerY* to produce a low product. The high half of the *registerZ* is multiplied by the high half of the *registerY* to produce a high product. The sum of the *registerW*, the negated low product and the high product is rounded according to the rounding mode. The resulting binary 32 floating-point number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

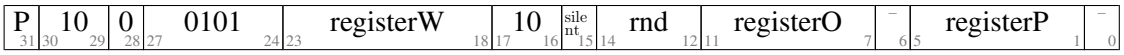
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmsa_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.41 FFDMSAWP Floating-Point Fused Dual Multiply Subtract Add Word Pair

Encoding Format MAU_FDMDAWP (Section 6.12.343), exucode2 = 0101



Syntax `ffdmsawp`*roundingsilent registerW* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of pairs of binary 32 floating-point numbers. The low half of each *registerP* inner pair is multiplied by the low half of each *registerO* inner pair to produce a low product pair. The high half of each *registerP* inner pair is multiplied by the high half of each *registerO* inner pair to produce a high product pair. The sums of the *registerW*, the negated low product pair and the high product pair are rounded according to the rounding mode. The resulting binary 32 floating-point number pair is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmsa_32_32_x2(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.42 FFDMSAWQ Floating-Point Fused Dual Multiply Subtract Add Word Quadruple

Encoding Format MAU_FDMDAWQ (Section 6.12.345), exucode2 = 1101

P	10	0	1101	registerM	0	10	silent	rnd	registerQ	–	–	registerR	–	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 8	7	6	5 2	1	0

Syntax `ffdmsawq`*roundingsilent registerM* = *registerR*, *registerQ*

Description The *registerQ* and the *registerR* are considered as quadruples of pairs of binary 32 floating-point numbers. The low half of each *registerR* inner pair is multiplied by the low half of each *registerQ* inner pair to produce a low product quadruple. The high half of each *registerR* inner pair is multiplied by the high half of each *registerQ* inner pair to produce a high product quadruple. The sums of the *registerM*, the negated low product quadruple and the high product quadruple are rounded according to the rounding mode. The resulting binary 32 floating-point number quadruple is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

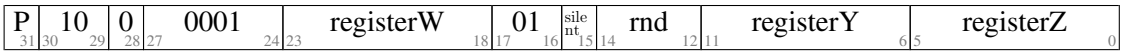
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = QGR[registerQ];
  new argument2 = QGR[registerR];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdmsa_32_32_x4(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.43 FFDMSW Floating-Point Fused Dual Multiply Subtract Word

Encoding Format MAU_FDMAW (Section 6.12.334), exucode2 = 0001



Syntax `ffdmsw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as pairs of binary 32 floating-point numbers. The low half of the *registerZ* is multiplied by the low half of the *registerY* to produce a low product. The high half of the *registerZ* is multiplied by the high half of the *registerY* to produce a high product. The sum of the low product and the negated high product is rounded according to the rounding mode. The resulting binary 32 floating-point number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdms_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.44 FFDMSWP Floating-Point Fused Dual Multiply Subtract Word Pair

Encoding Format MAU_FDMAWP (Section 6.12.335), exucode2 = 0011

P	10	0	0011	registerW	01	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 18	17 16	15	14 12	11 7	6	5 1	0

Syntax `ffdmswp`*roundingsilent registerW* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of pairs of binary 32 floating-point numbers. The low half of each *registerP* inner pair is multiplied by the low half of each *registerO* inner pair to produce a low product pair. The high half of each *registerP* inner pair is multiplied by the high half of each *registerO* inner pair to produce a high product pair. The sums of the low product pair and the negated high product pair are rounded according to the rounding mode. The resulting binary 32 floating-point number pair is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

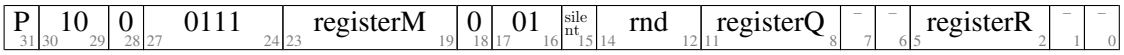
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdms_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.45 FFDMSWQ Floating-Point Fused Dual Multiply Subtract Word Quadruple

Encoding Format MAU_FDMAWQ (Section 6.12.337), exucode2 = 0111



Syntax `ffdmswq`*roundingsilent registerM* = *registerR*, *registerQ*

Description The *registerQ* and the *registerR* are considered as quadruples of pairs of binary 32 floating-point numbers. The low half of each *registerR* inner pair is multiplied by the low half of each *registerQ* inner pair to produce a low product quadruple. The high half of each *registerR* inner pair is multiplied by the high half of each *registerQ* inner pair to produce a high product quadruple. The sums of the low product quadruple and the negated high product quadruple are rounded according to the rounding mode. The resulting binary 32 floating-point number quadruple is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = QGR[registerQ];
  new argument2 = QGR[registerR];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffdms_32_32_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.46 FFMAD — Floating-Point Fused Multiply-Add Double

Encoding Format MAU_FMAD (Section 6.12.324), exucode2 = 0100

P	10	0	0100	registerW	00	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmad`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 64 floating-point numbers, and the product is added to the *registerW*. The result is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_64_64(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.47 FFMAHO Floating-Point Fused Multiply-Add Half Word Octuple

Encoding Format MAU_FQQQQ (Section 6.12.380), exucode2 = 1010

P	10	1	1010	registerM	0	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `ffmaho`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* is multiplied by the *registerP*, assuming binary 16 floating-point number octuples, and the product octuple is added to the *registerM*. The resulting octuple is rounded to the binary 16 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM + (registerO \cdot registerP)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_16_16_x8(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.48 FFMAHQ Floating-Point Fused Multiply-Add Half Word Quadruple

Encoding Format MAU_FDDDD (Section 6.12.372), exucode2 = 0011

P	10	1	0011	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmahq`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 16 floating-point number quadruples, and the product quadruple is added to the *registerW*. The resulting quadruple is rounded to the binary 16 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_16_16_x4(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.49 FFMAHW Floating-Point Fused Multiply-Add Half Words to Word

Encoding Format MAU_FSSSS (Section 6.12.375), exucode2 = 1000

P	10	1	1000	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmahw`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 16 floating-point numbers, and the product is added to the *registerW*, assuming binary 32 floating-point format. The result is rounded according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_16_32(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.50 FFMAHWQ Floating-Point Fused Multiply-Add Half Words to Word Quadruple

Encoding Format MAU_FQQDD (Section 6.12.377), exucode2 = 0001

P	10	1	0001	registerM	0	11	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 6	5 0

Syntax `ffmahwq`*roundingsilent registerM = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming quadruples of binary 16 floating-point numbers, and the quadruples of products is added to the *registerM*, assuming a quadruple of binary 32 floating-point format. The results are rounded according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_16_32_x4(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.51 FFMAW Floating-Point Fused Multiply-Add Words

Encoding Format MAU_FMAW (Section 6.12.322), exucode2 = 0000

P	10	0	0000	registerW	00	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmaw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 32 floating-point numbers, and the product is added to the *registerW*. The result is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

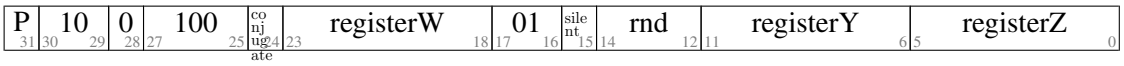
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_32_32(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.52 FFMAWC Floating-Point Multiply Add Word Complex

Encoding Format MAU_FMAWC (Section 6.12.339), exucode3 = 100



Syntax `ffmawc`*conjugateroundingsilent registerW* = *registerZ*, *registerY*

Description The *registerW*, the *registerY* and the *registerZ* are considered as binary 32 floating-point complex numbers. The *registerZ* is multiplied by the *registerY* to produce a complex product. If the *conjugate* is set, the complex conjugate of the *registerZ* is used in the complex multiplication. The complex product is added to the *registerW* and rounded according to the rounding mode. The resulting binary 32 floating-point complex number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- conjugate (cf. conjugate in section 6.10.2 on page 1418)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new conjugate = _ZX_1(conjugate);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  if (conjugate) {
    argument2 = fconj_32_32(argument2);
  }
  new result1 = ffmac_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
```


Kalray – all rights reserved

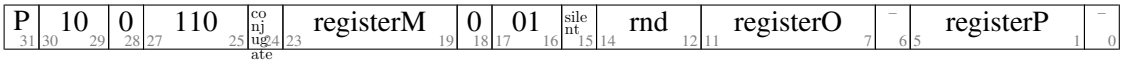
©2025 Kalray

```
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.53 FFMAWCP Floating-Point Multiply Add Word Complex Pair

Encoding Format MAU_FMAWCP (Section 6.12.340), exucode3 = 110



Syntax `ffmawcp`*conjugateroundingsilent registerM = registerP, registerO*

Description The *registerM*, the *registerO* and the *registerP* are considered as pairs of binary 32 floating-point complex numbers. The low half of the *registerP* is multiplied by the low half of the *registerO* to produce a low complex product. The high half of the *registerP* is multiplied by the high half of the *registerO* to produce a high complex product. If the *conjugate* is set, the complex conjugates of the *registerP* are used in the complex multiplication. The complex product pair is added to the *registerM* and rounded according to the rounding mode. The resulting binary 32 floating-point complex number pair is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- conjugate (cf. conjugate in section 6.10.2 on page 1418)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new conjugate = _ZX_1(conjugate);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  if (conjugate) {
    argument2 = fconj_32_32_x2(argument2);
  }
  new result1 = ffmac_32_32_x2(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
```


Kalray – all rights reserved

©2025 Kalray

```
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.54 FFMAWD Floating-Point Fused Multiply-Add Words to Double Word

Encoding Format MAU_FDDDD (Section 6.12.372), exucode2 = 0001

P	10	1	0001	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmawd`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 32 floating-point numbers, and the product is added to the *registerW*, assuming binary 64 floating-point format. The result is rounded according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_32_64(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.55 FFMAWDP Floating-Point Fused Multiply-Add Words to Double Word Pair

Encoding Format MAU_FQQDD (Section 6.12.377), exucode2 = 0000

P	10	1	0000	registerM	0	11	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 6	5 0

Syntax `ffmawdp`*roundingsilent registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming pairs of binary 32 floating-point numbers, and the pair of products is added to the *registerM*, assuming a pair of binary 64 floating-point format. The results are rounded according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_32_64_x2(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.56 FFMAWP Floating-Point Fused Multiply-Add Word Pair

Encoding Format MAU_FMAWP (Section 6.12.323), exucode2 = 0010

P	10	0	0010	registerW	00	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmawp`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 32 floating-point number pairs, and the product pair is added to the *registerW*. The resulting pair is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_32_32_x2(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.57 FFMAWQ Floating-Point Fused Multiply-Add Word Quadruple

Encoding Format MAU_FMAWQ (Section 6.12.325), exucode2 = 0110

P	10	0	0110	registerM	0	00	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `ffmawq`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* is multiplied by the *registerP*, assuming binary 32 floating-point number quadruples, and the product quadruple is added to the *registerM*. The resulting quadruple is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM + (registerO \cdot registerP)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffma_32_32_x4(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.58 FFMSD — Floating-Point Fused Multiply-Subtract Double

Encoding Format MAU_FMAD (Section 6.12.324), exucode2 = 0101

P	10	0	0101	registerW	00	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmsd`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 64 floating-point numbers, and the product is subtracted from the *registerW*. The result is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW - (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

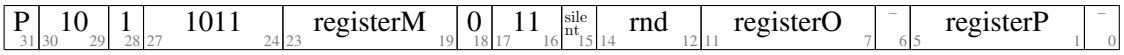
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_64_64(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.59 FFMSHO Floating-Point Fused Multiply-Subtract Half Word Octuple

Encoding Format MAU_FQQQQ (Section 6.12.380), exucode2 = 1011



Syntax `ffmsho`*roundingsilent registerM = registerP, registerO*

Description The *registerO* is multiplied by the *registerP*, assuming binary 16 floating-point number octuples, and the product octuple is subtracted from the *registerM*. The resulting octuple is rounded to the binary 16 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM + (registerO \cdot registerP)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_16_16_x8(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.60 FFMSHQ Floating-Point Fused Multiply-Subtract Half Word Quadruple

Encoding Format MAU_FDDDD (Section 6.12.372), exucode2 = 0111

P	10	1	0111	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmshq`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 16 floating-point number quadruples, and the product quadruple is subtracted from the *registerW*. The resulting quadruple is rounded to the binary 16 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_16_16_x4(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.61 FFMSHW Floating-Point Fused Multiply-Subtract Half Words to Word

Encoding Format MAU_FSSSS (Section 6.12.375), exucode2 = 1010

P	10	1	1010	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmshw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 16 floating-point numbers, and the product is subtracted from the *registerW*, assuming binary 32 floating-point format. The result is rounded according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW - (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

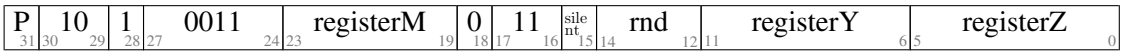
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_16_32(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.62 FFMSHWQ Floating-Point Fused Multiply-Subtract Half Words to Word Quadruple

Encoding Format MAU_FQQDD (Section 6.12.377), exucode2 = 0011



Syntax `ffmshwq`*roundingsilent registerM = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming quadruples of binary 16 floating-point numbers, and the quadruples of products is subtracted from the *registerM*, assuming a quadruple of binary 32 floating-point format. The results are rounded according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_16_32_x4(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.63 FFMSW Floating-Point Fused Multiply-Subtract Words

Encoding Format MAU_FMAW (Section 6.12.322), exucode2 = 0001

P	10	0	0001	registerW	00	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmsw`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 32 floating-point numbers, and the product is subtracted from the *registerW*. The result is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW - (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

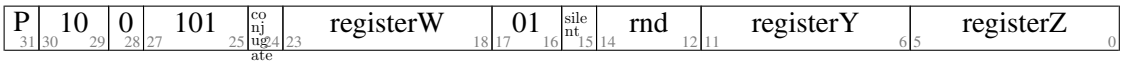
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_32_32(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.64 FFMSWC Floating-Point Multiply Subtract Word Complex

Encoding Format MAU_FMAWC (Section 6.12.339), exucode3 = 101



Syntax `ffmswc`*conjugateroundingsilent registerW* = *registerZ*, *registerY*

Description The *registerW*, the *registerY* and the *registerZ* are considered as binary 32 floating-point complex numbers. The *registerZ* is multiplied by the *registerY* to produce a complex product. If the *conjugate* is set, the complex conjugate of the *registerZ* is used in the complex multiplication. The complex product is subtracted from the *registerW* and rounded according to the rounding mode. The resulting binary 32 floating-point complex number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- conjugate (cf. conjugate in section 6.10.2 on page 1418)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new conjugate = _ZX_1(conjugate);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  if (conjugate) {
    argument2 = fconj_32_32(argument2);
  }
  new result1 = ffmsc_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
```


Kalray – all rights reserved

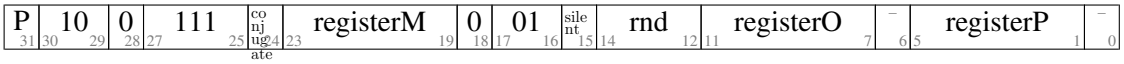
©2025 Kalray

```
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.65 FFMSWCP Floating-Point Multiply Subtract Word Complex Pair

Encoding Format MAU_FMAWCP (Section 6.12.340), exucode3 = 111



Syntax `ffmswcp`*conjugateroundingsilent registerM = registerP, registerO*

Description The *registerM*, the *registerO* and the *registerP* are considered as pairs of binary 32 floating-point complex numbers. The low half of the *registerP* is multiplied by the low half of the *registerO* to produce a low complex product. The high half of the *registerP* is multiplied by the high half of the *registerO* to produce a high complex product. If the *conjugate* is set, the complex conjugates of the *registerP* are used in the complex multiplication. The complex product pair is subtracted from the *registerM* and rounded according to the rounding mode. The resulting binary 32 floating-point complex number pair is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- conjugate (cf. conjugate in section 6.10.2 on page 1418)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new conjugate = _ZX_1(conjugate);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  if (conjugate) {
    argument2 = fconj_32_32_x2(argument2);
  }
  new result1 = ffmsc_32_32_x2(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
```


Kalray – all rights reserved

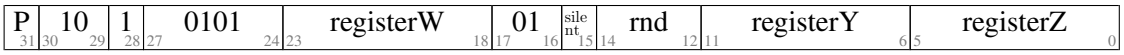
©2025 Kalray

```
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.66 FFMSWD Floating-Point Fused Multiply-Subtract Words to Double Word

Encoding Format MAU_FDDDD (Section 6.12.372), exucode2 = 0101



Syntax `ffmswd`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 32 floating-point numbers, and the product is subtracted from the *registerW*, assuming binary 64 floating-point format. The result is rounded according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW - (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_32_64(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.67 FFMSWDP Floating-Point Fused Multiply-Subtract Words to Double Word Pair

Encoding Format MAU_FQQDD (Section 6.12.377), exucode2 = 0010

P	10	1	0010	registerM	0	11	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 6	5 0

Syntax `ffmswdp`*roundingsilent registerM* = *registerZ*, *registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming pairs of binary 32 floating-point numbers, and the pair of products is subtracted from the *registerM*, assuming a pair of binary 64 floating-point format. The results are rounded according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM + (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_32_64_x2(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.68 FFMSWP Floating-Point Fused Multiply-Subtract Word Pair

Encoding Format MAU_FMAWP (Section 6.12.323), exucode2 = 0011

P	10	0	0011	registerW	00	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `ffmswp`*roundingsilent registerW = registerZ, registerY*

Description The *registerY* is multiplied by the *registerZ*, assuming binary 32 floating-point number pairs, and the product pair is subtracted from the *registerW*. The resulting pair is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerW = registerW - (registerY \cdot registerZ)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = GPR[registerW];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_32_32_x2(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.69 FFMSWQ Floating-Point Fused Multiply-Subtract Word Quadruple

Encoding Format MAU_FMAWQ (Section 6.12.325), exucode2 = 0111

P	10	0	0111	registerM	0	00	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `ffmswq`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* is multiplied by the *registerP*, assuming binary 32 floating-point number quadruples, and the product quadruple is subtracted from the *registerM*. The resulting quadruple is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

$$registerM = registerM - (registerO \cdot registerP)$$

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = ffms_32_32_x4(RM, argument1, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.70 FIXEDD Floating Point Conversion to Fixed-Point Double

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 0110

P	10	0	0110	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fixedd`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 64-bit floating-point number is multiplied by (1<<*unsigned6*) then converted to a 64-bit signed integer according to the rounding mode *rounding*. This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fixed_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.71 FIXEDUD Floating Point Conversion to Unsigned Fixed-Point Double

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 0111

P	10	0	0111	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fixedud`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 64-bit floating-point number is multiplied by (1<<*unsigned6*) then converted to a 64-bit unsigned integer according to the rounding mode *rounding*. This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

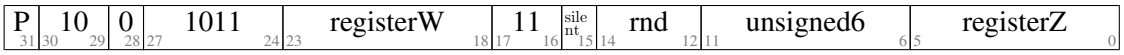
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fixedu_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.72 FIXEDUW Floating Point Conversion to Unsigned Fixed-Point

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1011



Syntax `fixeduw`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit floating-point number is multiplied by (1<<*unsigned6*) then converted to a 32-bit unsigned integer according to the rounding mode *rounding*. This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fixedu_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.73 FIXEDUWP Floating Point Conversion to Unsigned Fixed-Point Word Pair

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1111

P	10	0	1111	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fixeduwp`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit floating-point number pair is multiplied by (1<<*unsigned6*) then converted to a 32-bit unsigned integer pair according to the rounding mode *rounding*. This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fixedu_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.74 FIXEDW — Floating Point Conversion to Fixed-Point

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1010

P	10	0	1010	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fixedw`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit floating-point number is multiplied by (1<<*unsigned6*) then converted to a 32-bit signed integer according to the rounding mode *rounding*. This instruction may raise inexact or invalid exception bits in the CS register. As this instructions belongs to the 32-bit ISA subset, the 32 upper bits of the destination register are cleared even if the resulting word is negative.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fixed_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.75 FIXEDWP Floating Point Conversion to Fixed-Point Word Pair

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1110

P	10	0	1110	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fixedwp`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit floating-point number pair is multiplied by (1<<*unsigned6*) then converted to a 32-bit signed integer pair according to the rounding mode *rounding*. This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fixed_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.76 FLOATD Floating Point Conversion from Fixed-Point Double

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 0100

P	10	0	0100	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `floatd`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 64-bit signed integer is converted to a 64-bit floating-point number according to the rounding mode *rounding* and then divided by (1≪*unsigned6*). This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = float_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.77 FLOATUD Floating Point Conversion from Unsigned Fixed-Point Double

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 0101

P	10	0	0101	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `floatud`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 64-bit unsigned integer is converted to a 64-bit floating-point number according to the rounding mode *rounding* and then divided by (1<<*unsigned6*). This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = floatu_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.78 FLOATUW Floating Point Conversion from Unsigned Fixed-Point

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1001

P	10	0	1001	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `floatuw`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit unsigned integer is converted to a 32-bit floating-point number according to the rounding mode *rounding* and then divided by (1≪*unsigned6*). This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = floatu_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.79 FLOATUWP Floating Point Conversion from Unsigned Fixed-Point Word Pair

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1101

P	10	0	1101	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `floatuwp`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit unsigned integer pair is converted to a 32-bit floating-point number pair according to the rounding mode *rounding* and then divided by (1≪*unsigned6*). This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = floatu_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.80 FLOATW — Floating Point Conversion from Fixed-Point

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1000

P	10	0	1000	registerW	11	silent	rnd	unsigned6	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `floatw`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit signed integer is converted to a 32-bit floating-point number according to the rounding mode *rounding* and then divided by (1≪*unsigned6*). This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

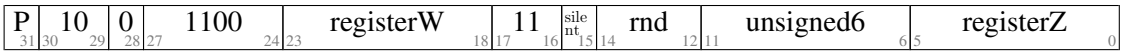
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = float_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.81 FLOATWP Floating Point Conversion from Fixed-Point Word Pair

Encoding Format MAU_FDD (Section 6.12.347), exucode2 = 1100



Syntax `floatwp`*roundingsilent registerW* = *registerZ*, *unsigned6*

Description The *registerZ* interpreted as a 32-bit signed integer pair is converted to a 32-bit floating-point number pair according to the rounding mode *rounding* and then divided by (1≪*unsigned6*). This instruction may raise inexact or invalid exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new argument3 = _ZX_6(unsigned6);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = float_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.82 FMAXD — Floating-Point Maximum Double Word

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0001

P	11	1	0001	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fmaxd` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming binary 64 floating-point numbers. The maximum is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fmaxNum_64(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.83 FMAXHQ Floating-Point Maximum Half Word Quadruple

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0111

P	11	1	0111	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fmaxhq` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming quadruple of binary 16 floating-point numbers. The quadruple of maximum values is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fmaxNum_16_x4(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.84 FMAXW — Floating-Point Maximum Word

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0011

P	11	1	0011	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fmaxw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming binary 32 floating-point numbers. The maximum is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fmaxNum_32(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.85 FMAXWP Floating-Point Maximum Word Pair

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0101

P	11	1	0101	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fmaxwp` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming pairs of binary 32 floating-point numbers. The pair of maximum values is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fmaxNum_32_x2(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.86 FMIND Floating-Point Minimum Double Word

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0000

P	11	1	0000	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fmind` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming binary 64 floating-point numbers. The minimum is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fminNum_64(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.87 FMINHQ Floating-Point Minimum Half Word Quadruple

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0110

P	11	1	0110	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fminhq` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming quadruple of binary 16 floating-point numbers. The quadruple of minimum values is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fminNum_16_x4(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.88 FMINW — Floating-Point Minimum Word

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0010

P	11	1	0010	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fminw` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming binary 32 floating-point numbers. The minimum is stored into the *registerW*.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fminNum_32(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.89 FMINWP Floating-Point Minimum Word Pair

Encoding Format ALU_DFWRD (Section 6.12.304), exucode2 = 0100

P	11	1	0100	registerW	11	1000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Syntax `fminwp` *registerW* = *registerZ*, *registerY*

Description The *registerZ* is compared to the *registerY*, assuming pairs of binary 32 floating-point numbers. The pair of minimum values is stored into the *registerW*.

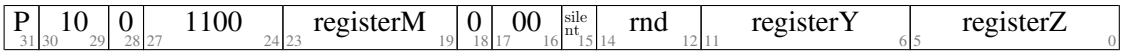
Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new result1 = fminNum_32_x2(argument2, argument3);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.90 FMM212W Floating-Point Matrix Multiply 2x1x2 Word

Encoding Format MAU_FMMW (Section 6.12.330), exucode2 = 1100



Syntax `fmm212w`*roundingsilent registerM = registerZ, registerY*

Description The *registerM* is considered as a 2×2 matrix of binary 32 floating-point numbers packed into 128 bits. The *registerY* and the *registerZ* are considered as 2-element vectors of 32 floating-point numbers. The matrix multiplication between the *registerZ* words and the *registerY* words is computed using one product per element. The resulting four binary 32 floating-point numbers are stored into the *registerM* in row-major order. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

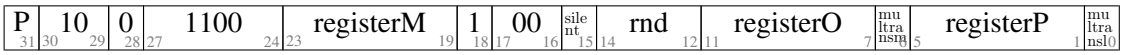
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmm212_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.91 FMM222W Floating-Point Matrix Multiply 2x2x2 Word

Encoding Format MAU_FMMWP (Section 6.12.332), exucode2 = 1100



Syntax `fmm222w`*multransm_multransl**rounding**silent registerM* = *registerP*, *registerO*

Description The *registerM*, the *registerO* and the *registerP* are considered as a 2×2 matrix of binary 32 floating-point numbers packed into 128 bits. The *registerP* and the *registerO* are assumed row-major or column-major layout, as respectively specified by bits 0 and 1 of the *multransm_multransl*. The matrix multiplication between the *registerP* words and the *registerO* words is computed using a 2-deep dot product per element. The resulting four binary 32 floating-point numbers are stored into the *registerM* in row-major order. Each dot product is rounded once according to the rounding mode. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- multransm_multransl (cf. transpose in section 6.10.2 on page 1431)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

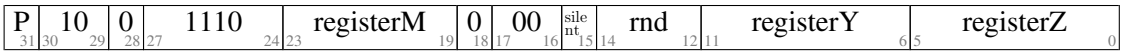
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new transpose = _ZX_2(multransm_multransl);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmm222_32_32(RM, argument2, argument3, transpose);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.92 FMMA212W Floating-Point Matrix Multiply Add 2x1x2 Word

Encoding Format MAU_FMMAW (Section 6.12.331), exucode2 = 1110



Syntax `fmma212w`*roundingsilent registerM = registerZ, registerY*

Description The *registerM* is considered as a 2×2 matrix of binary 32 floating-point numbers packed into 128 bits. The *registerY* and the *registerZ* are considered as 2-element vectors of 32 floating-point numbers. The matrix multiplication between the *registerZ* words and the *registerY* words is computed using one product per element. The resulting four binary 32 floating-point numbers are added to the *registerM* elements in row-major order. The sum of each product and the corresponding element of the *registerM* is rounded according to the rounding mode. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

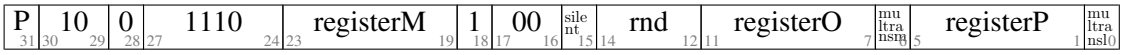
```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmma212_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.93 FMMA222W Floating-Point Matrix Multiply Add 2x2x2 Word

Encoding Format MAU_FMMAWP (Section 6.12.333), exucode2 = 1110



Syntax `fmma222w`*multransm_multransl**rounding**silent registerM* = *registerP*, *registerO*

Description The *registerM*, the *registerO* and the *registerP* are considered as a 2×2 matrix of binary 32 floating-point numbers packed into 128 bits. The *registerP* and the *registerO* are assumed row-major or column-major layout, as respectively specified by bits 0 and 1 of the *multransm_multransl*. The matrix multiplication between the *registerP* words and the *registerO* words is computed using a 2-deep dot product per element. The resulting four binary 32 floating-point numbers are added to the *registerM* elements in row-major order. The sum of each dot product and the corresponding element of the *registerM* is rounded according to the rounding mode. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- multransm_multransl (cf. transpose in section 6.10.2 on page 1431)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new transpose = _ZX_2(multransm_multransl);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmma222_32_32(RM, argument2, argument3, argument1, transpose)
      ;
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
```


Kalray – all rights reserved

```
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.94 FMMS212W Floating-Point Matrix Multiply Subtract 2x1x2 Word

Encoding Format MAU_FMMAW (Section 6.12.331), exucode2 = 1111

P	10	0	1111	registerM	0	00	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 6	5 0

Syntax `fmms212w`*roundingsilent registerM = registerZ, registerY*

Description The *registerM* is considered as a 2×2 matrix of binary 32 floating-point numbers packed into 128 bits. The *registerY* and the *registerZ* are considered as 2-element vectors of 32 floating-point numbers. The matrix multiplication between the *registerZ* words and the *registerY* words is computed using one product per element. The resulting four binary 32 floating-point numbers are subtracted from the *registerM* elements in row-major order. The sum of each negated product and the corresponding element of the *registerM* is rounded according to the rounding mode. The resulting four binary 32 floating-point numbers are stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

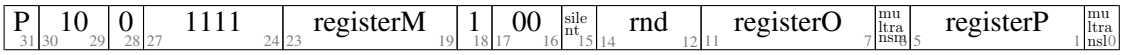
Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmms212_32_32(RM, argument2, argument3, argument1);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.95 FMMS222W Floating-Point Matrix Multiply Subtract 2x2x2 Word

Encoding Format MAU_FMMAWP (Section 6.12.333), exucode2 = 1111



Syntax `fmms222w`*multransm_multransl*`rounding`*silent registerM* = *registerP*, *registerO*

Description The *registerM*, the *registerO* and the *registerP* are considered as a 2×2 matrix of binary 32 floating-point numbers packed into 128 bits. The *registerP* and the *registerO* are assumed row-major or column-major layout, as respectively specified by bits 0 and 1 of the *multransm_multransl*. The matrix multiplication between the *registerP* words and the *registerO* words is computed using a 2-deep dot product per element. The resulting four binary 32 floating-point numbers subtracted from to the *registerM* elements in row-major order. The sum of each negated dot product and the corresponding element of the *registerM* is rounded according to the rounding mode. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- multransm_multransl (cf. transpose in section 6.10.2 on page 1431)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU_AUXR (Section 6.14.45)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new transpose = _ZX_2(multransm_multransl);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new argument1 = PGR[registerM];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmms222_32_32(RM, argument2, argument3, argument1, transpose)
      ;
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
```


Kalray – all rights reserved

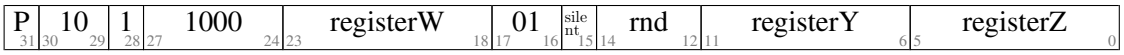
©2025 Kalray

```
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.96 FMULD — Floating-Point Multiply Double Word

Encoding Format MAU_FDDD (Section 6.12.373), exucode2 = 1000



Syntax `fmuld`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 64 floating-point numbers. The result is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_64_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.97 FMULHO Floating-Point Multiply Half Word Octuple

Encoding Format MAU_FQQQ2 (Section 6.12.379), exucode2 = 0101

P	10	1	0101	registerM	1	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fmulho`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerP* is multiplied by the *registerO*, assuming binary 16 floating-point number octuples. The resulting octuple is rounded to the binary 16 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_16_16_x8(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.98 FMULHQ Floating-Point Multiply Half Word Quadruple

Encoding Format MAU_FDDD (Section 6.12.373), exucode2 = 1011

P	10	1	1011	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fmulhq`*roundingsilent registerW = registerZ, registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 16 floating-point number quadruples. The results are rounded to the binary 16 floating-point format according to the rounding mode set in the CS register, packed and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_16_16_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.99 FMULHW Floating-Point Multiply Half Words to Word

Encoding Format MAU_FSSS (Section 6.12.376), exucode2 = 1111

P	10	1	1111	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fmulhw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 16 floating-point numbers. The result is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_16_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.100 FMULHWQ Floating-Point Multiply Half Word to Word Quadruple

Encoding Format MAU_FQDD (Section 6.12.378), exucode2 = 0001

P	10	1	0001	registerM	1	11	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 6	5 0

Syntax `fmulhwq`*roundingsilent registerM = registerZ, registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 16 floating-point number quadruples. The results are rounded to the binary 32 floating-point format according to the rounding mode set in the CS register, packed and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

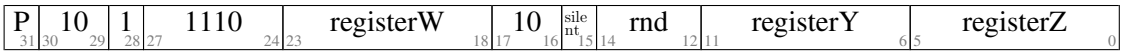
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_16_32_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.101 FMULW Floating-Point Multiply Words

Encoding Format MAU_FSSS (Section 6.12.376), exucode2 = 1110



Syntax `fmulw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 32 floating-point numbers. The result is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

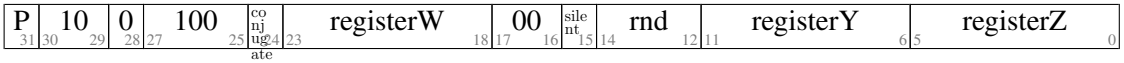
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.102 FMULWC Floating-Point Multiply Word Complex

Encoding Format MAU_FMULCW (Section 6.12.327), exucode3 = 100



Syntax `fmulwc`*conjugateroundingsilent registerW = registerZ, registerY*

Description The *registerY* and the *registerZ* are considered as binary 32 floating-point complex numbers. These numbers are multiplied according to the rounding mode set in the CS register. If the *conjugate* is set, the complex conjugate of the *registerZ* is used in the complex multiplication. The resulting binary 32 floating-point complex number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- conjugate (cf. conjugate in section 6.10.2 on page 1418)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

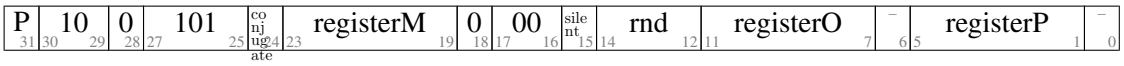
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new conjugate = _ZX_1(conjugate);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  if (conjugate) {
    argument2 = fconj_32_32(argument2);
  }
  new result1 = fmulc_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.103 FMULWCP Floating-Point Multiply Word Complex Pair

Encoding Format MAU_FMULWCP (Section 6.12.328), exucode3 = 101



Syntax `fmulwcp`*conjugateroundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of binary 32 floating-point complex numbers. The low half of the *registerP* is multiplied by the low half of the *registerO* to produce a low complex product. The high half of the *registerP* is multiplied by the high half of the *registerO* to produce a high complex product. If the *conjugate* is set, the complex conjugates of the *registerP* are used in the complex multiplication. The resulting binary 32 floating-point complex number pair is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- conjugate (cf. conjugate in section 6.10.2 on page 1418)
- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
  new conjugate = _ZX_1(conjugate);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  if (conjugate) {
    argument2 = fconj_32_32_x2(argument2);
  }
  new result1 = fmulc_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
```


Kalray – all rights reserved

```
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.104 FMULWD Floating-Point Multiply Words to Double Word

Encoding Format MAU_FDDD (Section 6.12.373), exucode2 = 1001

P	10	1	1001	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fmulwd`*roundingsilent registerW = registerZ, registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 32 floating-point numbers. The result is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_32_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.105 FMULWDP Floating-Point Multiply Word to Double Word Pair

Encoding Format MAU_FQDD (Section 6.12.378), exucode2 = 0000

P	10	1	0000	registerM	1	11	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 6	5 0

Syntax `fmulwdp`*roundingsilent registerM = registerZ, registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 32 floating-point number pairs. The resulting pair is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_32_64_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.106 FMULWP Floating-Point Multiply Word Pair

Encoding Format MAU_FDDD (Section 6.12.373), exucode2 = 1010

P	10	1	1010	registerW	01	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fmulwp`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is multiplied by the *registerY*, assuming binary 32 floating-point number pairs. The resulting pair is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

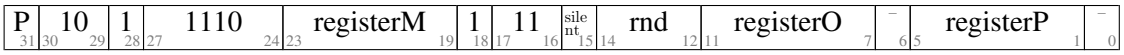
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.107 FMULWQ Floating-Point Multiply Word Quadruple

Encoding Format MAU_FQQQ1 (Section 6.12.382), exucode2 = 1110



Syntax `fmulwq`*roundingsilent registerM = registerP, registerO*

Description The *registerP* is multiplied by the *registerO*, assuming binary 32 floating-point number quadruples. The resulting quadruple is rounded to the binary 32 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fmul_32_32_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.108 FNARROWDW Floating Point Narrow Double Word to Word

Encoding Format ALU_FWRW (Section 6.12.302), exucode2 = 1000

P 31	11 30 29	1 28	1000 27 24	registerW 23 18	11 17 16	0110 15 12	silent2 11	rounding2 10 8	– – 7 6	registerZ 5 0

Syntax `fnarrowdw`*rounding2silent2 registerW = registerZ*

Description The *registerZ* interpreted as a binary 64 floating-point number is converted to a binary 32 floating-point number according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding2 (cf. rounding in section 6.10.2 on page 1424)
- silent2 (cf. silent in section 6.10.2 on page 1428)

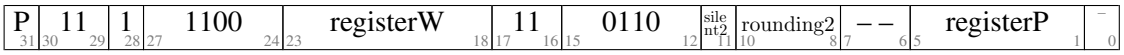
Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage ID:
  new silent = _ZX_1(silent2);
  new rounding = _ZX_3(rounding2);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fnarrow_64_32(RM, argument2);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.UN = funderflow() | CS_un;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.109 FNARROWDWP Floating Point Narrow Double to Word Pair

Encoding Format ALU_WPFWR (Section 6.12.303), exucode2 = 1100



Syntax `fnarrowdwp`*rounding2silent2 registerW = registerP*

Description The *registerP* is interpreted as a pair of binary 64 floating-point numbers, which are converted to binary 32 floating-point numbers according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding2 (cf. rounding in section 6.10.2 on page 1424)
- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage ID:
  new silent = _ZX_1(silent2);
  new rounding = _ZX_3(rounding2);
stage RR:
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fnarrow_64_32_x2(RM, argument2);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.UN = funderflow() | CS_un;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.110 FNARROWWH Floating Point Narrow Word to Half Word

Encoding Format ALU_FWRW (Section 6.12.302), exucode2 = 1010

P	11	1	1010	registerW	11	0110	silent2	rounding2	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fnarrowwh`*rounding2silent2 registerW = registerZ*

Description The *registerZ* interpreted as a binary 32 floating-point number is converted to a binary 16 floating-point number according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding2 (cf. rounding in section 6.10.2 on page 1424)
- silent2 (cf. silent in section 6.10.2 on page 1428)

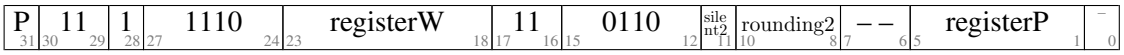
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
  new rounding = _ZX_3(rounding2);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fnarrow_32_16(RM, argument2.32[0]);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.UN = funderflow() | CS_un;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.111 FNARROWWHQ Floating Point Narrow Word to Half Word Quadruple

Encoding Format ALU_WPFWR (Section 6.12.303), exucode2 = 1110



Syntax `fnarrowwhq`*rounding2silent2 registerW = registerP*

Description The *registerP* is interpreted as a quadruple of binary 32 floating-point numbers, which are converted to binary 16 floating-point numbers according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding2 (cf. rounding in section 6.10.2 on page 1424)
- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
  new rounding = _ZX_3(rounding2);
stage RR:
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fnarrow_32_16_x4(RM, argument2);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.UN = funderflow() | CS_un;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

©2025 Kalray

6.8.112 FNEGD Floating-Point Negate Double Word

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0000

P	11	1	0000	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fnegd` *registerW* = *registerZ*

Description The negated value the *registerZ*, assuming binary 64 floating-point number, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ 0x8000000000000000;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.113 FNEGHQ Floating-Point Negate Half Word Quadruple

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0110

P	11	1	0110	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fneghq` *registerW* = *registerZ*

Description The negated values of the *registerZ*, assuming a quadruple of binary 16 floating-point numbers, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ 0x8000800080008000;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.8.114 FNEGW Floating-Point Negate Word

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0010

P	11	1	0010	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fnegw` *registerW* = *registerZ*

Description The negated value the *registerZ*, assuming binary 32 floating-point number, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = _ZX_32(argument2 ^ 0x80000000);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.115 FNEGWP Floating-Point Negate Word Pair

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 0100

P	11	1	0100	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fnegwp` *registerW* = *registerZ*

Description The negated values of the *registerZ*, assuming a pair of binary 32 floating-point numbers, is stored into the *registerW*. Implemented as a bitwise operation, does not raise FP exceptions.

Execution Reservation table ALU_TINY (Section 6.14.11)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = argument2 ^ 0x8000000080000000;
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.116 FRECW Floating-Point Reciprocal of Word

Encoding Format ALU_FWRWLL (Section 6.12.301), exucode2 = 0010

P	11	1	0010	registerW	11	0110	silent2	rounding2	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `frecw`*rounding2silent2 registerW = registerZ*

Description The reciprocal of the *registerZ*, assuming binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- rounding2 (cf. rounding in section 6.10.2 on page 1424)
- silent2 (cf. silent in section 6.10.2 on page 1428)

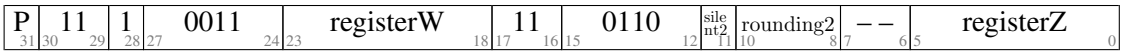
Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage ID:
  new silent = _ZX_1(silent2);
  new rounding = _ZX_3(rounding2);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = frec_32(RM, argument2);
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.UN = funderflow() | CS_un;
    CS.OV = foverflow() | CS_ov;
  }
stage SF:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.117 FRSRW Floating-Point Reciprocal of Square Root Word

Encoding Format ALU_FWRWLL (Section 6.12.301), exucode2 = 0011



Syntax `frsrw`*rounding2silent2 registerW = registerZ*

Description The reciprocal of square root of the *registerZ*, assuming binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- rounding2 (cf. rounding in section 6.10.2 on page 1424)
- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_FULL (Section 6.14.2)

```
stage ID:
  new silent = _ZX_1(silent2);
  new rounding = _ZX_3(rounding2);
stage RR:
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = frsq_32(RM, argument2);
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.UN = funderflow() | CS_un;
    CS.OV = foverflow() | CS_ov;
  }
stage SF:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.118 FSBFD Floating-Point Subtract From Double Word

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0100

P	10	1	0100	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fsbfd`*roundingsilent registerW = registerZ, registerY*

Description The *registerZ* is subtracted from the *registerY*, assuming binary 64 floating-point numbers. The result is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

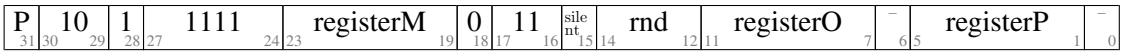
Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbf_64_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.119 FSBFDC.C Floating-Point Subtract From Double Word Complex Conjugate

Encoding Format MAU_FQQQ0 (Section 6.12.381), exucode2 = 1111



Syntax `fsbfdc.c`*roundingsilent registerM = registerP, registerO*

Description The *registerO* and the *registerP* are considered as binary 64 floating-point complex numbers. The conjugate of the *registerP* is subtracted from the *registerO*, according to the rounding mode set in the CS register. The resulting binary 64 floating-point complex number is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbfcc_64_64(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.120 FSBFDP Floating-Point Subtract Word to Double Word Pair

Encoding Format MAU_FQQQ0 (Section 6.12.381), exucode2 = 1110

P	10	1	1110	registerM	0	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fsbfdp`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerP* is subtracted from the *registerO*, assuming binary 64 floating-point number pairs. The resulting pair is rounded to the binary 64 floating-point format according to the rounding mode set in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbf_64_64_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.121 FSBFHO Floating-Point Subtract From Half Word Octuple

Encoding Format MAU_FQQQ2 (Section 6.12.379), exucode2 = 0111

P	10	1	0111	registerM	1	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fsbfho`*roundingsilent registerM = registerP, registerO*

Description The *registerP* is subtracted from the *registerO*, assuming binary 16 floating-point number octuples. The resulting octuple is rounded to the binary 16 floating-point format according to the rounding mode stored in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbf_16_16_x8(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.122 FSBFHQ Floating-Point Subtract From Half Word Quadruple

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0110

P	10	1	0110	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fsbfhq`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY*, assuming binary 16 floating-point number quadruples. The results are rounded to the binary 16 floating-point format according to the rounding mode set in the CS register, packed and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbf_16_16_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.123 FSBFW Floating-Point Subtract From Words

Encoding Format MAU_FSSS (Section 6.12.376), exucode2 = 1101

P	10	1	1101	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fsbfw`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY*, assuming binary 32 floating-point numbers. The result is rounded to the binary 32 floating-point format according to the rounding mode stored in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbf_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.124 FSBFWC.C Floating-Point Subtract From Word Complex Conjugate

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0111

P	10	1	0111	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fsbfwc.c`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerY* and the *registerZ* are considered as binary 32 floating-point complex numbers. The conjugate of the *registerZ* is subtracted from the *registerY*, according to the rounding mode set in the CS register. The resulting binary 32 floating-point complex number is stored back into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbfcc_32_32(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.125 FSBFWCP.C Floating-Point Subtract From Word Complex Pair Conjugate

Encoding Format MAU_FQQQ1 (Section 6.12.382), exucode2 = 1011

P	10	1	1011	registerM	1	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fsbfwcp.c`*roundingsilent registerM* = *registerP*, *registerO*

Description The *registerO* and the *registerP* are considered as pairs of binary 32 floating-point complex numbers. The conjugate of the *registerP* pair is subtracted from the *registerO* pair, according to the rounding mode set in the CS register. The resulting pair of binary 32 floating-point complex numbers is stored back into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbfcc_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.126 FSBFWP Floating-Point Subtract From Word Pair

Encoding Format MAU_FDDD1 (Section 6.12.374), exucode2 = 0101

P	10	1	0101	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `fsbfwp`*roundingsilent registerW* = *registerZ*, *registerY*

Description The *registerZ* is subtracted from the *registerY*, assuming binary 32 floating-point number pairs. The resulting pair is rounded to the binary 32 floating-point format according to the rounding mode stored in the CS register and stored into the *registerW*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = GPR[registerY];
  new argument2 = GPR[registerZ];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbf_32_32_x2(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.127 FSBFWQ Floating-Point Subtract From Word Quadruple

Encoding Format MAU_FQQQ1 (Section 6.12.382), exucode2 = 1010

P	10	1	1010	registerM	1	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Syntax `fsbfwq`*roundingsilent registerM = registerP, registerO*

Description The *registerP* is subtracted from the *registerO*, assuming binary 32 floating-point number quadruples. The resulting quadruple is rounded to the binary 32 floating-point format according to the rounding mode stored in the CS register and stored into the *registerM*. This instruction may raise inexact, invalid, overflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table MAU (Section 6.14.42)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = PGR[registerO];
  new argument2 = PGR[registerP];
  new RM = rounding == 7 ? CS.RM : rounding;
  new result1 = fsbf_32_32_x4(RM, argument2, argument3);
stage E4:
  if (silent == 0) {
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    new CS_un = CS.UN;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
    CS.UN = funderflow() | CS_un;
  }
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.8.128 FSDIVD Floating-Point Seed for Division Double Word

Encoding Format ALU_DFWRR (Section 6.12.300), exucode2 = 0000

P	11	1	0000	registerW	11	0101	silent2	– – –	– –	registerP	–
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 1	0

Syntax `fsdivd`*silent2 registerW = registerP*

Description An approximate value of the division of the least significant half by the most significant half of the *registerP*, interpreted as binary 64 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = PGR[registerP];
  new argument2_0 = _ZX_64(argument2);
  new argument2_1 = _ZX_64(argument2 >> 64);
  new result1 = fsdiv_64(argument2_0, argument2_1);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.129 FSDIVW — Floating-Point Seed for Word Division

Encoding Format ALU_DFWRR (Section 6.12.300), exucode2 = 0010

P	11	1	0010	registerW	11	0101	silent2	– – –	– –	registerP	–
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 1	0

Syntax `fsdivw`*silent2 registerW = registerP*

Description An approximate value of the division of the least significant half by the most significant half of the *registerP*, interpreted as binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = PGR[registerP];
  new argument2_0 = argument2.32[0];
  new argument2_1 = argument2.32[2];
  new result1 = fsdiv_32(argument2_0, argument2_1);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.130 FSDIVWP Floating-Point Seed for Division Word Pair

Encoding Format ALU_DFWRR (Section 6.12.300), exucode2 = 0100

P	11	1	0100	registerW	11	0101	silent2	– – –	– –	registerP	–
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 1	0

Syntax `fsdivwp`*silent2 registerW = registerP*

Description The pair of values of the special case for the division of the least significant half by the most significant half of the *registerP*, interpreted as pairs of binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

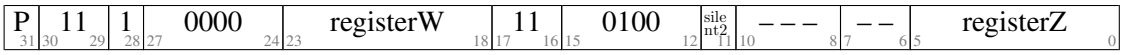
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = PGR[registerP];
  new argument2_0 = _ZX_64(argument2);
  new argument2_1 = _ZX_64(argument2 >> 64);
  new result1 = fsdiv_32_x2(argument2_0, argument2_1);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.131 FSRECD Floating-Point Seed for Reciprocal Double Word

Encoding Format ALU_HQFWR (Section 6.12.299), exucode2 = 0000



Syntax `fsrecd`*silent2 registerW = registerZ*

Description An approximation of the reciprocal of the *registerZ*, assuming binary 64 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fsrec_64(argument2);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.132 FSRECW Floating-Point Seed for Reciprocal Word

Encoding Format ALU_HQFWR (Section 6.12.299), exucode2 = 0010

P	11	1	0010	registerW	11	0100	silent2	− − −	− −	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fsrecw`*silent2 registerW = registerZ*

Description An approximation of the reciprocal of the *registerZ*, assuming binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

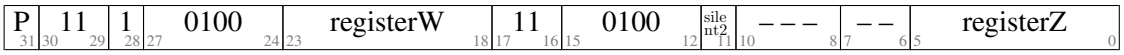
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fsrec_32(argument2);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.133 FSRECWP Floating-Point Seed for Reciprocal on Word Pair

Encoding Format ALU_HQFWR (Section 6.12.299), exucode2 = 0100



Syntax `fsrecwp`*silent2 registerW = registerZ*

Description An approximation of the reciprocal of the *registerZ*, assuming a pair of binary 32 floating-point numbers, is stored into the *registerW*. This instruction may raise division by zero, inexact, invalid or overflow exception bits in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fsrec_32_x2(argument2);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_dz = CS.DZ;
    new CS_in = CS.IN;
    new CS_io = CS.IO;
    new CS_ov = CS.OV;
    CS.DZ = fdivbyzero() | CS_dz;
    CS.IN = finexact() | CS_in;
    CS.IO = finvalid() | CS_io;
    CS.OV = foverflow() | CS_ov;
  }
```


Kalray – all rights reserved

6.8.134 FSRSRD Floating-Point Seed for Reciprocal Square Root Double Word

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 1000

P	11	1	1000	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fsrsrd` *registerW* = *registerZ*

Description An approximation of the reciprocal of square root of the *registerZ*, assuming binary 64 floating-point numbers, is stored into the *registerW*.

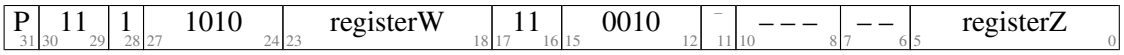
Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fsrsr_64(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.135 FSRSRW Floating-Point Seed for Reciprocal Square Root

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 1010



Syntax `fsrsrw` *registerW* = *registerZ*

Description An approximation of the reciprocal of square root of the *registerZ*, assuming binary 32 floating-point numbers, is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fsrsr_32(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.136 FSRSRWP Floating-Point Seed for Reciprocal Square Root on Word Pair

Encoding Format ALU_DFWR (Section 6.12.297), exucode2 = 1100

P	11	1	1100	registerW	11	0010	–	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fsrsrwp` *registerW* = *registerZ*

Description An approximation of the reciprocal of square root of the *registerZ*, assuming a pair of binary 32 floating-point numbers, is stored into the *registerW*.

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fsrsr_32_x2(argument2);
stage E1:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.8.137 FWIDENLHW Floating Point Widen Least Significant Half Word to Word

Encoding Format ALU_FWR (Section 6.12.298), exucode2 = 1010

P	11	1	1010	registerW	11	0011	silent2	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fwidenlhw`*silent2 registerW = registerZ*

Description The least significant half of the *registerZ* interpreted as a binary 16 floating-point number is converted to a binary 32 floating-point number and stored into the least significant word of the *registerW*. This instruction may raise invalid exception bit in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fwiden_16_32(argument2.16[0]);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_io = CS.IO;
    CS.IO = finvalid() | CS_io;
  }
```


Kalray – all rights reserved

6.8.138 FWIDENLHWP Floating Point Widen Least Significant Half Words to Word Pair

Encoding Format ALU_FWR (Section 6.12.298), exucode2 = 1100

Syntax `fwidenlhwp`*silent2 registerW = registerZ*

Description The least significant word of the *registerZ* interpreted as a half word pair. Each half word is interpreted as a binary 16 floating-point number and converted to a binary 32 floating-point number. These words are paired and stored into the *registerW*. This instruction may raise invalid exception bit in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fwiden_16_32_x2(argument2.32[0]);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_io = CS.IO;
    CS.IO = finvalid() | CS_io;
  }
```


Kalray – all rights reserved

6.8.139 FWIDENLWD Floating Point Widen Least Significant Word to Double

Encoding Format ALU_FWR (Section 6.12.298), exucode2 = 1000

P	11	1	1000	registerW	11	0011	sile nt2	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fwidenlwd`*silent2 registerW = registerZ*

Description The *registerZ* interpreted as a binary 32 floating-point number in the least significant word is converted to a binary 64 floating-point number and stored into the *registerW*. This instruction may raise invalid exception bit in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fwiden_32_64(argument2.32[0]);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_io = CS.IO;
    CS.IO = finvalid() | CS_io;
  }
```


Kalray – all rights reserved

6.8.140 FWIDENMHW Floating Point Widen Most Significant Half Word to Word

Encoding Format ALU_FWR (Section 6.12.298), exucode2 = 1011

P	11	1	1011	registerW	11	0011	silent2	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fwidenmhw`*silent2 registerW = registerZ*

Description The most significant half of the *registerZ* interpreted as a binary 16 floating-point number is converted to a binary 32 floating-point number and stored into the least significant word of the *registerW*. This instruction may raise invalid exception bit in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fwiden_16_32(_ZX_16(argument2 >> 16));
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_io = CS.IO;
    CS.IO = finvalid() | CS_io;
  }
```


Kalray – all rights reserved

6.8.141 FWIDENMHWP Floating Point Widen Most Significant Half Words to Word Pair

Encoding Format ALU_FWR (Section 6.12.298), exucode2 = 1101

Syntax `fwidenmhwp`*silent2 registerW = registerZ*

Description The most significant word of the *registerZ* interpreted as a half word pair. Each half word is interpreted as a binary 16 floating-point number and converted to a binary 32 floating-point number. These words are paired and stored into the *registerW*. This instruction may raise invalid exception bit in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fwiden_16_32_x2(_ZX_32(argument2 >> 32));
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_io = CS.IO;
    CS.IO = finvalid() | CS_io;
  }
```


Kalray – all rights reserved

6.8.142 FWIDENMWD Floating Point Widen Most Significant Word to Double

Encoding Format ALU_FWR (Section 6.12.298), exucode2 = 1001

P	11	1	1001	registerW	11	0011	silent2	– – –	– –	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Syntax `fwidenmwd`*silent2 registerW = registerZ*

Description The *registerZ* interpreted as a binary 32 floating-point number in the most significant word is converted to a binary 64 floating-point number and stored into the *registerW*. This instruction may raise invalid exception bit in the CS register.

Modifier(s)

- silent2 (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table ALU_LITE (Section 6.14.5)

```
stage ID:
  new silent = _ZX_1(silent2);
stage RR:
  new argument2 = GPR[registerZ];
  new result1 = fwiden_32_64(argument2.32[1]);
stage E1:
  GPR[registerW] = result1;
stage E4:
  if (silent == 0) {
    new CS_io = CS.IO;
    CS.IO = finvalid() | CS_io;
  }
```


Kalray – all rights reserved

6.9 EXT Instructions

6.9.1 XANDNO Bitwise And with Not Operand Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0110

P	00	00	111	registerA	11	0110	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xandno` *registerA* = *registerB*, *registerC*

Description Bitwise and of not the *registerB* with the *registerC*. The result is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = ~argument2 & argument3;
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.2 XANDO — Bitwise And Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0000

P	00	00	111	registerA	11	0000	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xando` *registerA* = *registerB*, *registerC*

Description Bitwise and of the *registerB* with the *registerC*. The result is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = argument2 & argument3;
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.3 XCLAMPWO — Clamp Word Octuple

Encoding Format EXT_CLAMPWO (Section 6.12.38), exucode4 = 11

P	00	00	101	registerA	11	----	registerC	registerD
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xclampwo` *registerA* = *registerC*, *registerD*

Description The *registerA*, *registerC*, *registerD* are interpreted as vectors of 8 signed integer words. The element-wise maximum between the *registerA* and the *registerC* is computed, producing a temporary vector. The element-wise minimum between this temporary vector and the *registerD* is computed, producing a result which is stored back into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument2 = XVR[registerC];
  new argument1 = XVR[registerA];
  new result1 = clamp_32_32_x8(argument1, argument2, argument3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.4 XCOPYO — Copy Extension Octuple Word

Encoding Format BCU_XCOPYO (Section 6.12.14), log2size = 000

P	00	00	001	registerA	00	0	000	registerC	------
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `xcopyo` *registerA* = *registerC*

Description The *registerA* operand receives the value of the *registerC* operand.

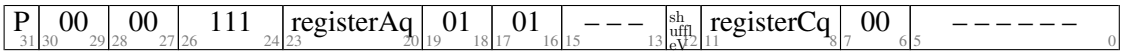
Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

```
stage RR:
  new argument2 = XVR[registerC];
  new result1 = argument2;
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.5 XCOPYV — Copy Word Vector

Encoding Format EXT_XCOPYV (Section 6.12.44), exucode4 = 01



Syntax `xcopyv`*shuffleV registerAq* = *registerCq*

Description The 4×4 double words that compose the *registerCq* are copied or transposed depending on the *shuffleV* and the result is stored back into the *registerAq*.

Modifier(s)

- shuffleV (cf. shuffleV in section 6.10.2 on page 1426)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new shuffleV = _ZX_1(shuffleV);
stage RR:
  new argument2_0 = XMR[registerCq]:0;
  new argument2_1 = XMR[registerCq]:1;
  new argument2_2 = XMR[registerCq]:2;
  new argument2_3 = XMR[registerCq]:3;
  new result1_0 = argument2_0;
  new result1_1 = argument2_1;
  new result1_2 = argument2_2;
  new result1_3 = argument2_3;
  if (shuffleV) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_1.64[0], argument2_2
        .64[0], argument2_3.64[0]);
    result1_1 = join_64_x4(argument2_0.64[1], argument2_1.64[1], argument2_2
        .64[1], argument2_3.64[1]);
    result1_2 = join_64_x4(argument2_0.64[2], argument2_1.64[2], argument2_2
        .64[2], argument2_3.64[2]);
    result1_3 = join_64_x4(argument2_0.64[3], argument2_1.64[3], argument2_2
        .64[3], argument2_3.64[3]);
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XMR[registerAq]:0 = result1_0;
  XMR[registerAq]:1 = result1_1;
  XMR[registerAq]:2 = result1_2;
  XMR[registerAq]:3 = result1_3;
```


Kalray – all rights reserved

6.9.6 XCOPYX — Copy Word Hexadecuple

Encoding Format EXT_XCOPYX (Section 6.12.42), exucode4 = 00

P	00	00	111	registerAp	1	00	–	shuffleX	registerCp	0	– – – – – –
31	30 29	28 27	26 24	23 19	18	17 16	15	14 12	11 7	6	5 0

Syntax `xcopyx`*shuffleX registerAp = registerCp*

Description The eight double words that compose the *registerCp* are copied or shuffled depending on the *shuffleX* and the result is stored back into the *registerAp*.

Modifier(s)

- shuffleX (cf. shuffleX in section 6.10.2 on page 1427)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new shuffleX = _ZX_3(shuffleX);
stage RR:
  new argument2_0 = XTR[registerCp]:0;
  new argument2_1 = XTR[registerCp]:1;
  new result1_0 = argument2_0;
  new result1_1 = argument2_1;
  if (shuffleX == 1) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_1.64[0], argument2_0
        .64[1], argument2_1.64[1]);
    result1_1 = join_64_x4(argument2_0.64[2], argument2_1.64[2], argument2_0
        .64[3], argument2_1.64[3]);
  }
  if (shuffleX == 2) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_0.64[2], argument2_1
        .64[0], argument2_1.64[2]);
    result1_1 = join_64_x4(argument2_0.64[1], argument2_0.64[3], argument2_1
        .64[1], argument2_1.64[3]);
  }
  if (shuffleX == 3) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_0.64[1], argument2_1
        .64[0], argument2_1.64[1]);
    result1_1 = join_64_x4(argument2_0.64[2], argument2_0.64[3], argument2_1
        .64[2], argument2_1.64[3]);
  }
  if (shuffleX == 4) {
    result1_0 = join_32_x8(argument2_0.32[0], argument2_0.32[2], argument2_0
        .32[1], argument2_0.32[3], argument2_1.32[0], argument2_1.32[2],
        argument2_1.32[1], argument2_1.32[3]);
    result1_1 = join_32_x8(argument2_0.32[4], argument2_0.32[6], argument2_0
        .32[5], argument2_0.32[7], argument2_1.32[4], argument2_1.32[6],
        argument2_1.32[5], argument2_1.32[7]);
```


Kalray – all rights reserved

```
  }
  if (shuffleX == 5) {
    result1_0 = join_32_x8(argument2_0.32[0], argument2_1.32[0], argument2_0
        .32[1], argument2_1.32[1], argument2_0.32[2], argument2_1.32[2],
        argument2_0.32[3], argument2_1.32[3]);
    result1_1 = join_32_x8(argument2_0.32[4], argument2_1.32[4], argument2_0
        .32[5], argument2_1.32[5], argument2_0.32[6], argument2_1.32[6],
        argument2_0.32[7], argument2_1.32[7]);
  }
  if (shuffleX == 6) {
    result1_0 = join_32_x8(argument2_0.32[0], argument2_0.32[2], argument2_0
        .32[4], argument2_0.32[6], argument2_1.32[0], argument2_1.32[2],
        argument2_1.32[4], argument2_1.32[6]);
    result1_1 = join_32_x8(argument2_0.32[1], argument2_0.32[3], argument2_0
        .32[5], argument2_0.32[7], argument2_1.32[1], argument2_1.32[3],
        argument2_1.32[5], argument2_1.32[7]);
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.7 XEORO Bitwise Exclusive Or Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0100

P	00	00	111	registerA	11	0100	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xeoro` *registerA* = *registerB*, *registerC*

Description Bitwise exclusive or of the *registerB* with the *registerC*. The result is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = argument2 ^ argument3;
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.8 XFFMA44HW Floating-Point Fused Multiply-Add 4x4 Half Words to Words

Encoding Format EXT_FFMAHW (Section 6.12.34), exucode4 = 10

P	00	00	100	registerAp	0	10	silent	rnd	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15	14 12	11 6	5 0

Syntax `xffma44hw`*roundingsilent registerAp* = *registerC*, *registerD*

Description The *registerC* and the *registerD* are interpreted as 4x4 matrices of binary 16 floating-point numbers, and the *registerAp* is interpreted as a 4x4 matrix of binary 32 floating-point numbers. Each element of the first two columns of the *registerC* is multiplied by the corresponding element of the *registerD* and the product is added to the corresponding elements of the *registerAp*. Each result is rounded and the resulting 4x4 matrix of words is stored back into the *registerAp*. This instruction may raise inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.32[0];
  new argument3_1 = argument3.32[1];
  new argument3_2 = argument3.32[2];
  new argument3_3 = argument3.32[3];
  new argument3_4 = argument3.32[4];
  new argument3_5 = argument3.32[5];
  new argument3_6 = argument3.32[6];
  new argument3_7 = argument3.32[7];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.32[0];
  new argument2_1 = argument2.32[1];
  new argument2_2 = argument2.32[2];
  new argument2_3 = argument2.32[3];
  new argument2_4 = argument2.32[4];
  new argument2_5 = argument2.32[5];
  new argument2_6 = argument2.32[6];
  new argument2_7 = argument2.32[7];
  new argument1_0 = XTR[registerAp]:0;
```


Kalray – all rights reserved

©2025 Kalray

```
  new argument1_1 = XTR[registerAp]:1;
  new XRM = rounding == 7 ? CS.XRM : rounding;
  new input3_0 = join_32_x4(argument3_0, argument3_2, argument3_4,
      argument3_6);
  new input2_0 = join_32_x4(argument2_0, argument2_2, argument2_4,
      argument2_6);
  new result1_0 = ffma_16_32_x8_reset(XRM, argument1_0, input2_0, input3_0);
  new input3_1 = join_32_x4(argument3_1, argument3_3, argument3_5,
      argument3_7);
  new input2_1 = join_32_x4(argument2_1, argument2_3, argument2_5,
      argument2_7);
  new result1_1 = ffma_16_32_x8_noreset(XRM, argument1_1, input2_1, input3_1)
      ;
stage E1:
  CS.XMF = 1;
stage E4:
  if (silent == 0) {
    new CS_xun = CS.XUN;
    new CS_xin = CS.XIN;
    new CS_xio = CS.XIO;
    new CS_xov = CS.XOV;
    CS.XUN = funderflow() | CS_xun;
    CS.XIN = finexact() | CS_xin;
    CS.XIO = finvalid() | CS_xio;
    CS.XOV = foverflow() | CS_xov;
  }
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.9 XFMAXHX Floating-Point Maximum Half Word Hexadecuple

Encoding Format EXT_FMINMAX (Section 6.12.37), exucode4 = 10

P	00	00	101	registerA	10	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xfmaxhx` *registerA* = *registerC*, *registerD*

Description The *registerC* is compared to the *registerD*, assuming hexadecuple of binary 16 floating-point numbers. The hexadecuple of maximum values is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument2 = XVR[registerC];
  new argument1 = XVR[registerA];
  new result1 = fmaxNum_16_x16(argument2, argument3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.10 XFMINHX Floating-Point Minimum Half Word Hexadecuple

Encoding Format EXT_FMINMAX (Section 6.12.37), exucode4 = 01

P	00	00	101	registerA	01	----	registerC	registerD
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xfminhx` *registerA* = *registerC*, *registerD*

Description The *registerC* is compared to the *registerD*, assuming hexadecuple of binary 16 floating-point numbers. The hexadecuple of minimum values is stored into the *registerA*.

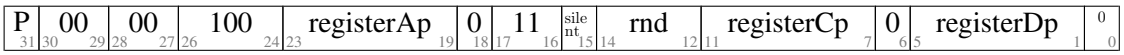
Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument2 = XVR[registerC];
  new argument1 = XVR[registerA];
  new result1 = fminNum_16_x16(argument2, argument3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.11 XFMMA484HWFloating-Point Matrix Multiply-Add 4x8x4 Half Words to Words

Encoding Format EXT_FMMAHW (Section 6.12.35), exucode4 = 11



Syntax `xfmma484hw`*roundingsilent registerAp* = *registerCp*, *registerDp*

Description The *registerAp* interpreted as a 4x4 matrix of words accumulates the product of the *registerCp* interpreted as a 4x8 matrix of half words by the transposed *registerDp* interpreted as a 4x8 matrix of half words.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3_0 = XTR[registerDp]:0;
  new argument3_0_0 = argument3_0.64[0];
  new argument3_0_1 = argument3_0.64[1];
  new argument3_0_2 = argument3_0.64[2];
  new argument3_0_3 = argument3_0.64[3];
  new argument3_1 = XTR[registerDp]:1;
  new argument3_1_0 = argument3_1.64[0];
  new argument3_1_1 = argument3_1.64[1];
  new argument3_1_2 = argument3_1.64[2];
  new argument3_1_3 = argument3_1.64[3];
  new argument2_0 = XTR[registerCp]:0;
  new argument2_0_0 = argument2_0.64[0];
  new argument2_0_1 = argument2_0.64[1];
  new argument2_0_2 = argument2_0.64[2];
  new argument2_0_3 = argument2_0.64[3];
  new argument2_1 = XTR[registerCp]:1;
  new argument2_1_0 = argument2_1.64[0];
  new argument2_1_1 = argument2_1.64[1];
  new argument2_1_2 = argument2_1.64[2];
  new argument2_1_3 = argument2_1.64[3];
  new argument2_r_0 = join_64_x2(argument2_0_0, argument2_1_0);
  new argument2_r_1 = join_64_x2(argument2_0_1, argument2_1_1);
  new argument2_r_2 = join_64_x2(argument2_0_2, argument2_1_2);
  new argument2_r_3 = join_64_x2(argument2_0_3, argument2_1_3);
```


Kalray – all rights reserved

©2025 Kalray

```
new argument3_r_0 = join_64_x2(argument3_0_0, argument3_1_0);
new argument3_r_1 = join_64_x2(argument3_0_1, argument3_1_1);
new argument3_r_2 = join_64_x2(argument3_0_2, argument3_1_2);
new argument3_r_3 = join_64_x2(argument3_0_3, argument3_1_3);
new argument1_0 = XTR[registerAp]:0;
new argument1_0_0_0 = argument1_0.32[0];
new argument1_0_0_1 = argument1_0.32[1];
new argument1_0_1_0 = argument1_0.32[2];
new argument1_0_1_1 = argument1_0.32[3];
new argument1_0_2_0 = argument1_0.32[4];
new argument1_0_2_1 = argument1_0.32[5];
new argument1_0_3_0 = argument1_0.32[6];
new argument1_0_3_1 = argument1_0.32[7];
new argument1_1 = XTR[registerAp]:1;
new argument1_1_0_0 = argument1_1.32[0];
new argument1_1_0_1 = argument1_1.32[1];
new argument1_1_1_0 = argument1_1.32[2];
new argument1_1_1_1 = argument1_1.32[3];
new argument1_1_2_0 = argument1_1.32[4];
new argument1_1_2_1 = argument1_1.32[5];
new argument1_1_3_0 = argument1_1.32[6];
new argument1_1_3_1 = argument1_1.32[7];
new XRM = rounding == 7 ? CS.XRM : rounding;
result1_0.32[0] = fdot8add_16_32_reset(XRM, argument2_r_0, argument3_r_0,
    argument1_0_0_0);
result1_0.32[1] = fdot8add_16_32_noreset(XRM, argument2_r_0, argument3_r_1,
     argument1_0_0_1);
result1_0.32[2] = fdot8add_16_32_noreset(XRM, argument2_r_1, argument3_r_0,
     argument1_0_1_0);
result1_0.32[3] = fdot8add_16_32_noreset(XRM, argument2_r_1, argument3_r_1,
     argument1_0_1_1);
result1_0.32[4] = fdot8add_16_32_noreset(XRM, argument2_r_2, argument3_r_0,
     argument1_0_2_0);
result1_0.32[5] = fdot8add_16_32_noreset(XRM, argument2_r_2, argument3_r_1,
     argument1_0_2_1);
result1_0.32[6] = fdot8add_16_32_noreset(XRM, argument2_r_3, argument3_r_0,
     argument1_0_3_0);
result1_0.32[7] = fdot8add_16_32_noreset(XRM, argument2_r_3, argument3_r_1,
     argument1_0_3_1);
result1_1.32[0] = fdot8add_16_32_noreset(XRM, argument2_r_0, argument3_r_2,
     argument1_1_0_0);
result1_1.32[1] = fdot8add_16_32_noreset(XRM, argument2_r_0, argument3_r_3,
     argument1_1_0_1);
result1_1.32[2] = fdot8add_16_32_noreset(XRM, argument2_r_1, argument3_r_2,
     argument1_1_1_0);
result1_1.32[3] = fdot8add_16_32_noreset(XRM, argument2_r_1, argument3_r_3,
     argument1_1_1_1);
result1_1.32[4] = fdot8add_16_32_noreset(XRM, argument2_r_2, argument3_r_2,
     argument1_1_2_0);
result1_1.32[5] = fdot8add_16_32_noreset(XRM, argument2_r_2, argument3_r_3,
     argument1_1_2_1);
result1_1.32[6] = fdot8add_16_32_noreset(XRM, argument2_r_3, argument3_r_2,
     argument1_1_3_0);
result1_1.32[7] = fdot8add_16_32_noreset(XRM, argument2_r_3, argument3_r_3,
```


Kalray – all rights reserved

```
      argument1_1_3_1);
stage E1:
  CS.XMF = 1;
stage E4:
  if (silent == 0) {
    new CS_xun = CS.XUN;
    new CS_xin = CS.XIN;
    new CS_xio = CS.XIO;
    new CS_xov = CS.XOV;
    CS.XUN = funderflow() | CS_xun;
    CS.XIN = finexact() | CS_xin;
    CS.XIO = finvalid() | CS_xio;
    CS.XOV = foverflow() | CS_xov;
  }
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.12 XFNARROW44WH Floating Point Narrow 4x4 Matrix Word to Half Word

Encoding Format EXT_FNARROWWH (Section 6.12.36), exucode4 = 00

P	00	00	101	registerA	00	silent	rnd	registerCp	0	------
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 7	6	5 0

Syntax `xfnarrow44wh`*roundingsilent registerA = registerCp*

Description The *registerCp* is interpreted as a 4x4 matrix of binary 32 floating-point numbers. This matrix is converted to a 4x4 matrix of binary 16 floating-point numbers according to the current rounding mode and stored into the *registerA*. This instruction may raise extension inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument2_0 = XTR[registerCp]:0;
  new argument2_0_0_0_1 = argument2_0.64[0];
  new argument2_1_0_1_1 = argument2_0.64[1];
  new argument2_2_0_2_1 = argument2_0.64[2];
  new argument2_3_0_3_1 = argument2_0.64[3];
  new argument2_1 = XTR[registerCp]:1;
  new argument2_0_2_0_3 = argument2_1.64[0];
  new argument2_1_2_1_3 = argument2_1.64[1];
  new argument2_2_2_2_3 = argument2_1.64[2];
  new argument2_3_2_3_3 = argument2_1.64[3];
  new XRM = rounding == 7 ? CS.XRM : rounding;
  new argument_0 = join_64_x4(argument2_0_0_0_1, argument2_0_2_0_3,
      argument2_1_0_1_1, argument2_1_2_1_3);
  new argument_1 = join_64_x4(argument2_2_0_2_1, argument2_2_2_2_3,
      argument2_3_0_3_1, argument2_3_2_3_3);
  new result1_0 = fnarrow_32_16_x8_reset(XRM, argument_0);
  new result1_1 = fnarrow_32_16_x8_noreset(XRM, argument_1);
stage E1:
  CS.XMF = 1;
stage E4:
  if (silent == 0) {
    new CS_xun = CS.XUN;
    new CS_xin = CS.XIN;
```


Kalray – all rights reserved

```
    new CS_xio = CS.XIO;
    new CS_xov = CS.XOV;
    CS.XUN = funderflow() | CS_xun;
    CS.XIN = finexact() | CS_xin;
    CS.XIO = finvalid() | CS_xio;
    CS.XOV = foverflow() | CS_xov;
  }
  XVR[registerA] = (result1_1 << 128) | _ZX_128(result1_0);
```


Kalray – all rights reserved

6.9.13 XFSCALEWO Floating Point Scale Word Vector

Encoding Format BCU_XFSCALEWO (Section 6.12.22), exucode4 = 11

P	00	00	001	registerA	11	silent	rnd	registerC	registerZ
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `xfscalewo`*roundingsilent registerA* = *registerC*, *registerZ*

Description The *registerC* is interpreted as a vector of 8 binary 32 floating-point numbers, which are scaled by the power of 2 given in *registerZ* interpreted as an integer. This instruction may raise extension inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

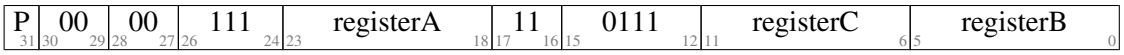
Execution Reservation table BCU_CRRP_CRWL_CRWH (Section 6.14.22)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new pow2scale = _SX_6(GPR[registerZ]);
  new argument2 = XVR[registerC];
  new XRM = rounding == 7 ? CS.XRM : rounding;
  new result1 = fpow2scale_32_32_x8(XRM, pow2scale, argument2);
stage E1:
  CS.XMF = 1;
stage E4:
  if (silent == 0) {
    new CS_xun = CS.XUN;
    new CS_xin = CS.XIN;
    new CS_xio = CS.XIO;
    new CS_xov = CS.XOV;
    CS.XUN = funderflow() | CS_xun;
    CS.XIN = finexact() | CS_xin;
    CS.XIO = finvalid() | CS_xio;
    CS.XOV = foverflow() | CS_xov;
  }
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.14 XIORNO Bitwise Or with Not Operand Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0111



Syntax `xiorno` *registerA* = *registerB*, *registerC*

Description Bitwise inclusive or of not the *registerB* with the *registerC*. The result is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = ~argument2 | argument3;
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.15 XIORO — Bitwise Or Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0010

P	00	00	111	registerA	11	0010	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xioro` *registerA* = *registerB*, *registerC*

Description Bitwise inclusive or of the *registerB* with the *registerC*. The result is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = argument2 | argument3;
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.16 XMADD44BW0 Multiply Add 4x4 Matrix Byte to Word 0

Encoding Format EXT_MADD44BW0 (Section 6.12.31), exucode4 = 00

P	00	00	011	registerAp	0	00	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmadd44bw0` *registerAp* = *registerC*, *registerD*

Description The *registerC* and the *registerD* are interpreted as 4x8 matrices of bytes, and the *registerAp* is interpreted as a 4x4 matrix of words. Each element of the first four columns of the *registerC* is sign-extended and multiplied by the corresponding sign-extended element of the *registerD* and the product is added to the corresponding elements of the *registerAp*. The resulting 4x4 matrix of words is stored back into the *registerAp*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.32[0];
  new argument3_1 = argument3.32[2];
  new argument3_2 = argument3.32[4];
  new argument3_3 = argument3.32[6];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.32[0];
  new argument2_1 = argument2.32[2];
  new argument2_2 = argument2.32[4];
  new argument2_3 = argument2.32[6];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new row_0 = mul_8_32_x4(argument2_0, argument3_0);
  new row_1 = mul_8_32_x4(argument2_1, argument3_1);
  new row_2 = mul_8_32_x4(argument2_2, argument3_2);
  new row_3 = mul_8_32_x4(argument2_3, argument3_3);
  new column_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new column_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new column_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new column_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
  new result1_0 = add_32_32_x8(argument1_0, column_0);
  new result1_1 = add_32_32_x8(argument1_1, column_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.17 XMADD44BW1 Multiply Add 4x4 Matrix Byte to Word 1

Encoding Format EXT_MADD44BW1 (Section 6.12.32), exucode4 = 00

P	00	00	011	registerAp	1	00	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmadd44bw1` *registerAp* = *registerC*, *registerD*

Description The *registerC* and the *registerD* are interpreted as 4x8 matrices of bytes, and the *registerAp* is interpreted as a 4x4 matrix of words. Each element of the last four columns of the *registerC* is sign-extended and multiplied by the corresponding sign-extended element of the *registerD* and the product is added to the corresponding elements of the *registerAp*. The resulting 4x4 matrix of words is stored back into the *registerAp*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.32[1];
  new argument3_1 = argument3.32[3];
  new argument3_2 = argument3.32[5];
  new argument3_3 = argument3.32[7];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.32[1];
  new argument2_1 = argument2.32[3];
  new argument2_2 = argument2.32[5];
  new argument2_3 = argument2.32[7];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new row_0 = mul_8_32_x4(argument2_0, argument3_0);
  new row_1 = mul_8_32_x4(argument2_1, argument3_1);
  new row_2 = mul_8_32_x4(argument2_2, argument3_2);
  new row_3 = mul_8_32_x4(argument2_3, argument3_3);
  new column_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new column_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new column_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new column_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
  new result1_0 = add_32_32_x8(argument1_0, column_0);
  new result1_1 = add_32_32_x8(argument1_1, column_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.18 XMADDIFWO Multiply Add Integer Floating Point Word Octuple

Encoding Format EXT_MADDIFWO (Section 6.12.33), exucode4 = 00

P	00	00	100	registerA	00	silent	rnd	registerC	registerD
31	30 29	28 27	26 24	23 18	17 16	15	14 12	11 6	5 0

Syntax `xmaddifwo`*roundingsilent registerA* = *registerC*, *registerD*

Description The *registerC* is interpreted as a vector of 8 32-bit integers, and the *registerD* as a vector of 8 binary 32 floating-point numbers. The element-wise product of these vectors is computed and added to the *registerA* interpreted as a vector of 8 32-bit integers. Those operations are computed with unlimited precision. The results are saturated to fit into 32-bit signed integers, and stored to the elements of the *registerA*. This instruction may raise extension inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = XVR[registerD];
  new argument2 = XVR[registerC];
  new argument1 = XVR[registerA];
  new XRM = rounding == 7 ? CS.XRM : rounding;
  new result1 = maddif_32_32_x8(XRM, argument1, argument2, argument3);
stage E1:
  CS.XMF = 1;
stage E4:
  if (silent == 0) {
    new CS_xun = CS.XUN;
    new CS_xin = CS.XIN;
    new CS_xio = CS.XIO;
    new CS_xov = CS.XOV;
    CS.XUN = funderflow() | CS_xun;
    CS.XIN = finexact() | CS_xin;
    CS.XIO = finvalid() | CS_xio;
    CS.XOV = foverflow() | CS_xov;
  }
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.19 XMADDSU44BW0 Multiply-Add Signed by Unsigned 4x4 Matrix Byte to Word 0

Encoding Format EXT_MADD44BW0 (Section 6.12.31), exucode4 = 10

P	00	00	011	registerAp	0	10	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmaddsu44bw0` *registerAp* = *registerC*, *registerD*

Description The *registerC* and the *registerD* are interpreted as 4x8 matrices of bytes, and the *registerAp* is interpreted as a 4x4 matrix of words. Each element of the first four columns of the *registerC* is sign-extended and multiplied by the corresponding zero-extended element of the *registerD* and the product is added to the corresponding elements of the *registerAp*. The resulting 4x4 matrix of words is stored back into the *registerAp*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.32[0];
  new argument3_1 = argument3.32[2];
  new argument3_2 = argument3.32[4];
  new argument3_3 = argument3.32[6];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.32[0];
  new argument2_1 = argument2.32[2];
  new argument2_2 = argument2.32[4];
  new argument2_3 = argument2.32[6];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new row_0 = mulsu_8_32_x4(argument2_0, argument3_0);
  new row_1 = mulsu_8_32_x4(argument2_1, argument3_1);
  new row_2 = mulsu_8_32_x4(argument2_2, argument3_2);
  new row_3 = mulsu_8_32_x4(argument2_3, argument3_3);
  new column_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new column_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new column_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new column_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
  new result1_0 = add_32_32_x8(argument1_0, column_0);
  new result1_1 = add_32_32_x8(argument1_1, column_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.20 XMADDSU44BW1 Multiply-Add Signed by Unsigned 4x4 Matrix Byte to Word 1

Encoding Format EXT_MADD44BW1 (Section 6.12.32), exucode4 = 10

P	00	00	011	registerAp	1	10	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmaddsu44bw1` *registerAp* = *registerC*, *registerD*

Description The *registerC* and the *registerD* are interpreted as 4x8 matrices of bytes, and the *registerAp* is interpreted as a 4x4 matrix of words. Each element of the last four columns of the *registerC* is sign-extended and multiplied by the corresponding zero-extended element of the *registerD* and the product is added to the corresponding elements of the *registerAp*. The resulting 4x4 matrix of words is stored back into the *registerAp*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.32[1];
  new argument3_1 = argument3.32[3];
  new argument3_2 = argument3.32[5];
  new argument3_3 = argument3.32[7];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.32[1];
  new argument2_1 = argument2.32[3];
  new argument2_2 = argument2.32[5];
  new argument2_3 = argument2.32[7];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new row_0 = mulsu_8_32_x4(argument2_0, argument3_0);
  new row_1 = mulsu_8_32_x4(argument2_1, argument3_1);
  new row_2 = mulsu_8_32_x4(argument2_2, argument3_2);
  new row_3 = mulsu_8_32_x4(argument2_3, argument3_3);
  new column_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new column_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new column_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new column_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
  new result1_0 = add_32_32_x8(argument1_0, column_0);
  new result1_1 = add_32_32_x8(argument1_1, column_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.21 XMADDU44BW0 Multiply-Add Unsigned 4x4 Matrix Byte to Word 0

Encoding Format EXT_MADD44BW0 (Section 6.12.31), exucode4 = 01

P	00	00	011	registerAp	0	01	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmaddu44bw0` *registerAp* = *registerC*, *registerD*

Description The *registerC* and the *registerD* are interpreted as 4x8 matrices of bytes, and the *registerAp* is interpreted as a 4x4 matrix of words. Each element of the first four columns of the *registerC* is zero-extended and multiplied by the corresponding zero-extended element of the *registerD* and the product is added to the corresponding elements of the *registerAp*. The resulting 4x4 matrix of words is stored back into the *registerAp*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.32[0];
  new argument3_1 = argument3.32[2];
  new argument3_2 = argument3.32[4];
  new argument3_3 = argument3.32[6];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.32[0];
  new argument2_1 = argument2.32[2];
  new argument2_2 = argument2.32[4];
  new argument2_3 = argument2.32[6];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new row_0 = mulu_8_32_x4(argument2_0, argument3_0);
  new row_1 = mulu_8_32_x4(argument2_1, argument3_1);
  new row_2 = mulu_8_32_x4(argument2_2, argument3_2);
  new row_3 = mulu_8_32_x4(argument2_3, argument3_3);
  new column_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new column_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new column_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new column_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
  new result1_0 = add_32_32_x8(argument1_0, column_0);
  new result1_1 = add_32_32_x8(argument1_1, column_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.22 XMADDU44BW1 Multiply-Add Unsigned 4x4 Matrix Byte to Word 1

Encoding Format EXT_MADD44BW1 (Section 6.12.32), exucode4 = 01

P	00	00	011	registerAp	1	01	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmaddu44bw1` *registerAp* = *registerC*, *registerD*

Description The *registerC* and the *registerD* are interpreted as 4x8 matrices of bytes, and the *registerAp* is interpreted as a 4x4 matrix of words. Each element of the last four columns of the *registerC* is zero-extended and multiplied by the corresponding zero-extended element of the *registerD* and the product is added to the corresponding elements of the *registerAp*. The resulting 4x4 matrix of words is stored back into the *registerAp*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.32[1];
  new argument3_1 = argument3.32[3];
  new argument3_2 = argument3.32[5];
  new argument3_3 = argument3.32[7];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.32[1];
  new argument2_1 = argument2.32[3];
  new argument2_2 = argument2.32[5];
  new argument2_3 = argument2.32[7];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new row_0 = mulu_8_32_x4(argument2_0, argument3_0);
  new row_1 = mulu_8_32_x4(argument2_1, argument3_1);
  new row_2 = mulu_8_32_x4(argument2_2, argument3_2);
  new row_3 = mulu_8_32_x4(argument2_3, argument3_3);
  new column_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new column_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new column_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new column_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
  new result1_0 = add_32_32_x8(argument1_0, column_0);
  new result1_1 = add_32_32_x8(argument1_1, column_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.23 XMMA4164BW Matrix Multiply-Add 4x16x4 Bytes to Words

Encoding Format EXT_MMABW0 (Section 6.12.23), exucode4 = 00

P	00	00	010	registerAp	0	00	– – – –	registerCp	0	registerDp	0
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 7	6	5 1	0

Syntax `xmma4164bw` *registerAp* = *registerCp*, *registerDp*

Description The *registerAp* interpreted as a 4x4 matrix of words accumulates the product of the *registerCp* interpreted as a 4x16 matrix of signed bytes by the transposed *registerDp* interpreted as 4x16 matrix of signed bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3_0 = XTR[registerDp]:0;
  new argument3_0_0 = argument3_0.64[0];
  new argument3_0_1 = argument3_0.64[1];
  new argument3_0_2 = argument3_0.64[2];
  new argument3_0_3 = argument3_0.64[3];
  new argument3_1 = XTR[registerDp]:1;
  new argument3_1_0 = argument3_1.64[0];
  new argument3_1_1 = argument3_1.64[1];
  new argument3_1_2 = argument3_1.64[2];
  new argument3_1_3 = argument3_1.64[3];
  new argument2_0 = XTR[registerCp]:0;
  new argument2_0_0 = argument2_0.64[0];
  new argument2_0_1 = argument2_0.64[1];
  new argument2_0_2 = argument2_0.64[2];
  new argument2_0_3 = argument2_0.64[3];
  new argument2_1 = XTR[registerCp]:1;
  new argument2_1_0 = argument2_1.64[0];
  new argument2_1_1 = argument2_1.64[1];
  new argument2_1_2 = argument2_1.64[2];
  new argument2_1_3 = argument2_1.64[3];
  new argument2_r_0 = join_64_x2(argument2_0_0, argument2_1_0);
  new argument2_r_1 = join_64_x2(argument2_0_1, argument2_1_1);
  new argument2_r_2 = join_64_x2(argument2_0_2, argument2_1_2);
  new argument2_r_3 = join_64_x2(argument2_0_3, argument2_1_3);
  new argument3_r_0 = join_64_x2(argument3_0_0, argument3_1_0);
  new argument3_r_1 = join_64_x2(argument3_0_1, argument3_1_1);
  new argument3_r_2 = join_64_x2(argument3_0_2, argument3_1_2);
  new argument3_r_3 = join_64_x2(argument3_0_3, argument3_1_3);
  new argument1_0 = XTR[registerAp]:0;
  new argument1_0_0 = argument1_0.32[0];
  new argument1_0_1 = argument1_0.32[1];
  new argument1_1_0 = argument1_0.32[2];
  new argument1_1_1 = argument1_0.32[3];
  new argument1_2_0 = argument1_0.32[4];
```


Kalray – all rights reserved

```
  new argument1_2_1 = argument1_0.32[5];
  new argument1_3_0 = argument1_0.32[6];
  new argument1_3_1 = argument1_0.32[7];
  new argument1_1 = XTR[registerAp]:1;
  new argument1_0_2 = argument1_1.32[0];
  new argument1_0_3 = argument1_1.32[1];
  new argument1_1_2 = argument1_1.32[2];
  new argument1_1_3 = argument1_1.32[3];
  new argument1_2_2 = argument1_1.32[4];
  new argument1_2_3 = argument1_1.32[5];
  new argument1_3_2 = argument1_1.32[6];
  new argument1_3_3 = argument1_1.32[7];
  result1_0.32[0] = dot16add_8_32(argument2_r_0, argument3_r_0, argument1_0_0
      );
  result1_0.32[1] = dot16add_8_32(argument2_r_0, argument3_r_1, argument1_0_1
      );
  result1_0.32[2] = dot16add_8_32(argument2_r_1, argument3_r_0, argument1_1_0
      );
  result1_0.32[3] = dot16add_8_32(argument2_r_1, argument3_r_1, argument1_1_1
      );
  result1_0.32[4] = dot16add_8_32(argument2_r_2, argument3_r_0, argument1_2_0
      );
  result1_0.32[5] = dot16add_8_32(argument2_r_2, argument3_r_1, argument1_2_1
      );
  result1_0.32[6] = dot16add_8_32(argument2_r_3, argument3_r_0, argument1_3_0
      );
  result1_0.32[7] = dot16add_8_32(argument2_r_3, argument3_r_1, argument1_3_1
      );
  result1_1.32[0] = dot16add_8_32(argument2_r_0, argument3_r_2, argument1_0_2
      );
  result1_1.32[1] = dot16add_8_32(argument2_r_0, argument3_r_3, argument1_0_3
      );
  result1_1.32[2] = dot16add_8_32(argument2_r_1, argument3_r_2, argument1_1_2
      );
  result1_1.32[3] = dot16add_8_32(argument2_r_1, argument3_r_3, argument1_1_3
      );
  result1_1.32[4] = dot16add_8_32(argument2_r_2, argument3_r_2, argument1_2_2
      );
  result1_1.32[5] = dot16add_8_32(argument2_r_2, argument3_r_3, argument1_2_3
      );
  result1_1.32[6] = dot16add_8_32(argument2_r_3, argument3_r_2, argument1_3_2
      );
  result1_1.32[7] = dot16add_8_32(argument2_r_3, argument3_r_3, argument1_3_3
      );
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.24 XMMA484BW Extension Matrix Multiply-Add 4x8x4 Bytes to Words

Encoding Format EXT_MMABW (Section 6.12.27), exucode4 = 00

P	00	00	010	registerAp	1	00	– – – –	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmma484bw` *registerAp* = *registerC*, *registerD*

Description The *registerAp* interpreted as a 4x4 matrix of words receives the sum of the *registerAp* interpreted as a 4x4 matrix of words to the product of the *registerC* interpreted as a 4x8 matrix of signed bytes by the transposed *registerD* interpreted as a 4x8 matrix of signed bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.64[0];
  new argument3_1 = argument3.64[1];
  new argument3_2 = argument3.64[2];
  new argument3_3 = argument3.64[3];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.64[0];
  new argument2_1 = argument2.64[1];
  new argument2_2 = argument2.64[2];
  new argument2_3 = argument2.64[3];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new result1_0_0 = dot8_8_32(argument2_0, argument3_0);
  new result1_0_1 = dot8_8_32(argument2_0, argument3_1);
  new result1_0_2 = dot8_8_32(argument2_0, argument3_2);
  new result1_0_3 = dot8_8_32(argument2_0, argument3_3);
  new result1_1_0 = dot8_8_32(argument2_1, argument3_0);
  new result1_1_1 = dot8_8_32(argument2_1, argument3_1);
  new result1_1_2 = dot8_8_32(argument2_1, argument3_2);
  new result1_1_3 = dot8_8_32(argument2_1, argument3_3);
  new result1_2_0 = dot8_8_32(argument2_2, argument3_0);
  new result1_2_1 = dot8_8_32(argument2_2, argument3_1);
  new result1_2_2 = dot8_8_32(argument2_2, argument3_2);
  new result1_2_3 = dot8_8_32(argument2_2, argument3_3);
  new result1_3_0 = dot8_8_32(argument2_3, argument3_0);
  new result1_3_1 = dot8_8_32(argument2_3, argument3_1);
  new result1_3_2 = dot8_8_32(argument2_3, argument3_2);
  new result1_3_3 = dot8_8_32(argument2_3, argument3_3);
  new result1_0 = join_32_x8(result1_0_0, result1_0_1, result1_1_0,
      result1_1_1, result1_2_0, result1_2_1, result1_3_0, result1_3_1);
  new result1_1 = join_32_x8(result1_0_2, result1_0_3, result1_1_2,
      result1_1_3, result1_2_2, result1_2_3, result1_3_2, result1_3_3);
  result1_0 = add_32_32_x8(argument1_0, result1_0);
  result1_1 = add_32_32_x8(argument1_1, result1_1);
```


Kalray – all rights reserved

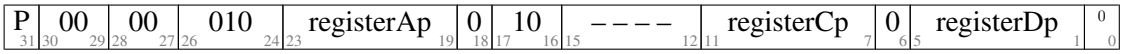
```
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.25 XMMASU4164BW Matrix Multiply-Add 4x16x4 Signed by Unsigned Bytes to Words

Encoding Format EXT_MMABW2 (Section 6.12.25), exucode4 = 10



Syntax `xmmasu4164bw` *registerAp* = *registerCp*, *registerDp*

Description The *registerAp* interpreted as a 4x4 matrix of words accumulates the product of the *registerCp* interpreted as a 4x16 matrix of signed bytes by the transposed *registerDp* interpreted as a 4x16 matrix of unsigned bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3_0 = XTR[registerDp]:0;
  new argument3_0_0 = argument3_0.64[0];
  new argument3_0_1 = argument3_0.64[1];
  new argument3_0_2 = argument3_0.64[2];
  new argument3_0_3 = argument3_0.64[3];
  new argument3_1 = XTR[registerDp]:1;
  new argument3_1_0 = argument3_1.64[0];
  new argument3_1_1 = argument3_1.64[1];
  new argument3_1_2 = argument3_1.64[2];
  new argument3_1_3 = argument3_1.64[3];
  new argument2_0 = XTR[registerCp]:0;
  new argument2_0_0 = argument2_0.64[0];
  new argument2_0_1 = argument2_0.64[1];
  new argument2_0_2 = argument2_0.64[2];
  new argument2_0_3 = argument2_0.64[3];
  new argument2_1 = XTR[registerCp]:1;
  new argument2_1_0 = argument2_1.64[0];
  new argument2_1_1 = argument2_1.64[1];
  new argument2_1_2 = argument2_1.64[2];
  new argument2_1_3 = argument2_1.64[3];
  new argument2_r_0 = join_64_x2(argument2_0_0, argument2_1_0);
  new argument2_r_1 = join_64_x2(argument2_0_1, argument2_1_1);
  new argument2_r_2 = join_64_x2(argument2_0_2, argument2_1_2);
  new argument2_r_3 = join_64_x2(argument2_0_3, argument2_1_3);
  new argument3_r_0 = join_64_x2(argument3_0_0, argument3_1_0);
  new argument3_r_1 = join_64_x2(argument3_0_1, argument3_1_1);
  new argument3_r_2 = join_64_x2(argument3_0_2, argument3_1_2);
  new argument3_r_3 = join_64_x2(argument3_0_3, argument3_1_3);
  new argument1_0 = XTR[registerAp]:0;
  new argument1_0_0 = argument1_0.32[0];
  new argument1_0_1 = argument1_0.32[1];
  new argument1_1_0 = argument1_0.32[2];
  new argument1_1_1 = argument1_0.32[3];
```


Kalray – all rights reserved

```
  new argument1_2_0 = argument1_0.32[4];
  new argument1_2_1 = argument1_0.32[5];
  new argument1_3_0 = argument1_0.32[6];
  new argument1_3_1 = argument1_0.32[7];
  new argument1_1 = XTR[registerAp]:1;
  new argument1_0_2 = argument1_1.32[0];
  new argument1_0_3 = argument1_1.32[1];
  new argument1_1_2 = argument1_1.32[2];
  new argument1_1_3 = argument1_1.32[3];
  new argument1_2_2 = argument1_1.32[4];
  new argument1_2_3 = argument1_1.32[5];
  new argument1_3_2 = argument1_1.32[6];
  new argument1_3_3 = argument1_1.32[7];
  result1_0.32[0] = dot16addsu_8_32(argument2_r_0, argument3_r_0,
      argument1_0_0);
  result1_0.32[1] = dot16addsu_8_32(argument2_r_0, argument3_r_1,
      argument1_0_1);
  result1_0.32[2] = dot16addsu_8_32(argument2_r_1, argument3_r_0,
      argument1_1_0);
  result1_0.32[3] = dot16addsu_8_32(argument2_r_1, argument3_r_1,
      argument1_1_1);
  result1_0.32[4] = dot16addsu_8_32(argument2_r_2, argument3_r_0,
      argument1_2_0);
  result1_0.32[5] = dot16addsu_8_32(argument2_r_2, argument3_r_1,
      argument1_2_1);
  result1_0.32[6] = dot16addsu_8_32(argument2_r_3, argument3_r_0,
      argument1_3_0);
  result1_0.32[7] = dot16addsu_8_32(argument2_r_3, argument3_r_1,
      argument1_3_1);
  result1_1.32[0] = dot16addsu_8_32(argument2_r_0, argument3_r_2,
      argument1_0_2);
  result1_1.32[1] = dot16addsu_8_32(argument2_r_0, argument3_r_3,
      argument1_0_3);
  result1_1.32[2] = dot16addsu_8_32(argument2_r_1, argument3_r_2,
      argument1_1_2);
  result1_1.32[3] = dot16addsu_8_32(argument2_r_1, argument3_r_3,
      argument1_1_3);
  result1_1.32[4] = dot16addsu_8_32(argument2_r_2, argument3_r_2,
      argument1_2_2);
  result1_1.32[5] = dot16addsu_8_32(argument2_r_2, argument3_r_3,
      argument1_2_3);
  result1_1.32[6] = dot16addsu_8_32(argument2_r_3, argument3_r_2,
      argument1_3_2);
  result1_1.32[7] = dot16addsu_8_32(argument2_r_3, argument3_r_3,
      argument1_3_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.26 XMMASU484BW Matrix Multiply-Add 4x8x4 Signed by Unsigned Bytes to Words

Encoding Format EXT_MMASUBW (Section 6.12.29), exucode4 = 10

P	00	00	010	registerAp	1	10	----	registerC	registerD
31	30 29	28 27	26 24	23 19	18	17 16	15 12	11 6	5 0

Syntax `xmmasu484bw` *registerAp* = *registerC*, *registerD*

Description The *registerAp* interpreted as a 4x4 matrix of words receives the sum of the *registerAp* interpreted as a 4x4 matrix of words to the product of the *registerC* interpreted as a 4x8 matrix of signed bytes by the transposed *registerD* interpreted as a 4x8 matrix of unsigned bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.64[0];
  new argument3_1 = argument3.64[1];
  new argument3_2 = argument3.64[2];
  new argument3_3 = argument3.64[3];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.64[0];
  new argument2_1 = argument2.64[1];
  new argument2_2 = argument2.64[2];
  new argument2_3 = argument2.64[3];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new result1_0_0 = dot8su_8_32(argument2_0, argument3_0);
  new result1_0_1 = dot8su_8_32(argument2_0, argument3_1);
  new result1_0_2 = dot8su_8_32(argument2_0, argument3_2);
  new result1_0_3 = dot8su_8_32(argument2_0, argument3_3);
  new result1_1_0 = dot8su_8_32(argument2_1, argument3_0);
  new result1_1_1 = dot8su_8_32(argument2_1, argument3_1);
  new result1_1_2 = dot8su_8_32(argument2_1, argument3_2);
  new result1_1_3 = dot8su_8_32(argument2_1, argument3_3);
  new result1_2_0 = dot8su_8_32(argument2_2, argument3_0);
  new result1_2_1 = dot8su_8_32(argument2_2, argument3_1);
  new result1_2_2 = dot8su_8_32(argument2_2, argument3_2);
  new result1_2_3 = dot8su_8_32(argument2_2, argument3_3);
  new result1_3_0 = dot8su_8_32(argument2_3, argument3_0);
  new result1_3_1 = dot8su_8_32(argument2_3, argument3_1);
  new result1_3_2 = dot8su_8_32(argument2_3, argument3_2);
  new result1_3_3 = dot8su_8_32(argument2_3, argument3_3);
  new result1_0 = join_32_x8(result1_0_0, result1_0_1, result1_1_0,
      result1_1_1, result1_2_0, result1_2_1, result1_3_0, result1_3_1);
  new result1_1 = join_32_x8(result1_0_2, result1_0_3, result1_1_2,
      result1_1_3, result1_2_2, result1_2_3, result1_3_2, result1_3_3);
  result1_0 = add_32_32_x8(argument1_0, result1_0);
```

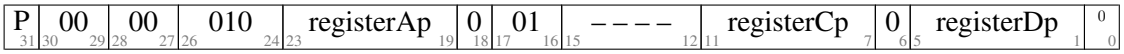

Kalray – all rights reserved

```
  result1_1 = add_32_32_x8(argument1_1, result1_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.27 XMMAU4164BW Matrix Multiply-Add 4x16x4 Unsigned by Unsigned Bytes to Words

Encoding Format EXT_MMABW1 (Section 6.12.24), exucode4 = 01



Syntax `xmmau4164bw` *registerAp* = *registerCp*, *registerDp*

Description The *registerAp* interpreted as a 4x4 matrix of words accumulates the product of the *registerCp* interpreted as a 4x16 matrix of unsigned bytes by the transposed *registerDp* interpreted as a 4x16 matrix of unsigned bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3_0 = XTR[registerDp]:0;
  new argument3_0_0 = argument3_0.64[0];
  new argument3_0_1 = argument3_0.64[1];
  new argument3_0_2 = argument3_0.64[2];
  new argument3_0_3 = argument3_0.64[3];
  new argument3_1 = XTR[registerDp]:1;
  new argument3_1_0 = argument3_1.64[0];
  new argument3_1_1 = argument3_1.64[1];
  new argument3_1_2 = argument3_1.64[2];
  new argument3_1_3 = argument3_1.64[3];
  new argument2_0 = XTR[registerCp]:0;
  new argument2_0_0 = argument2_0.64[0];
  new argument2_0_1 = argument2_0.64[1];
  new argument2_0_2 = argument2_0.64[2];
  new argument2_0_3 = argument2_0.64[3];
  new argument2_1 = XTR[registerCp]:1;
  new argument2_1_0 = argument2_1.64[0];
  new argument2_1_1 = argument2_1.64[1];
  new argument2_1_2 = argument2_1.64[2];
  new argument2_1_3 = argument2_1.64[3];
  new argument2_r_0 = join_64_x2(argument2_0_0, argument2_1_0);
  new argument2_r_1 = join_64_x2(argument2_0_1, argument2_1_1);
  new argument2_r_2 = join_64_x2(argument2_0_2, argument2_1_2);
  new argument2_r_3 = join_64_x2(argument2_0_3, argument2_1_3);
  new argument3_r_0 = join_64_x2(argument3_0_0, argument3_1_0);
  new argument3_r_1 = join_64_x2(argument3_0_1, argument3_1_1);
  new argument3_r_2 = join_64_x2(argument3_0_2, argument3_1_2);
  new argument3_r_3 = join_64_x2(argument3_0_3, argument3_1_3);
  new argument1_0 = XTR[registerAp]:0;
  new argument1_0_0 = argument1_0.32[0];
  new argument1_0_1 = argument1_0.32[1];
  new argument1_1_0 = argument1_0.32[2];
  new argument1_1_1 = argument1_0.32[3];
```

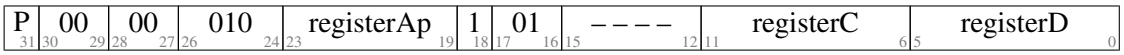

Kalray – all rights reserved

```
  new argument1_2_0 = argument1_0.32[4];
  new argument1_2_1 = argument1_0.32[5];
  new argument1_3_0 = argument1_0.32[6];
  new argument1_3_1 = argument1_0.32[7];
  new argument1_1 = XTR[registerAp]:1;
  new argument1_0_2 = argument1_1.32[0];
  new argument1_0_3 = argument1_1.32[1];
  new argument1_1_2 = argument1_1.32[2];
  new argument1_1_3 = argument1_1.32[3];
  new argument1_2_2 = argument1_1.32[4];
  new argument1_2_3 = argument1_1.32[5];
  new argument1_3_2 = argument1_1.32[6];
  new argument1_3_3 = argument1_1.32[7];
  result1_0.32[0] = dot16addu_8_32(argument2_r_0, argument3_r_0,
      argument1_0_0);
  result1_0.32[1] = dot16addu_8_32(argument2_r_0, argument3_r_1,
      argument1_0_1);
  result1_0.32[2] = dot16addu_8_32(argument2_r_1, argument3_r_0,
      argument1_1_0);
  result1_0.32[3] = dot16addu_8_32(argument2_r_1, argument3_r_1,
      argument1_1_1);
  result1_0.32[4] = dot16addu_8_32(argument2_r_2, argument3_r_0,
      argument1_2_0);
  result1_0.32[5] = dot16addu_8_32(argument2_r_2, argument3_r_1,
      argument1_2_1);
  result1_0.32[6] = dot16addu_8_32(argument2_r_3, argument3_r_0,
      argument1_3_0);
  result1_0.32[7] = dot16addu_8_32(argument2_r_3, argument3_r_1,
      argument1_3_1);
  result1_1.32[0] = dot16addu_8_32(argument2_r_0, argument3_r_2,
      argument1_0_2);
  result1_1.32[1] = dot16addu_8_32(argument2_r_0, argument3_r_3,
      argument1_0_3);
  result1_1.32[2] = dot16addu_8_32(argument2_r_1, argument3_r_2,
      argument1_1_2);
  result1_1.32[3] = dot16addu_8_32(argument2_r_1, argument3_r_3,
      argument1_1_3);
  result1_1.32[4] = dot16addu_8_32(argument2_r_2, argument3_r_2,
      argument1_2_2);
  result1_1.32[5] = dot16addu_8_32(argument2_r_2, argument3_r_3,
      argument1_2_3);
  result1_1.32[6] = dot16addu_8_32(argument2_r_3, argument3_r_2,
      argument1_3_2);
  result1_1.32[7] = dot16addu_8_32(argument2_r_3, argument3_r_3,
      argument1_3_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.28 XMMAU484BW Matrix Multiply-Add 4x8x4 Unsigned by Unsigned Bytes to Words

Encoding Format EXT_MMAUBW (Section 6.12.28), exucode4 = 01



Syntax `xmmau484bw` *registerAp* = *registerC*, *registerD*

Description The *registerAp* interpreted as a 4x4 matrix of words receives the sum of the *registerAp* interpreted as a 4x4 matrix of words to the product of the *registerC* interpreted as a 4x8 matrix of unsigned bytes by the transposed *registerD* interpreted as a 4x8 matrix of unsigned bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.64[0];
  new argument3_1 = argument3.64[1];
  new argument3_2 = argument3.64[2];
  new argument3_3 = argument3.64[3];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.64[0];
  new argument2_1 = argument2.64[1];
  new argument2_2 = argument2.64[2];
  new argument2_3 = argument2.64[3];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new result1_0_0 = dot8u_8_32(argument2_0, argument3_0);
  new result1_0_1 = dot8u_8_32(argument2_0, argument3_1);
  new result1_0_2 = dot8u_8_32(argument2_0, argument3_2);
  new result1_0_3 = dot8u_8_32(argument2_0, argument3_3);
  new result1_1_0 = dot8u_8_32(argument2_1, argument3_0);
  new result1_1_1 = dot8u_8_32(argument2_1, argument3_1);
  new result1_1_2 = dot8u_8_32(argument2_1, argument3_2);
  new result1_1_3 = dot8u_8_32(argument2_1, argument3_3);
  new result1_2_0 = dot8u_8_32(argument2_2, argument3_0);
  new result1_2_1 = dot8u_8_32(argument2_2, argument3_1);
  new result1_2_2 = dot8u_8_32(argument2_2, argument3_2);
  new result1_2_3 = dot8u_8_32(argument2_2, argument3_3);
  new result1_3_0 = dot8u_8_32(argument2_3, argument3_0);
  new result1_3_1 = dot8u_8_32(argument2_3, argument3_1);
  new result1_3_2 = dot8u_8_32(argument2_3, argument3_2);
  new result1_3_3 = dot8u_8_32(argument2_3, argument3_3);
  new result1_0 = join_32_x8(result1_0_0, result1_0_1, result1_1_0,
      result1_1_1, result1_2_0, result1_2_1, result1_3_0, result1_3_1);
  new result1_1 = join_32_x8(result1_0_2, result1_0_3, result1_1_2,
      result1_1_3, result1_2_2, result1_2_3, result1_3_2, result1_3_3);
```


Kalray – all rights reserved

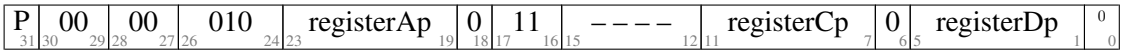
```
  result1_0 = add_32_32_x8(argument1_0, result1_0);
  result1_1 = add_32_32_x8(argument1_1, result1_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.29 XMMAUS4164BW Matrix Multiply-Add 4x16x4 Unsigned by Signed Bytes to Words

Encoding Format EXT_MMABW3 (Section 6.12.26), exucode4 = 11



Syntax `xmmaus4164bw` *registerAp* = *registerCp*, *registerDp*

Description The *registerAp* interpreted as a 4x4 matrix of words accumulates the product of the *registerCp* interpreted as a 4x16 matrix of unsigned bytes by the transposed *registerDp* interpreted as a 4x16 matrix of signed bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3_0 = XTR[registerDp]:0;
  new argument3_0_0 = argument3_0.64[0];
  new argument3_0_1 = argument3_0.64[1];
  new argument3_0_2 = argument3_0.64[2];
  new argument3_0_3 = argument3_0.64[3];
  new argument3_1 = XTR[registerDp]:1;
  new argument3_1_0 = argument3_1.64[0];
  new argument3_1_1 = argument3_1.64[1];
  new argument3_1_2 = argument3_1.64[2];
  new argument3_1_3 = argument3_1.64[3];
  new argument2_0 = XTR[registerCp]:0;
  new argument2_0_0 = argument2_0.64[0];
  new argument2_0_1 = argument2_0.64[1];
  new argument2_0_2 = argument2_0.64[2];
  new argument2_0_3 = argument2_0.64[3];
  new argument2_1 = XTR[registerCp]:1;
  new argument2_1_0 = argument2_1.64[0];
  new argument2_1_1 = argument2_1.64[1];
  new argument2_1_2 = argument2_1.64[2];
  new argument2_1_3 = argument2_1.64[3];
  new argument2_r_0 = join_64_x2(argument2_0_0, argument2_1_0);
  new argument2_r_1 = join_64_x2(argument2_0_1, argument2_1_1);
  new argument2_r_2 = join_64_x2(argument2_0_2, argument2_1_2);
  new argument2_r_3 = join_64_x2(argument2_0_3, argument2_1_3);
  new argument3_r_0 = join_64_x2(argument3_0_0, argument3_1_0);
  new argument3_r_1 = join_64_x2(argument3_0_1, argument3_1_1);
  new argument3_r_2 = join_64_x2(argument3_0_2, argument3_1_2);
  new argument3_r_3 = join_64_x2(argument3_0_3, argument3_1_3);
  new argument1_0 = XTR[registerAp]:0;
  new argument1_0_0 = argument1_0.32[0];
  new argument1_0_1 = argument1_0.32[1];
  new argument1_1_0 = argument1_0.32[2];
  new argument1_1_1 = argument1_0.32[3];
```


Kalray – all rights reserved

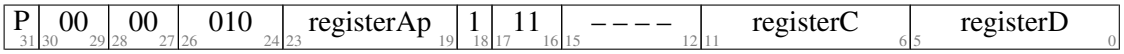
```
  new argument1_2_0 = argument1_0.32[4];
  new argument1_2_1 = argument1_0.32[5];
  new argument1_3_0 = argument1_0.32[6];
  new argument1_3_1 = argument1_0.32[7];
  new argument1_1 = XTR[registerAp]:1;
  new argument1_0_2 = argument1_1.32[0];
  new argument1_0_3 = argument1_1.32[1];
  new argument1_1_2 = argument1_1.32[2];
  new argument1_1_3 = argument1_1.32[3];
  new argument1_2_2 = argument1_1.32[4];
  new argument1_2_3 = argument1_1.32[5];
  new argument1_3_2 = argument1_1.32[6];
  new argument1_3_3 = argument1_1.32[7];
  result1_0.32[0] = dot16addus_8_32(argument2_r_0, argument3_r_0,
      argument1_0_0);
  result1_0.32[1] = dot16addus_8_32(argument2_r_0, argument3_r_1,
      argument1_0_1);
  result1_0.32[2] = dot16addus_8_32(argument2_r_1, argument3_r_0,
      argument1_1_0);
  result1_0.32[3] = dot16addus_8_32(argument2_r_1, argument3_r_1,
      argument1_1_1);
  result1_0.32[4] = dot16addus_8_32(argument2_r_2, argument3_r_0,
      argument1_2_0);
  result1_0.32[5] = dot16addus_8_32(argument2_r_2, argument3_r_1,
      argument1_2_1);
  result1_0.32[6] = dot16addus_8_32(argument2_r_3, argument3_r_0,
      argument1_3_0);
  result1_0.32[7] = dot16addus_8_32(argument2_r_3, argument3_r_1,
      argument1_3_1);
  result1_1.32[0] = dot16addus_8_32(argument2_r_0, argument3_r_2,
      argument1_0_2);
  result1_1.32[1] = dot16addus_8_32(argument2_r_0, argument3_r_3,
      argument1_0_3);
  result1_1.32[2] = dot16addus_8_32(argument2_r_1, argument3_r_2,
      argument1_1_2);
  result1_1.32[3] = dot16addus_8_32(argument2_r_1, argument3_r_3,
      argument1_1_3);
  result1_1.32[4] = dot16addus_8_32(argument2_r_2, argument3_r_2,
      argument1_2_2);
  result1_1.32[5] = dot16addus_8_32(argument2_r_2, argument3_r_3,
      argument1_2_3);
  result1_1.32[6] = dot16addus_8_32(argument2_r_3, argument3_r_2,
      argument1_3_2);
  result1_1.32[7] = dot16addus_8_32(argument2_r_3, argument3_r_3,
      argument1_3_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.30 XMMAUS484BW Matrix Multiply-Add 4x8x4 Unsigned by Signed Bytes to Words

Encoding Format EXT_MMAUSBW (Section 6.12.30), exucode4 = 11



Syntax `xmmaus484bw` *registerAp* = *registerC*, *registerD*

Description The *registerAp* interpreted as a 4x4 matrix of words receives the sum of the *registerAp* interpreted as a 4x4 matrix of words to the product of the *registerC* interpreted as a 4x8 matrix of unsigned bytes by the transposed *registerD* interpreted as a 4x8 matrix of signed bytes.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerD];
  new argument3_0 = argument3.64[0];
  new argument3_1 = argument3.64[1];
  new argument3_2 = argument3.64[2];
  new argument3_3 = argument3.64[3];
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.64[0];
  new argument2_1 = argument2.64[1];
  new argument2_2 = argument2.64[2];
  new argument2_3 = argument2.64[3];
  new argument1_0 = XTR[registerAp]:0;
  new argument1_1 = XTR[registerAp]:1;
  new result1_0_0 = dot8us_8_32(argument2_0, argument3_0);
  new result1_0_1 = dot8us_8_32(argument2_0, argument3_1);
  new result1_0_2 = dot8us_8_32(argument2_0, argument3_2);
  new result1_0_3 = dot8us_8_32(argument2_0, argument3_3);
  new result1_1_0 = dot8us_8_32(argument2_1, argument3_0);
  new result1_1_1 = dot8us_8_32(argument2_1, argument3_1);
  new result1_1_2 = dot8us_8_32(argument2_1, argument3_2);
  new result1_1_3 = dot8us_8_32(argument2_1, argument3_3);
  new result1_2_0 = dot8us_8_32(argument2_2, argument3_0);
  new result1_2_1 = dot8us_8_32(argument2_2, argument3_1);
  new result1_2_2 = dot8us_8_32(argument2_2, argument3_2);
  new result1_2_3 = dot8us_8_32(argument2_2, argument3_3);
  new result1_3_0 = dot8us_8_32(argument2_3, argument3_0);
  new result1_3_1 = dot8us_8_32(argument2_3, argument3_1);
  new result1_3_2 = dot8us_8_32(argument2_3, argument3_2);
  new result1_3_3 = dot8us_8_32(argument2_3, argument3_3);
  new result1_0 = join_32_x8(result1_0_0, result1_0_1, result1_1_0,
      result1_1_1, result1_2_0, result1_2_1, result1_3_0, result1_3_1);
  new result1_1 = join_32_x8(result1_0_2, result1_0_3, result1_1_2,
      result1_1_3, result1_2_2, result1_2_3, result1_3_2, result1_3_3);
  result1_0 = add_32_32_x8(argument1_0, result1_0);
```


Kalray – all rights reserved

```
  result1_1 = add_32_32_x8(argument1_1, result1_1);
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

6.9.31 XMOVEFD Move From Extension Double Word

Encoding Format BCU_XMOVEFD (Section 6.12.5), tcacode1 = 000

P	00	00	000	registerW	00	1	1	qselectC	registerCc	– – – – – –
31	30 29	28 27	26 24	23 18	17 16	15	14	13 12	11 6	5 0

Syntax `xmovefd` *registerW* = *registerCc_qselectC*

Description The *registerW* operand receives the contents of the *registerCc_qselectC*.

Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new argument2 = XCR[registerCc_qselectC];
  new result1 = _ZX_64(argument2);
stage E1:
  CS.XMF = 1;
stage E3:
  GPR[registerW] = result1;
```


Kalray – all rights reserved

6.9.32 XMOVEFO — Move From Extension Octuple Word

Encoding Format BCU_XMOVEFO (Section 6.12.7), tcacode1 = 000

P	00	00	000	registerN	01	00	1	000	registerC	------
31	30 29	28 27	26 24	23 20	19 18	17 16	15	14 12	11 6	5 0

Syntax `xmovefo` *registerN* = *registerC*

Description The *registerN* operand receives the contents of the *registerC*.

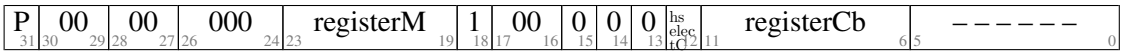
Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new argument2 = XVR[registerC];
  new result1 = _ZX_256(argument2);
stage E1:
  CS.XMF = 1;
stage E3:
  QGR[registerN] = result1;
```


Kalray – all rights reserved

6.9.33 XMOVEFQ Move From Extension Quadruple Word

Encoding Format BCU_XMOVEFQ (Section 6.12.6), tcacode1 = 000



Syntax `xmovefq` *registerM* = *registerCb_hselectC*

Description The *registerM* operand receives the contents of the *registerCb_hselectC*.

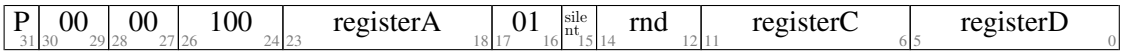
Execution Reservation table BCU_TINY_AUXW_CRRP (Section 6.14.23)

```
stage RR:
  new argument2 = XBR[registerCb_hselectC];
  new result1 = _ZX_128(argument2);
stage E1:
  CS.XMF = 1;
stage E3:
  PGR[registerM] = result1;
```


Kalray – all rights reserved

6.9.34 XMSBFIFWO Multiply Subtract Integer Floating Point Word Octuple

Encoding Format EXT_MADDIFWO (Section 6.12.33), exucode4 = 01



Syntax `xmsbfifwo`*roundingsilent registerA = registerC, registerD*

Description The *registerC* is interpreted as a vector of 8 32-bit integers, and the *registerD* as a vector of 8 binary 32 floating-point numbers. The element-wise product of these vectors is computed and subtracted from the *registerA* interpreted as a vector of 8 32-bit integers. Those operations are computed with unlimited precision. The results are saturated to fit into 32-bit signed integers, and stored to the elements of the *registerA*. This instruction may raise extension inexact, invalid, overflow or underflow exception bits in the CS register.

Modifier(s)

- rounding (cf. rounding in section 6.10.2 on page 1424)
- silent (cf. silent in section 6.10.2 on page 1428)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new silent = _ZX_1(silent);
  new rounding = _ZX_3(rounding);
stage RR:
  new argument3 = XVR[registerD];
  new argument2 = XVR[registerC];
  new argument1 = XVR[registerA];
  new XRM = rounding == 7 ? CS.XRM : rounding;
  new result1 = msbfif_32_32_x8(XRM, argument1, argument2, argument3);
stage E1:
  CS.XMF = 1;
stage E4:
  if (silent == 0) {
    new CS_xun = CS.XUN;
    new CS_xin = CS.XIN;
    new CS_xio = CS.XIO;
    new CS_xov = CS.XOV;
    CS.XUN = funderflow() | CS_xun;
    CS.XIN = finexact() | CS_xin;
    CS.XIO = finvalid() | CS_xio;
    CS.XOV = foverflow() | CS_xov;
  }
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.35 XNANDO Bitwise Not And Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0001

P	00	00	111	registerA	11	0001	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xnando` *registerA* = *registerB*, *registerC*

Description Bitwise not of bitwise and of the *registerB* with the *registerC*. The result is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = ~(argument2 & argument3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.36 XNEORO Bitwise Not Exclusive Or Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0101

P	00	00	111	registerA	11	0101	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xneoro` *registerA* = *registerB*, *registerC*

Description Bitwise not of bitwise exclusive or of the *registerB* with the *registerC*. The result is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = ~(argument2 ^ argument3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.37 XNIORO Bitwise Not Or Between Octuple Words

Encoding Format EXT_XBWO (Section 6.12.46), exucode5 = 0011

P	00	00	111	registerA	11	0011	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xnioro` *registerA* = *registerB*, *registerC*

Description Bitwise not of bitwise inclusive or of the *registerB* with the *registerC*. The result is stored into the *registerA*.

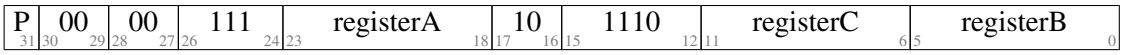
Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1 = ~(argument2 | argument3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.38 XSBMM8DQ Swapped Bit Matrix Multiplication 8×8 Double Quadruple

Encoding Format EXT_XSBMM8 (Section 6.12.45), exucode5 = 1110



Syntax `xsbmm8dq` *registerA* = *registerB*, *registerC*

Description The *registerC* and the *registerB* are interpreted as a vector of four 64-bit elements. Each *registerC* element interpreted as a 8×8 bit matrix is multiplied by the corresponding *registerB* element interpreted as a 8×8 bit matrix. The resulting 8×8 bit matrices are packed and stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1_0 = _BMM_8(_ZX_64(argument3.64[0]), _ZX_64(argument2.64[0]));
  new result1_1 = _BMM_8(_ZX_64(argument3.64[1]), _ZX_64(argument2.64[1]));
  new result1_2 = _BMM_8(_ZX_64(argument3.64[2]), _ZX_64(argument2.64[2]));
  new result1_3 = _BMM_8(_ZX_64(argument3.64[3]), _ZX_64(argument2.64[3]));
  new result1 = join_64_x4(result1_0, result1_1, result1_2, result1_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.39 XSBMMT8DQ Swapped Bit Matrix Multiplication 8×8 Double Quadruple

Encoding Format EXT_XSBMM8 (Section 6.12.45), exucode5 = 1111

P	00	00	111	registerA	10	1111	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Syntax `xsbmmt8dq` *registerA* = *registerB*, *registerC*

Description The *registerC* and the *registerB* are interpreted as a vector of four 64-bit elements. Each *registerC* element interpreted as a 8×8 bit matrix is multiplied by the corresponding *registerB* element interpreted as a 8×8 bit matrix. The resulting 8×8 bit matrices are transposed, packed and stored into the *registerA*.

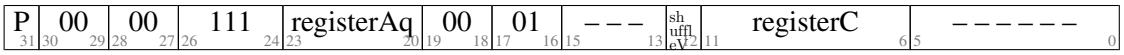
Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument3 = XVR[registerC];
  new argument2 = XVR[registerB];
  new result1_0 = _BMT_8(_BMM_8(_ZX_64(argument3.64[0]), _ZX_64(argument2
      .64[0])));
  new result1_1 = _BMT_8(_BMM_8(_ZX_64(argument3.64[1]), _ZX_64(argument2
      .64[1])));
  new result1_2 = _BMT_8(_BMM_8(_ZX_64(argument3.64[2]), _ZX_64(argument2
      .64[2])));
  new result1_3 = _BMT_8(_BMM_8(_ZX_64(argument3.64[3]), _ZX_64(argument2
      .64[3])));
  new result1 = join_64_x4(result1_0, result1_1, result1_2, result1_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

6.9.40 XSPLATOV — Splat Word Octuple to Vector

Encoding Format EXT_XSPLATOV (Section 6.12.43), exucode4 = 01



Syntax `xsplatov`*shuffleV registerAq* = *registerC*

Description The *registerC* is duplicated to produce a vector of words (1024 bits). Depending on the *shuffleV*, the 4×4 double words that compose these 1024 bits are transposed and the result is stored back into the *registerAq*.

Modifier(s)

- shuffleV (cf. shuffleV in section 6.10.2 on page 1426)

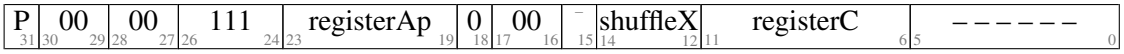
Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new shuffleV = _ZX_1(shuffleV);
stage RR:
  new argument2 = XVR[registerC];
  new argument2_0 = argument2;
  new argument2_1 = argument2;
  new argument2_2 = argument2;
  new argument2_3 = argument2;
  new result1_0 = argument2_0;
  new result1_1 = argument2_1;
  new result1_2 = argument2_2;
  new result1_3 = argument2_3;
  if (shuffleV) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_1.64[0], argument2_2
        .64[0], argument2_3.64[0]);
    result1_1 = join_64_x4(argument2_0.64[1], argument2_1.64[1], argument2_2
        .64[1], argument2_3.64[1]);
    result1_2 = join_64_x4(argument2_0.64[2], argument2_1.64[2], argument2_2
        .64[2], argument2_3.64[2]);
    result1_3 = join_64_x4(argument2_0.64[3], argument2_1.64[3], argument2_2
        .64[3], argument2_3.64[3]);
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XMR[registerAq]:0 = result1_0;
  XMR[registerAq]:1 = result1_1;
  XMR[registerAq]:2 = result1_2;
  XMR[registerAq]:3 = result1_3;
```


Kalray – all rights reserved

6.9.41 XSPLATOX Splat Word Octuple to Hexadecuple

Encoding Format EXT_XSPLATOX (Section 6.12.41), exucode4 = 00



Syntax `xsplatox`*shuffleX registerAp = registerC*

Description The *registerC* is duplicated to produce a hexadecuple word (512 bits). Depending on the *shuffleX*, the eight double words that compose these 512 bits are shuffled and the result is stored back into the *registerAp*.

Modifier(s)

- shuffleX (cf. shuffleX in section 6.10.2 on page 1427)

Execution Reservation table EXT (Section 6.14.26)

```
stage ID:
  new shuffleX = _ZX_3(shuffleX);
stage RR:
  new argument2 = XVR[registerC];
  new argument2_0 = argument2;
  new argument2_1 = argument2;
  new result1_0 = argument2_0;
  new result1_1 = argument2_1;
  if (shuffleX == 1) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_1.64[0], argument2_0
        .64[1], argument2_1.64[1]);
    result1_1 = join_64_x4(argument2_0.64[2], argument2_1.64[2], argument2_0
        .64[3], argument2_1.64[3]);
  }
  if (shuffleX == 2) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_0.64[2], argument2_1
        .64[0], argument2_1.64[2]);
    result1_1 = join_64_x4(argument2_0.64[1], argument2_0.64[3], argument2_1
        .64[1], argument2_1.64[3]);
  }
  if (shuffleX == 3) {
    result1_0 = join_64_x4(argument2_0.64[0], argument2_0.64[1], argument2_1
        .64[0], argument2_1.64[1]);
    result1_1 = join_64_x4(argument2_0.64[2], argument2_0.64[3], argument2_1
        .64[2], argument2_1.64[3]);
  }
  if (shuffleX == 4) {
    result1_0 = join_32_x8(argument2_0.32[0], argument2_0.32[2], argument2_0
        .32[1], argument2_0.32[3], argument2_1.32[0], argument2_1.32[2],
       argument2_1.32[1], argument2_1.32[3]);
```


Kalray – all rights reserved

```
    result1_1 = join_32_x8(argument2_0.32[4], argument2_0.32[6], argument2_0
        .32[5], argument2_0.32[7], argument2_1.32[4], argument2_1.32[6],
        argument2_1.32[5], argument2_1.32[7]);
  }
  if (shuffleX == 5) {
    result1_0 = join_32_x8(argument2_0.32[0], argument2_1.32[0], argument2_0
        .32[1], argument2_1.32[1], argument2_0.32[2], argument2_1.32[2],
        argument2_0.32[3], argument2_1.32[3]);
    result1_1 = join_32_x8(argument2_0.32[4], argument2_1.32[4], argument2_0
        .32[5], argument2_1.32[5], argument2_0.32[6], argument2_1.32[6],
        argument2_0.32[7], argument2_1.32[7]);
  }
  if (shuffleX == 6) {
    result1_0 = join_32_x8(argument2_0.32[0], argument2_0.32[2], argument2_0
        .32[4], argument2_0.32[6], argument2_1.32[0], argument2_1.32[2],
        argument2_1.32[4], argument2_1.32[6]);
    result1_1 = join_32_x8(argument2_0.32[1], argument2_0.32[3], argument2_0
        .32[5], argument2_0.32[7], argument2_1.32[1], argument2_1.32[3],
        argument2_1.32[5], argument2_1.32[7]);
  }
stage E1:
  CS.XMF = 1;
stage E3:
  XTR[registerAp]:0 = result1_0;
  XTR[registerAp]:1 = result1_1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.42 XSX48BW Sign Extend 4x8 Matrix Byte to Word

Encoding Format EXT_XEXTBW (Section 6.12.40), tcacode3 = 00

P	00	00	110	registerAq	00	01	– – – –	registerC	– – – – – –
31	30 29	28 27	26 24	23 20	19 18	17 16	15 12	11 6	5 0

Syntax `xsx48bw` *registerAq* = *registerC*

Description The *registerC* is interpreted a 4x8 matrix of bytes. Each byte element is sign-extended to 32 bits, and the resulting 4x8 matrix of integer words is stored into the *registerAq*.

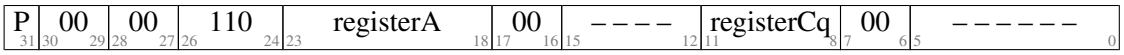
Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.64[0];
  new argument2_1 = argument2.64[1];
  new argument2_2 = argument2.64[2];
  new argument2_3 = argument2.64[3];
  new row_0 = sx_8_32_x8(argument2_0);
  new row_1 = sx_8_32_x8(argument2_1);
  new row_2 = sx_8_32_x8(argument2_2);
  new row_3 = sx_8_32_x8(argument2_3);
  new result1_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new result1_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new result1_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new result1_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XMR[registerAq]:0 = result1_0;
  XMR[registerAq]:1 = result1_1;
  XMR[registerAq]:2 = result1_2;
  XMR[registerAq]:3 = result1_3;
```


Kalray – all rights reserved

6.9.43 XTRUNC48WB Truncate 4x8 Matrix Word to Byte

Encoding Format EXT_TRUNCWB (Section 6.12.39), exucode4 = 00



Syntax `xtrunc48wb` *registerA* = *registerCq*

Description The *registerCq* is interpreted as a 4x8 matrix of signed integer words. This matrix is truncated element-wise to a 4x8 matrix of bytes, which is stored into the *registerA*.

Execution Reservation table EXT (Section 6.14.26)

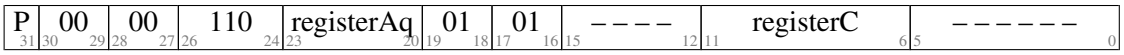
```
stage RR:
  new argument2_0 = XMR[registerCq]:0;
  new argument2_1 = XMR[registerCq]:1;
  new argument2_2 = XMR[registerCq]:2;
  new argument2_3 = XMR[registerCq]:3;
  new row_0 = transpose_64_4x4_0(argument2_0, argument2_1, argument2_2,
      argument2_3);
  new row_1 = transpose_64_4x4_1(argument2_0, argument2_1, argument2_2,
      argument2_3);
  new row_2 = transpose_64_4x4_2(argument2_0, argument2_1, argument2_2,
      argument2_3);
  new row_3 = transpose_64_4x4_3(argument2_0, argument2_1, argument2_2,
      argument2_3);
  new result1_0 = trunc_32_8_x8(row_0);
  new result1_1 = trunc_32_8_x8(row_1);
  new result1_2 = trunc_32_8_x8(row_2);
  new result1_3 = trunc_32_8_x8(row_3);
  new result1 = join_64_x4(result1_0, result1_1, result1_2, result1_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XVR[registerA] = result1;
```


Kalray – all rights reserved

©2025 Kalray

6.9.44 XZX48BW Zero Extend 4x8 Matrix Byte to Word

Encoding Format EXT_XEXTBW (Section 6.12.40), tcacode3 = 01



Syntax `xzx48bw` *registerAq* = *registerC*

Description The *registerC* is interpreted a 4x8 matrix of bytes. Each byte element is zero-extended to 32 bits, and the resulting 4x8 matrix of integer words is stored into the *registerAq*.

Execution Reservation table EXT (Section 6.14.26)

```
stage RR:
  new argument2 = XVR[registerC];
  new argument2_0 = argument2.64[0];
  new argument2_1 = argument2.64[1];
  new argument2_2 = argument2.64[2];
  new argument2_3 = argument2.64[3];
  new row_0 = zx_8_32_x8(argument2_0);
  new row_1 = zx_8_32_x8(argument2_1);
  new row_2 = zx_8_32_x8(argument2_2);
  new row_3 = zx_8_32_x8(argument2_3);
  new result1_0 = transpose_64_4x4_0(row_0, row_1, row_2, row_3);
  new result1_1 = transpose_64_4x4_1(row_0, row_1, row_2, row_3);
  new result1_2 = transpose_64_4x4_2(row_0, row_1, row_2, row_3);
  new result1_3 = transpose_64_4x4_3(row_0, row_1, row_2, row_3);
stage E1:
  CS.XMF = 1;
stage E3:
  XMR[registerAq]:0 = result1_0;
  XMR[registerAq]:1 = result1_1;
  XMR[registerAq]:2 = result1_2;
  XMR[registerAq]:3 = result1_3;
```


Kalray – all rights reserved

6.10 Instruction operands

6.10.1 Immediate classes

Immediate classes	Operands	What
brknumber	brknumber	Break Number
pcrel17	pcrel17	CB Offset
pcrel27	pcrel27	GOTO / CALL Offset
signed10	signed10	Sign Extended Immediate 10 bits
signed16	signed16	Sign Extended Immediate 16 bits
signed27	offset27	Sign Extended Immediate 27 bits
signed37	upper27_lower10	Sign Extended Immediate 37 bits
signed43	extend6_upper27_lower10	Sign Extended Immediate 43 bits
signed54	extend27_offset27	Sign Extended Immediate 54 bits
sysnumber	sysnumber	System Number
unsigned6	startbit, stopbit2_stopbit4, unsigned6	Zero Extended Immediate 6 bits
wrapped32	upper27_lower5	Wrapped Immediate 32 bits
wrapped64	extend27_upper27_lower10	Wrapped Immediate 64 bits

Kalray – all rights reserved

©2025 Kalray

Immediate classes	Extend	Shift	Width	Values
brknumber	Unsigned	0	2	0x0 to 0x3
pcrel17	Signed	2	17	0x40000 to 0x3fffc
pcrel27	Signed	2	27	0x10000000 to 0xfffffc
signed10	Signed	0	10	0x200 to 0x1ff
signed16	Signed	0	16	0x8000 to 0x7fff
signed27	Signed	0	27	0x4000000 to 0x3ffffff
signed37	Signed	0	37	0x1000000000 to 0xfffffffff
signed43	Signed	0	43	0x40000000000 to 0x3fffffffff
signed54	Signed	0	54	0x20000000000000 to 0x1fffffffffffff
sysnumber	Unsigned	0	12	0x0 to 0xfff
unsigned6	Unsigned	0	6	0x0 to 0x3f
wrapped32	Wrap	0	32	0x80000000 to 0xffffffff
wrapped64	Wrap	0	64	0x8000000000000000 to 0x7fffffffffffffff

Kalray – all rights reserved

6.10.2 Operand Modifiers

Operand Modifiers	Instances
accesses	accesses
boolcas	boolcas
cachelev	cachelev
channel	rchannel, schannel
coherency	coherency
comparison	comparison
conjugate	conjugate
doscale	doscale
floatcomp	floatcomp
lsomask	lsomask
lsumask	lsumask
qindex	qindex
rounding	rounding, rounding2
scalarcond	branchcond, lsucond, scalarcond
shuffleV	shuffleV
shuffleX	shuffleX
silent	silent, silent2
simdcond	simdcond
splat32	splat32
transpose	multransm_multransl
variant	variant

Kalray – all rights reserved

©2025 Kalray

Modifier accesses:

Name	Case	Value
.	All Accesses	0
.W	Write Accesses	1
.R	Read Accesses	2
.WA	Write Accesses (asynchronous)	3

Kalray – all rights reserved

Modifier boolcas:

Name	Case	Value
.V	Valued CAS	0
.	Boolean CAS	1

Kalray – all rights reserved

©2025 Kalray

Modifier cachelev:

Name	Case	Value
.L1	L1 Cache	0
.L2	L2 Cache	1

Kalray – all rights reserved

Modifier channel:

Name	Case	Value
.F	Forward Channel	0
.B	Backward Channel	1

Kalray – all rights reserved

©2025 Kalray

Modifier coherency:

Name	Case	Value
.	Local	0
.G	Global	1
.S	Reserved	2

Kalray – all rights reserved

Modifier comparison:

Name	Case	Value
.NE	Not Equal	0
.EQ	Equal	1
.LT	Less Than	2
.GE	Greater Than or Equal	3
.LE	Less Than or Equal	4
.GT	Greater Than	5
.LTU	Less Than Unsigned	6
.GEU	Greater Than or Equal Unsigned	7
.LEU	Less Than or Equal Unsigned	8
.GTU	Greater Than Unsigned	9
.ALL	All Bits Set in Mask	10
.NALL	Not All Bits Set in Mask	11
.ANY	Any Bits Set in Mask	12
.NONE	Not Any Bits Set in Mask	13

Kalray – all rights reserved

©2025 Kalray

Modifier conjugate:

Name	Case	Value
.	Default	0
.C	Conjugate	1

Kalray – all rights reserved

Modifier doscale:

Name	Case	Value
.	Scale by 1	0
.XS	Scale by sizeof	1

Kalray – all rights reserved

©2025 Kalray

Modifier floatcomp:

Name	Case	Value
.ONE	Ordered and Not Equal	0
.UEQ	Unordered or Equal	1
.OEQ	Ordered and Equal	2
.UNE	Unordered or Not Equal	3
.OLT	Ordered and Less Than	4
.UGE	Unordered or Greater Than or Equal	5
.OGE	Ordered and Greater Than or Equal	6
.ULT	Unordered or Less Than	7

Kalray – all rights reserved

Modifier Isomask:

Name	Case	Value
.U0	Unused 0	0
.U1	Unused 1	1
.U2	Unused 2	2
.U3	Unused 3	3
.MT	Mask True	4
.MF	Mask False	5
.MTC	Mask True or Clear	6
.MFC	Mask False or Clear	7

Kalray – all rights reserved

©2025 Kalray

Modifier lsumask:

Name	Case	Value
.DNEZ	Double Not Equal to Zero	0
.DEQZ	Double Equal to Zero	1
.WNEZ	Word Not Equal to Zero	2
.WEQZ	Word Equal to Zero	3
.MT	Mask True	4
.MF	Mask False	5
.MTC	Mask True or Clear	6
.MFC	Mask False or Clear	7

Kalray – all rights reserved

Modifier qindex:

Name	Case	Value
.Q0	Quarter 0	0
.Q1	Quarter 1	1
.Q2	Quarter 2	2
.Q3	Quarter 3	3

Kalray – all rights reserved

©2025 Kalray

Modifier rounding:

Name	Case	Value
.RN	Round to Nearest, ties to Even	0
.RU	Round Upward	1
.RD	Round Downward	2
.RZ	Round toward Zero	3
.R4	Reserved 4	4
.R5	Reserved 5	5
.RO	Reserved 6	6
.	Use CS rounding	7

Kalray – all rights reserved

Modifier scalarcond:

Name	Case	Value
.DNEZ	Double Not Equal to Zero	0
.DEQZ	Double Equal to Zero	1
.DLTZ	Double Less Than Zero	2
.DGEZ	Double Greater Than or Equal to Zero	3
.DLEZ	Double Less Than or Equal to Zero	4
.DGTZ	Double Greater Than Zero	5
.ODD	Odd (LSB Set)	6
.EVEN	Even (LSB Clear)	7
.WNEZ	Word Not Equal to Zero	8
.WEQZ	Word Equal to Zero	9
.WLTZ	Word Less Than Zero	10
.WGEZ	Word Greater Than or Equal to Zero	11
.WLEZ	Word Less Than or Equal to Zero	12
.WGTZ	Word Greater Than Zero	13

Kalray – all rights reserved

©2025 Kalray

Modifier shuffleV:

Name	Case	Value
.	No Shuffle	0
.TD	Transpose Double Words 4x4	1

Kalray – all rights reserved

Modifier shuffleX:

Name	Case	Value
.	No Shuffle	0
.ZD	Zip Double Words	1
.UD	Unzip Double Words	2
.TQ	Transpose Quadruple Words 2x2	3
.TW	Transpose Words 4x4	4
.ZW	Zip Words	5
.UW	Unzip Words	6

Kalray – all rights reserved

©2025 Kalray

Modifier silent:

Name	Case	Value
.	Effects on CS	0
.S	Silent on CS	1

Kalray – all rights reserved

Modifier simdcond:

Name	Case	Value
.NEZ	Not Equal to Zero	0
.EQZ	Equal to Zero	1
.LTZ	Less Than Zero	2
.GEZ	Greater Than or Equal to Zero	3
.LEZ	Less Than or Equal to Zero	4
.GTZ	Greater Than Zero	5
.ODD	Odd (LSB Set)	6
.EVEN	Even (LSB Clear)	7

Kalray – all rights reserved

©2025 Kalray

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

Modifier transpose:

Name	Case	Value
.	Normal Normal	0
.TN	Transpose Normal	1
.NT	Normal Transpose	2
.TT	Transpose Transpose	3

Kalray – all rights reserved

©2025 Kalray

Modifier variant:

Name	Case	Value
.	Cached	0
.S	Speculative	1
.U	Uncached	2
.US	Uncached Speculative	3

Kalray – all rights reserved

6.10.3 Register Classes

Register Classes	Instances	Width
aloneReg	systemAlone	64
buffer16Reg	registerCj, registerGj	0
buffer2Reg	registerCg, registerGg	0
buffer32Reg	registerCk, registerGk	0
buffer4Reg	registerCh, registerGh	0
buffer64Reg	registerCl, registerGl	0
buffer8Reg	registerCi, registerGi	0
onlyfxReg	systemT2	64
onlygetReg	systemS2	64
onlyraReg	systemRA	64
onlysetReg	systemT3	64
onlyswapReg	systemS4	64
pairedReg	registerM, registerO, registerP, registerU	128
quadReg	registerN, registerQ, registerR, registerV	256
singleReg	registerT, registerW, registerY, registerZ	64
xworddRegLo	registerCc_qselectC	64
xworddRegLo0M4	registerAx	64
xworddRegLo1M4	registerAy	64
xworddRegLo2M4	registerAz	64
xworddRegLo3M4	registerAt	64
xwordoRegLo	registerA, registerB, registerC, registerD, registerE, registerG	256
xwordqRegLo	registerCb_hselectC	128
xwordqRegLoE	registerAE	128
xwordqRegLoO	registerAO	128
xwordvRegLo	registerAq, registerCq, registerGq	1024
xwordxRegLo	registerAp, registerCp, registerDp	512

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
aloneReg	SFR	$mmc $s28 $ps $s1 $syow $s96 $htow $s97 $itow $s98 $dow $s99 $mow $s100 $psow $s101 $sps $s132 $sps_pl0 $s68 $sps_pl1 $s69 $sps_pl2 $s70 $sps_pl3 $s71

Kalray – all rights reserved

Register Class	Register File	Registers
buffer16Reg	X16R	$a0..a15 $a16..a31 $a32..a47 $a48..a63

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
buffer2Reg	X2R	$a0..a1
		$a2..a3
		$a4..a5
		$a6..a7
		$a8..a9
		$a10..a11
		$a12..a13
		$a14..a15
		$a16..a17
		$a18..a19
		$a20..a21
		$a22..a23
		$a24..a25
		$a26..a27
		$a28..a29
		$a30..a31
		$a32..a33
		$a34..a35
		$a36..a37
		$a38..a39
		$a40..a41
		$a42..a43
		$a44..a45
		$a46..a47
		$a48..a49
		$a50..a51
		$a52..a53
		$a54..a55
		$a56..a57
		$a58..a59
		$a60..a61
		$a62..a63

Kalray – all rights reserved

Register Class	Register File	Registers
buffer32Reg	X32R	`$a0..a31` `$a32..a63`

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
buffer4Reg	X4R	$a0..a3 $a4..a7 $a8..a11 $a12..a15 $a16..a19 $a20..a23 $a24..a27 $a28..a31 $a32..a35 $a36..a39 $a40..a43 $a44..a47 $a48..a51 $a52..a55 $a56..a59 $a60..a63

Kalray – all rights reserved

Register Class	Register File	Registers
buffer64Reg	X64R	$a0..a63

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
buffer8Reg	X8R	$a0..a7 $a8..a15 $a16..a23 $a24..a31 $a32..a39 $a40..a47 $a48..a55 $a56..a63

Kalray – all rights reserved

Register Class	Register File	Registers
onlyfxReg	SFR	$pcr $s2 $cs $s4 $csit $s5 $ipe $s10 $men $s11 $pmc $s12 $tcr $s18 $ile $s25 $ill $s26 $ilr $s27 $mmc $s28 $tel $s29 $teh $s30 $ixc $s31 $mes $s45 $ws $s46 $dc0 $s47 $dc1 $s48 $dc2 $s49 $dc3 $s50 $pmc2 $s61 $sps_pl0 $s68 $sps_pl1 $s69 $sps_pl2 $s70 $sps_pl3 $s71 $es_pl0 $s84 $es_pl1 $s85 $es_pl2 $s86 $es_pl3 $s87 $sps $s132 $es $s148 $tpcc_pl0 $s104 $tpcc_pl1 $s105 $tpcc_pl2 $s106 $tpcc_pl3 $s107 $tpcc $s168

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
onlygetReg	SFR	$pc $s0 $ps $s1 $pcr $s2 $ra $s3 $cs $s4 $csit $s5 $aespc $s6 $ls $s7 $le $s8 $lc $s9 $ipe $s10 $men $s11 $pmc $s12 $pm0 $s13 $pm1 $s14 $pm2 $s15 $pm3 $s16 $pmsa $s17 $tcr $s18 $t0v $s19 $t1v $s20 $t0r $s21 $t1r $s22 $wdv $s23 $wdr $s24 $ile $s25 $ill $s26 $ilr $s27 $mmc $s28 $tel $s29 $teh $s30 $ixc $s31 $syo $s32 $hto $s33 $ito $s34 $do $s35 $mo $s36 $pso $s37 $tpcm0 $s38 $tpcm1 $s39 $res40 $s40 $dba0 $s41 $dba1 $s42 $dwa0 $s43 $dwa1 $s44 $mes $s45 $ws $s46 $dc0 $s47 $dc1 $s48 $dc2 $s49 $dc3 $s50 $dba2 $s51 $dba3 $s52 $dwa2 $s53 $dwa3 $s54 $tpcm2 $s55 $tpcmc $s56 $pm4 $s57 $pm5 $s58 $pm6 $s59 $pm7 $s60 $pmc2 $s61 $srhpc $s62 $frcc $s63 $spc_pl0 $s64 $spc_pl1 $s65 $spc_pl2 $s66 $spc_pl3 $s67 $sps_pl0 $s68 $sps_pl1 $s69 $sps_pl2 $s70 $sps_pl3 $s71 $ea_pl0 $s72 $ea_pl1 $s73 $ea_pl2 $s74

Kalray – all rights reserved

Register Class	Register File	Registers
onlyraReg	SFR	`$ra $s3`

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
onlysetReg	SFR	$ps $s1 $pcr $s2 $ra $s3 $cs $s4 $csit $s5 $aespc $s6 $ls $s7 $le $s8 $lc $s9 $ipe $s10 $men $s11 $pmc $s12 $pm0 $s13 $pm1 $s14 $pm2 $s15 $pm3 $s16 $pmsa $s17 $tcr $s18 $t0v $s19 $t1v $s20 $t0r $s21 $t1r $s22 $wdv $s23 $wdr $s24 $ile $s25 $ill $s26 $ilr $s27 $mmc $s28 $tel $s29 $teh $s30 $ixc $s31 $tpcm0 $s38 $tpcm1 $s39 $res40 $s40 $dba0 $s41 $dba1 $s42 $dwa0 $s43 $dwa1 $s44 $mes $s45 $ws $s46 $dc0 $s47 $dc1 $s48 $dc2 $s49 $dc3 $s50 $dba2 $s51 $dba3 $s52 $dwa2 $s53 $dwa3 $s54 $tpcm2 $s55 $tpcmc $s56 $pm4 $s57 $pm5 $s58 $pm6 $s59 $pm7 $s60 $pmc2 $s61 $srhpc $s62 $frcc $s63 $spc_pl0 $s64 $spc_pl1 $s65 $spc_pl2 $s66 $spc_pl3 $s67 $sps_pl0 $s68 $sps_pl1 $s69 $sps_pl2 $s70 $sps_pl3 $s71 $ea_pl0 $s72 $ea_pl1 $s73 $ea_pl2 $s74 $ea_pl3 $s75 $ev_pl0 $s76 $ev_pl1 $s77 $ev_pl2 $s78 $ev_pl3 $s79 $sr_pl0 $s80 $sr_pl1 $s81

Kalray – all rights reserved

Register Class	Register File	Registers
onlyswapReg	SFR	$pc $s0 $pcr $s2 $ra $s3 $cs $s4 $csit $s5 $aespc $s6 $ls $s7 $le $s8 $lc $s9 $ipe $s10 $men $s11 $pmc $s12 $pm0 $s13 $pm1 $s14 $pm2 $s15 $pm3 $s16 $pmsa $s17 $tcr $s18 $t0v $s19 $t1v $s20 $t0r $s21 $t1r $s22 $wdv $s23 $wdr $s24 $ile $s25 $ill $s26 $ilr $s27 $mmc $s28 $tel $s29 $teh $s30 $ixc $s31 $syo $s32 $hto $s33 $ito $s34 $do $s35 $mo $s36 $pso $s37 $tpcm0 $s38 $tpcm1 $s39 $res40 $s40 $dba0 $s41 $dba1 $s42 $dwa0 $s43 $dwa1 $s44 $mes $s45 $ws $s46 $dc0 $s47 $dc1 $s48 $dc2 $s49 $dc3 $s50 $dba2 $s51 $dba3 $s52 $dwa2 $s53 $dwa3 $s54 $tpcm2 $s55 $tpcmc $s56 $pm4 $s57 $pm5 $s58 $pm6 $s59 $pm7 $s60 $pmc2 $s61 $srhpc $s62 $frcc $s63 $spc_pl0 $s64 $spc_pl1 $s65 $spc_pl2 $s66 $spc_pl3 $s67 $sps_pl0 $s68 $sps_pl1 $s69 $sps_pl2 $s70 $sps_pl3 $s71 $ea_pl0 $s72 $ea_pl1 $s73 $ea_pl2 $s74 $ea_pl3 $s75

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
pairedReg	PGR	$r0r1 $r0r1r2r3.lo $r2r3 $r0r1r2r3.hi $r4r5 $r4r5r6r7.lo $r6r7 $r4r5r6r7.hi $r8r9 $r8r9r10r11.lo $r10r11 $r8r9r10r11.hi $r12r13 $r12r13r14r15.lo $r14r15 $r12r13r14r15.hi $r16r17 $r16r17r18r19.lo $r18r19 $r16r17r18r19.hi $r20r21 $r20r21r22r23.lo $r22r23 $r20r21r22r23.hi $r24r25 $r24r25r26r27.lo $r26r27 $r24r25r26r27.hi $r28r29 $r28r29r30r31.lo $r30r31 $r28r29r30r31.hi $r32r33 $r32r33r34r35.lo $r34r35 $r32r33r34r35.hi $r36r37 $r36r37r38r39.lo $r38r39 $r36r37r38r39.hi $r40r41 $r40r41r42r43.lo $r42r43 $r40r41r42r43.hi $r44r45 $r44r45r46r47.lo $r46r47 $r44r45r46r47.hi $r48r49 $r48r49r50r51.lo $r50r51 $r48r49r50r51.hi $r52r53 $r52r53r54r55.lo $r54r55 $r52r53r54r55.hi $r56r57 $r56r57r58r59.lo $r58r59 $r56r57r58r59.hi $r60r61 $r60r61r62r63.lo $r62r63 $r60r61r62r63.hi

Kalray – all rights reserved

Register Class	Register File	Registers
quadReg	QGR	$r0r1r2r3 $r4r5r6r7 $r8r9r10r11 $r12r13r14r15 $r16r17r18r19 $r20r21r22r23 $r24r25r26r27 $r28r29r30r31 $r32r33r34r35 $r36r37r38r39 $r40r41r42r43 $r44r45r46r47 $r48r49r50r51 $r52r53r54r55 $r56r57r58r59 $r60r61r62r63

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
singleReg	GPR	$r0 $r0r1.lo $r0r1r2r3.x $r1 $r0r1.hi $r0r1r2r3.y $r2 $r2r3.lo $r0r1r2r3.z $r3 $r2r3.hi $r0r1r2r3.t $r4 $r4r5.lo $r4r5r6r7.x $r5 $r4r5.hi $r4r5r6r7.y $r6 $r6r7.lo $r4r5r6r7.z $r7 $r6r7.hi $r4r5r6r7.t $r8 $r8r9.lo $r8r9r10r11.x $r9 $r8r9.hi $r8r9r10r11.y $r10 $r10r11.lo $r8r9r10r11.z $r11 $r10r11.hi $r8r9r10r11.t $r12 $sp $r13 $tp $r14 $fp $r14r15.lo $r15 $rp $r14r15.hi $r16 $r16r17.lo $r16r17r18r19.x $r17 $r16r17.hi $r16r17r18r19.y $r18 $r18r19.lo $r16r17r18r19.z $r19 $r18r19.hi $r16r17r18r19.t $r20 $r20r21.lo $r20r21r22r23.x $r21 $r20r21.hi $r20r21r22r23.y $r22 $r22r23.lo $r20r21r22r23.z $r23 $r22r23.hi $r20r21r22r23.t $r24 $r24r25.lo $r24r25r26r27.x $r25 $r24r25.hi $r24r25r26r27.y $r26 $r26r27.lo $r24r25r26r27.z $r27 $r26r27.hi $r24r25r26r27.t $r28 $r28r29.lo $r28r29r30r31.x $r29 $r28r29.hi $r28r29r30r31.y $r30 $r30r31.lo $r28r29r30r31.z $r31 $r30r31.hi $r28r29r30r31.t $r32 $r32r33.lo $r32r33r34r35.x $r33 $r32r33.hi $r32r33r34r35.y $r34 $r34r35.lo $r32r33r34r35.z $r35 $r34r35.hi $r32r33r34r35.t $r36 $r36r37.lo $r36r37r38r39.x $r37 $r36r37.hi $r36r37r38r39.y $r38 $r38r39.lo $r36r37r38r39.z $r39 $r38r39.hi $r36r37r38r39.t $r40 $r40r41.lo $r40r41r42r43.x $r41 $r40r41.hi $r40r41r42r43.y $r42 $r42r43.lo $r40r41r42r43.z $r43 $r42r43.hi $r40r41r42r43.t $r44 $r44r45.lo $r44r45r46r47.x $r45 $r44r45.hi $r44r45r46r47.y $r46 $r46r47.lo $r44r45r46r47.z $r47 $r46r47.hi $r44r45r46r47.t $r48 $r48r49.lo $r48r49r50r51.x $r49 $r48r49.hi $r48r49r50r51.y $r50 $r50r51.lo $r48r49r50r51.z $r51 $r50r51.hi $r48r49r50r51.t $r52 $r52r53.lo $r52r53r54r55.x $r53 $r52r53.hi $r52r53r54r55.y $r54 $r54r55.lo $r52r53r54r55.z $r55 $r54r55.hi $r52r53r54r55.t $r56 $r56r57.lo $r56r57r58r59.x $r57 $r56r57.hi $r56r57r58r59.y $r58 $r58r59.lo $r56r57r58r59.z $r59 $r58r59.hi $r56r57r58r59.t $r60 $r60r61.lo $r60r61r62r63.x $r61 $r60r61.hi $r60r61r62r63.y $r62 $r62r63.lo $r60r61r62r63.z $r63 $r62r63.hi $r60r61r62r63.t

Kalray – all rights reserved

Register Class	Register File	Registers
xworddRegLo	XCR	$a0_x $a0.x $a0_y $a0.y $a0_z $a0.z $a0_t $a0.t $a1_x $a1.x $a1_y $a1.y $a1_z $a1.z $a1_t $a1.t $a2_x $a2.x $a2_y $a2.y $a2_z $a2.z $a2_t $a2.t $a3_x $a3.x $a3_y $a3.y $a3_z $a3.z $a3_t $a3.t $a4_x $a4.x $a4_y $a4.y $a4_z $a4.z $a4_t $a4.t $a5_x $a5.x $a5_y $a5.y $a5_z $a5.z $a5_t $a5.t $a6_x $a6.x $a6_y $a6.y $a6_z $a6.z $a6_t $a6.t $a7_x $a7.x $a7_y $a7.y $a7_z $a7.z $a7_t $a7.t $a8_x $a8.x $a8_y $a8.y $a8_z $a8.z $a8_t $a8.t $a9_x $a9.x $a9_y $a9.y $a9_z $a9.z $a9_t $a9.t $a10_x $a10.x $a10_y $a10.y $a10_z $a10.z $a10_t $a10.t $a11_x $a11.x $a11_y $a11.y $a11_z $a11.z $a11_t $a11.t $a12_x $a12.x $a12_y $a12.y $a12_z $a12.z $a12_t $a12.t $a13_x $a13.x $a13_y $a13.y $a13_z $a13.z $a13_t $a13.t $a14_x $a14.x $a14_y $a14.y $a14_z $a14.z $a14_t $a14.t $a15_x $a15.x $a15_y $a15.y $a15_z $a15.z $a15_t $a15.t $a16_x $a16.x $a16_y $a16.y $a16_z $a16.z $a16_t $a16.t $a17_x $a17.x $a17_y $a17.y $a17_z $a17.z $a17_t $a17.t $a18_x $a18.x $a18_y $a18.y $a18_z $a18.z

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
xworddRegLo0M4	XCR	$a0_x $a0.x $a1_x $a1.x $a2_x $a2.x $a3_x $a3.x $a4_x $a4.x $a5_x $a5.x $a6_x $a6.x $a7_x $a7.x $a8_x $a8.x $a9_x $a9.x $a10_x $a10.x $a11_x $a11.x $a12_x $a12.x $a13_x $a13.x $a14_x $a14.x $a15_x $a15.x $a16_x $a16.x $a17_x $a17.x $a18_x $a18.x $a19_x $a19.x $a20_x $a20.x $a21_x $a21.x $a22_x $a22.x $a23_x $a23.x $a24_x $a24.x $a25_x $a25.x $a26_x $a26.x $a27_x $a27.x $a28_x $a28.x $a29_x $a29.x $a30_x $a30.x $a31_x $a31.x $a32_x $a32.x $a33_x $a33.x $a34_x $a34.x $a35_x $a35.x $a36_x $a36.x $a37_x $a37.x $a38_x $a38.x $a39_x $a39.x $a40_x $a40.x $a41_x $a41.x $a42_x $a42.x $a43_x $a43.x $a44_x $a44.x $a45_x $a45.x $a46_x $a46.x $a47_x $a47.x $a48_x $a48.x $a49_x $a49.x $a50_x $a50.x $a51_x $a51.x $a52_x $a52.x $a53_x $a53.x $a54_x $a54.x $a55_x $a55.x $a56_x $a56.x $a57_x $a57.x $a58_x $a58.x $a59_x $a59.x $a60_x $a60.x $a61_x $a61.x $a62_x $a62.x $a63_x $a63.x

Kalray – all rights reserved

Register Class	Register File	Registers
xworddRegLo1M4	XCR	$a0_y $a0.y $a1_y $a1.y $a2_y $a2.y $a3_y $a3.y $a4_y $a4.y $a5_y $a5.y $a6_y $a6.y $a7_y $a7.y $a8_y $a8.y $a9_y $a9.y $a10_y $a10.y $a11_y $a11.y $a12_y $a12.y $a13_y $a13.y $a14_y $a14.y $a15_y $a15.y $a16_y $a16.y $a17_y $a17.y $a18_y $a18.y $a19_y $a19.y $a20_y $a20.y $a21_y $a21.y $a22_y $a22.y $a23_y $a23.y $a24_y $a24.y $a25_y $a25.y $a26_y $a26.y $a27_y $a27.y $a28_y $a28.y $a29_y $a29.y $a30_y $a30.y $a31_y $a31.y $a32_y $a32.y $a33_y $a33.y $a34_y $a34.y $a35_y $a35.y $a36_y $a36.y $a37_y $a37.y $a38_y $a38.y $a39_y $a39.y $a40_y $a40.y $a41_y $a41.y $a42_y $a42.y $a43_y $a43.y $a44_y $a44.y $a45_y $a45.y $a46_y $a46.y $a47_y $a47.y $a48_y $a48.y $a49_y $a49.y $a50_y $a50.y $a51_y $a51.y $a52_y $a52.y $a53_y $a53.y $a54_y $a54.y $a55_y $a55.y $a56_y $a56.y $a57_y $a57.y $a58_y $a58.y $a59_y $a59.y $a60_y $a60.y $a61_y $a61.y $a62_y $a62.y $a63_y $a63.y

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
xworddRegLo2M4	XCR	$a0_z $a0.z $a1_z $a1.z $a2_z $a2.z $a3_z $a3.z $a4_z $a4.z $a5_z $a5.z $a6_z $a6.z $a7_z $a7.z $a8_z $a8.z $a9_z $a9.z $a10_z $a10.z $a11_z $a11.z $a12_z $a12.z $a13_z $a13.z $a14_z $a14.z $a15_z $a15.z $a16_z $a16.z $a17_z $a17.z $a18_z $a18.z $a19_z $a19.z $a20_z $a20.z $a21_z $a21.z $a22_z $a22.z $a23_z $a23.z $a24_z $a24.z $a25_z $a25.z $a26_z $a26.z $a27_z $a27.z $a28_z $a28.z $a29_z $a29.z $a30_z $a30.z $a31_z $a31.z $a32_z $a32.z $a33_z $a33.z $a34_z $a34.z $a35_z $a35.z $a36_z $a36.z $a37_z $a37.z $a38_z $a38.z $a39_z $a39.z $a40_z $a40.z $a41_z $a41.z $a42_z $a42.z $a43_z $a43.z $a44_z $a44.z $a45_z $a45.z $a46_z $a46.z $a47_z $a47.z $a48_z $a48.z $a49_z $a49.z $a50_z $a50.z $a51_z $a51.z $a52_z $a52.z $a53_z $a53.z $a54_z $a54.z $a55_z $a55.z $a56_z $a56.z $a57_z $a57.z $a58_z $a58.z $a59_z $a59.z $a60_z $a60.z $a61_z $a61.z $a62_z $a62.z $a63_z $a63.z

Kalray – all rights reserved

Register Class	Register File	Registers
xworddRegLo3M4	XCR	$a0_t $a0.t $a1_t $a1.t $a2_t $a2.t $a3_t $a3.t $a4_t $a4.t $a5_t $a5.t $a6_t $a6.t $a7_t $a7.t $a8_t $a8.t $a9_t $a9.t $a10_t $a10.t $a11_t $a11.t $a12_t $a12.t $a13_t $a13.t $a14_t $a14.t $a15_t $a15.t $a16_t $a16.t $a17_t $a17.t $a18_t $a18.t $a19_t $a19.t $a20_t $a20.t $a21_t $a21.t $a22_t $a22.t $a23_t $a23.t $a24_t $a24.t $a25_t $a25.t $a26_t $a26.t $a27_t $a27.t $a28_t $a28.t $a29_t $a29.t $a30_t $a30.t $a31_t $a31.t $a32_t $a32.t $a33_t $a33.t $a34_t $a34.t $a35_t $a35.t $a36_t $a36.t $a37_t $a37.t $a38_t $a38.t $a39_t $a39.t $a40_t $a40.t $a41_t $a41.t $a42_t $a42.t $a43_t $a43.t $a44_t $a44.t $a45_t $a45.t $a46_t $a46.t $a47_t $a47.t $a48_t $a48.t $a49_t $a49.t $a50_t $a50.t $a51_t $a51.t $a52_t $a52.t $a53_t $a53.t $a54_t $a54.t $a55_t $a55.t $a56_t $a56.t $a57_t $a57.t $a58_t $a58.t $a59_t $a59.t $a60_t $a60.t $a61_t $a61.t $a62_t $a62.t $a63_t $a63.t

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
xwordoRegLo	XVR	$a0 $a0a1.lo $a0a1a2a3.x $a1 $a0a1.hi $a0a1a2a3.y $a2 $a2a3.lo $a0a1a2a3.z $a3 $a2a3.hi $a0a1a2a3.t $a4 $a4a5.lo $a4a5a6a7.x $a5 $a4a5.hi $a4a5a6a7.y $a6 $a6a7.lo $a4a5a6a7.z $a7 $a6a7.hi $a4a5a6a7.t $a8 $a8a9.lo $a8a9a10a11.x $a9 $a8a9.hi $a8a9a10a11.y $a10 $a10a11.lo $a8a9a10a11.z $a11 $a10a11.hi $a8a9a10a11.t $a12 $a12a13.lo $a12a13a14a15.x $a13 $a12a13.hi $a12a13a14a15.y $a14 $a14a15.lo $a12a13a14a15.z $a15 $a14a15.hi $a12a13a14a15.t $a16 $a16a17.lo $a16a17a18a19.x $a17 $a16a17.hi $a16a17a18a19.y $a18 $a18a19.lo $a16a17a18a19.z $a19 $a18a19.hi $a16a17a18a19.t $a20 $a20a21.lo $a20a21a22a23.x $a21 $a20a21.hi $a20a21a22a23.y $a22 $a22a23.lo $a20a21a22a23.z $a23 $a22a23.hi $a20a21a22a23.t $a24 $a24a25.lo $a24a25a26a27.x $a25 $a24a25.hi $a24a25a26a27.y $a26 $a26a27.lo $a24a25a26a27.z $a27 $a26a27.hi $a24a25a26a27.t $a28 $a28a29.lo $a28a29a30a31.x $a29 $a28a29.hi $a28a29a30a31.y $a30 $a30a31.lo $a28a29a30a31.z $a31 $a30a31.hi $a28a29a30a31.t $a32 $a32a33.lo $a32a33a34a35.x $a33 $a32a33.hi $a32a33a34a35.y $a34 $a34a35.lo $a32a33a34a35.z $a35 $a34a35.hi $a32a33a34a35.t $a36 $a36a37.lo $a36a37a38a39.x $a37 $a36a37.hi $a36a37a38a39.y $a38 $a38a39.lo $a36a37a38a39.z $a39 $a38a39.hi $a36a37a38a39.t $a40 $a40a41.lo $a40a41a42a43.x $a41 $a40a41.hi $a40a41a42a43.y $a42 $a42a43.lo $a40a41a42a43.z $a43 $a42a43.hi $a40a41a42a43.t $a44 $a44a45.lo $a44a45a46a47.x $a45 $a44a45.hi $a44a45a46a47.y $a46 $a46a47.lo $a44a45a46a47.z $a47 $a46a47.hi $a44a45a46a47.t $a48 $a48a49.lo $a48a49a50a51.x $a49 $a48a49.hi $a48a49a50a51.y $a50 $a50a51.lo $a48a49a50a51.z $a51 $a50a51.hi $a48a49a50a51.t $a52 $a52a53.lo $a52a53a54a55.x $a53 $a52a53.hi $a52a53a54a55.y $a54 $a54a55.lo $a52a53a54a55.z $a55 $a54a55.hi $a52a53a54a55.t $a56 $a56a57.lo $a56a57a58a59.x $a57 $a56a57.hi $a56a57a58a59.y $a58 $a58a59.lo $a56a57a58a59.z $a59 $a58a59.hi $a56a57a58a59.t $a60 $a60a61.lo $a60a61a62a63.x $a61 $a60a61.hi $a60a61a62a63.y $a62 $a62a63.lo $a60a61a62a63.z $a63 $a62a63.hi $a60a61a62a63.t

Kalray – all rights reserved

Register Class	Register File	Registers
xwordqRegLo	XBR	$a0_lo $a0.lo $a0_hi $a0.hi $a1_lo $a1.lo $a1_hi $a1.hi $a2_lo $a2.lo $a2_hi $a2.hi $a3_lo $a3.lo $a3_hi $a3.hi $a4_lo $a4.lo $a4_hi $a4.hi $a5_lo $a5.lo $a5_hi $a5.hi $a6_lo $a6.lo $a6_hi $a6.hi $a7_lo $a7.lo $a7_hi $a7.hi $a8_lo $a8.lo $a8_hi $a8.hi $a9_lo $a9.lo $a9_hi $a9.hi $a10_lo $a10.lo $a10_hi $a10.hi $a11_lo $a11.lo $a11_hi $a11.hi $a12_lo $a12.lo $a12_hi $a12.hi $a13_lo $a13.lo $a13_hi $a13.hi $a14_lo $a14.lo $a14_hi $a14.hi $a15_lo $a15.lo $a15_hi $a15.hi $a16_lo $a16.lo $a16_hi $a16.hi $a17_lo $a17.lo $a17_hi $a17.hi $a18_lo $a18.lo $a18_hi $a18.hi $a19_lo $a19.lo $a19_hi $a19.hi $a20_lo $a20.lo $a20_hi $a20.hi $a21_lo $a21.lo $a21_hi $a21.hi $a22_lo $a22.lo $a22_hi $a22.hi $a23_lo $a23.lo $a23_hi $a23.hi $a24_lo $a24.lo $a24_hi $a24.hi $a25_lo $a25.lo $a25_hi $a25.hi $a26_lo $a26.lo $a26_hi $a26.hi $a27_lo $a27.lo $a27_hi $a27.hi $a28_lo $a28.lo $a28_hi $a28.hi $a29_lo $a29.lo $a29_hi $a29.hi $a30_lo $a30.lo $a30_hi $a30.hi $a31_lo $a31.lo $a31_hi $a31.hi $a32_lo $a32.lo $a32_hi $a32.hi $a33_lo $a33.lo $a33_hi $a33.hi $a34_lo $a34.lo $a34_hi $a34.hi $a35_lo $a35.lo $a35_hi $a35.hi $a36_lo $a36.lo $a36_hi $a36.hi $a37_lo $a37.lo

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
xwordqRegLoE	XBR	$a0_lo $a0.lo $a1_lo $a1.lo $a2_lo $a2.lo $a3_lo $a3.lo $a4_lo $a4.lo $a5_lo $a5.lo $a6_lo $a6.lo $a7_lo $a7.lo $a8_lo $a8.lo $a9_lo $a9.lo $a10_lo $a10.lo $a11_lo $a11.lo $a12_lo $a12.lo $a13_lo $a13.lo $a14_lo $a14.lo $a15_lo $a15.lo $a16_lo $a16.lo $a17_lo $a17.lo $a18_lo $a18.lo $a19_lo $a19.lo $a20_lo $a20.lo $a21_lo $a21.lo $a22_lo $a22.lo $a23_lo $a23.lo $a24_lo $a24.lo $a25_lo $a25.lo $a26_lo $a26.lo $a27_lo $a27.lo $a28_lo $a28.lo $a29_lo $a29.lo $a30_lo $a30.lo $a31_lo $a31.lo $a32_lo $a32.lo $a33_lo $a33.lo $a34_lo $a34.lo $a35_lo $a35.lo $a36_lo $a36.lo $a37_lo $a37.lo $a38_lo $a38.lo $a39_lo $a39.lo $a40_lo $a40.lo $a41_lo $a41.lo $a42_lo $a42.lo $a43_lo $a43.lo $a44_lo $a44.lo $a45_lo $a45.lo $a46_lo $a46.lo $a47_lo $a47.lo $a48_lo $a48.lo $a49_lo $a49.lo $a50_lo $a50.lo $a51_lo $a51.lo $a52_lo $a52.lo $a53_lo $a53.lo $a54_lo $a54.lo $a55_lo $a55.lo $a56_lo $a56.lo $a57_lo $a57.lo $a58_lo $a58.lo $a59_lo $a59.lo $a60_lo $a60.lo $a61_lo $a61.lo $a62_lo $a62.lo $a63_lo $a63.lo

Kalray – all rights reserved

Register Class	Register File	Registers
xwordqRegLoO	XBR	$a0_hi $a0.hi
		$a1_hi $a1.hi
		$a2_hi $a2.hi
		$a3_hi $a3.hi
		$a4_hi $a4.hi
		$a5_hi $a5.hi
		$a6_hi $a6.hi
		$a7_hi $a7.hi
		$a8_hi $a8.hi
		$a9_hi $a9.hi
		$a10_hi $a10.hi
		$a11_hi $a11.hi
		$a12_hi $a12.hi
		$a13_hi $a13.hi
		$a14_hi $a14.hi
		$a15_hi $a15.hi
		$a16_hi $a16.hi
		$a17_hi $a17.hi
		$a18_hi $a18.hi
		$a19_hi $a19.hi
		$a20_hi $a20.hi
		$a21_hi $a21.hi
		$a22_hi $a22.hi
		$a23_hi $a23.hi
		$a24_hi $a24.hi
		$a25_hi $a25.hi
		$a26_hi $a26.hi
		$a27_hi $a27.hi
		$a28_hi $a28.hi
		$a29_hi $a29.hi
		$a30_hi $a30.hi
		$a31_hi $a31.hi
		$a32_hi $a32.hi
		$a33_hi $a33.hi
		$a34_hi $a34.hi
		$a35_hi $a35.hi
		$a36_hi $a36.hi
		$a37_hi $a37.hi
		$a38_hi $a38.hi
		$a39_hi $a39.hi
		$a40_hi $a40.hi
		$a41_hi $a41.hi
		$a42_hi $a42.hi
		$a43_hi $a43.hi
		$a44_hi $a44.hi
		$a45_hi $a45.hi
		$a46_hi $a46.hi
		$a47_hi $a47.hi
		$a48_hi $a48.hi
		$a49_hi $a49.hi
		$a50_hi $a50.hi
		$a51_hi $a51.hi
		$a52_hi $a52.hi
		$a53_hi $a53.hi
		$a54_hi $a54.hi
		$a55_hi $a55.hi
		$a56_hi $a56.hi
		$a57_hi $a57.hi
		$a58_hi $a58.hi
		$a59_hi $a59.hi
		$a60_hi $a60.hi
		$a61_hi $a61.hi
		$a62_hi $a62.hi
		$a63_hi $a63.hi

Kalray – all rights reserved

©2025 Kalray

Register Class	Register File	Registers
xwordvRegLo	XMR	$a0a1a2a3 $a4a5a6a7 $a8a9a10a11 $a12a13a14a15 $a16a17a18a19 $a20a21a22a23 $a24a25a26a27 $a28a29a30a31 $a32a33a34a35 $a36a37a38a39 $a40a41a42a43 $a44a45a46a47 $a48a49a50a51 $a52a53a54a55 $a56a57a58a59 $a60a61a62a63

Kalray – all rights reserved

Register Class	Register File	Registers
xwordxRegLo	XTR	$a0a1 $a0a1a2a3.lo $a2a3 $a0a1a2a3.hi $a4a5 $a4a5a6a7.lo $a6a7 $a4a5a6a7.hi $a8a9 $a8a9a10a11.lo $a10a11 $a8a9a10a11.hi $a12a13 $a12a13a14a15.lo $a14a15 $a12a13a14a15.hi $a16a17 $a16a17a18a19.lo $a18a19 $a16a17a18a19.hi $a20a21 $a20a21a22a23.lo $a22a23 $a20a21a22a23.hi $a24a25 $a24a25a26a27.lo $a26a27 $a24a25a26a27.hi $a28a29 $a28a29a30a31.lo $a30a31 $a28a29a30a31.hi $a32a33 $a32a33a34a35.lo $a34a35 $a32a33a34a35.hi $a36a37 $a36a37a38a39.lo $a38a39 $a36a37a38a39.hi $a40a41 $a40a41a42a43.lo $a42a43 $a40a41a42a43.hi $a44a45 $a44a45a46a47.lo $a46a47 $a44a45a46a47.hi $a48a49 $a48a49a50a51.lo $a50a51 $a48a49a50a51.hi $a52a53 $a52a53a54a55.lo $a54a55 $a52a53a54a55.hi $a56a57 $a56a57a58a59.lo $a58a59 $a56a57a58a59.hi $a60a61 $a60a61a62a63.lo $a62a63 $a60a61a62a63.hi

Kalray – all rights reserved

©2025 Kalray

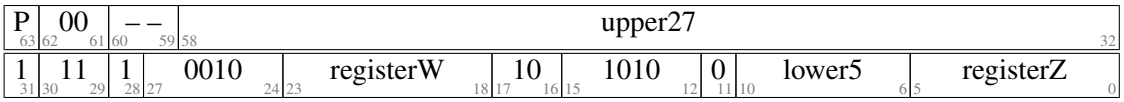
6.11 Synthetic Instructions

Synthetic instructions are special cases of regular instructions, where one or more operands are forced to constant values.

Kalray – all rights reserved

6.11.1 ABSBO — Absolute Value of Byte Octuple

Encoding Format ALU_BOWRR0.M, exucode2 = 0010



Specializes Instruction ABDBO, format ALU_BOWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `absbo` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

©2025 Kalray

6.11.2 ABSD — Absolute Value of Double Word

Encoding Format ALU_DWRI, exucode2 = 0001

P 31	11 30 29	0 28	0001 27 24	registerW 23 18	00 17 16	0000000000 15 6	registerZ 5 0

Specializes Instruction ABDD, format ALU_DWRI

Forced operands *signed10* = 0

Syntax `absd` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

6.11.3 ABSHQ Absolute Value of Half Word Quadruple

Encoding Format ALU_HQWRR0.M, exucode2 = 0001

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0001	registerW	01	0011	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction ABDHQ, format ALU_HQWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `abshq` *registerW* = *registerZ*, *00*

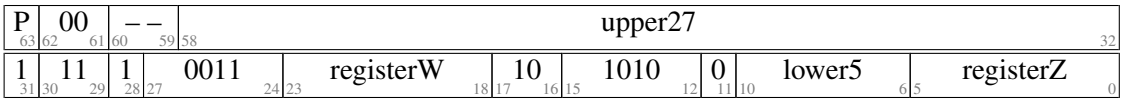
Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.4 ABSSBO Absolute Value Saturated of Byte Octuple

Encoding Format ALU_BOWRR0.M, exucode2 = 0011



Specializes Instruction ABDSBO, format ALU_BOWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `abssbo` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.5 ABSSD Absolute Value Saturated of Double Word

Encoding Format ALU_DWRR0.M, exucode2 = 0000

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0000	registerW	01	0000	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction ABDSD, format ALU_DWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `abssd` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.6 ABSSHQ Absolute Value Saturated of Half Quadruple

Encoding Format ALU_HQWRR0.M, exucode2 = 0000

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	11	1	0000				registerW						01		0011			0	lower5					registerZ									
31	30	29	28	27		24	23					18	17	16	15		12	11	10				6	5								0

Specializes Instruction ABDSHQ, format ALU_HQWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `absshq` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.7 ABSSW — Absolute Value Saturated of Word

Encoding Format ALU_WWRR0.W, exucode2 = 0000

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0000	registerW	01	0001	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction ABDSW, format ALU_WWRR0.W

Forced operands *upper27_lower5* = 0

Syntax `abssw` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

6.11.8 ABSSWP Absolute Value Saturated of Word Pair

Encoding Format ALU_WPWRR0.M, exucode2 = 0000

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	11	1	0000				registerW						01		0010			0	lower5					registerZ									
31	30	29	28	27		24	23					18	17	16	15		12	11	10			6	5										0

Specializes Instruction ABDSWP, format ALU_WPWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `absswp` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.9 ABSW — Absolute Value of Word

Encoding Format ALU_WRIW, exucode2 = 0001

P	11	1	0001	registerW	00	0000000000	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction ABDW, format ALU_WRIW

Forced operands *signed10* = 0

Syntax `absw` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

6.11.10 ABSWP — Absolute Value of Word Pair

Encoding Format ALU_WPWRR0.M, exucode2 = 0001

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0001	registerW	01	0010	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction ABDWP, format ALU_WPWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `abswp` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.11 COPYD — Copy Double Word

Encoding Format ALU_DWRI, exucode2 = 1010

P	11	0	1010	registerW	00	0000000000	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORD, format ALU_DWRI

Forced operands *signed10* = 0

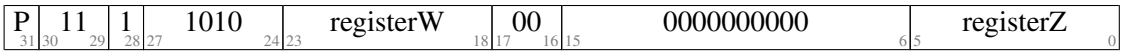
Syntax `copyd` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

©2025 Kalray

6.11.12 COPYW — Copy Word

Encoding Format ALU_WRIW, exucode2 = 1010



Specializes Instruction IORW, format ALU_WRIW

Forced operands *signed10* = 0

Syntax `copyw` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

6.11.13 LNORD

Alias of LNIORD

Encoding Format ALU_DSWRR, exucode2 = 0011

P	11	1	0011	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction LNIORD, format ALU_DSWRR

Forced operands

Syntax `lnord` *registerW* = *registerZ*, *registerY*

Kalray – all rights reserved

©2025 Kalray

6.11.14 LNORW

Alias of LNIORW

Encoding Format ALU_WSWRR, exucode2 = 0011

P	11	1	0011	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction LNIORW, format ALU_WSWRR

Forced operands

Syntax `lnorw` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_WSWRR.W, exucode2 = 0011

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0011	registerW	10	0111	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction LNIORW, format ALU_WSWRR.W

Forced operands

Syntax `lnorw` *registerW* = *registerZ*, *upper27_lower5*

Kalray – all rights reserved

6.11.15 LORD

Alias of LIORD

Encoding Format ALU_DSWRR, exucode2 = 0010

P	11	1	0010	registerW	10	0110	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction LIORD, format ALU_DSWRR

Forced operands

Syntax `lord` *registerW* = *registerZ*, *registerY*

Kalray – all rights reserved

©2025 Kalray

6.11.16 LORW

Alias of LIORW

Encoding Format ALU_WSWRR, exucode2 = 0010

P	11	1	0010	registerW	10	0111	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction LIORW, format ALU_WSWRR

Forced operands

Syntax `lorw` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_WSWRR.W, exucode2 = 0010

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	0010	registerW	10	0111	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction LIORW, format ALU_WSWRR.W

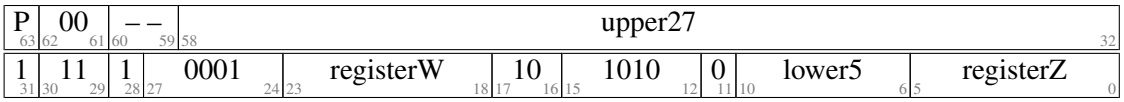
Forced operands

Syntax `lorw` *registerW* = *registerZ*, *upper27_lower5*

Kalray – all rights reserved

6.11.17 NEGBO — Negate Word Byte Octuple

Encoding Format ALU_BOWRR0.M, exucode2 = 0001



Specializes Instruction SBFBO, format ALU_BOWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `negbo` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

©2025 Kalray

6.11.18 NEGD — Negate Double Word

Encoding Format ALU_DWRI, exucode2 = 0011

P	11	0	0011	registerW	00	0000000000	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction SBFD, format ALU_DWRI

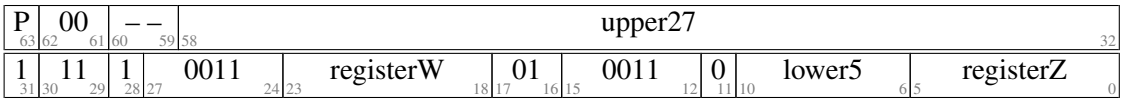
Forced operands *signed10* = 0

Syntax `negd` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

6.11.19 NEGHQ — Negate Word Half Word Quadruple

Encoding Format ALU_HQWRR0.M, exucode2 = 0011



Specializes Instruction SBFHQ, format ALU_HQWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `neghq` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.20 NEGSBO Negate Word Saturated Byte Octuple

Encoding Format ALU_BOWRR1.M, exucode2 = 1101

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1101	registerW	10	1011	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction SBFSBO, format ALU_BOWRR1.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `negsbo` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.21 NEGSD — Negate Saturated Double Word

Encoding Format ALU_DWRR1.M, exucode2 = 1101

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1101	registerW	01	0100	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction SBFSD, format ALU_DWRR1.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `negsd` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.22 NEGSHQ Negate Word Half Word Quadruple

Encoding Format ALU_HQWRR1.M, exucode2 = 1101

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1101	registerW	01	0111	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction SBFSHQ, format ALU_HQWRR1.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `negshq` *registerW* = *registerZ*, *00*

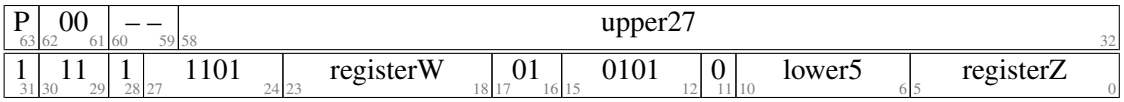
Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.23 NEGSW — Negate Saturated Word

Encoding Format ALU_WWRR1.W, exucode2 = 1101



Specializes Instruction SBFSW, format ALU_WWRR1.W

Forced operands *upper27_lower5* = 0

Syntax `negsw` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

6.11.24 NEGSWP Negate Saturated Word Pair

Encoding Format ALU_WPWRR1.M, exucode2 = 1101

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1101	registerW	01	0110	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction SBFSWP, format ALU_WPWRR1.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `negswp` *registerW* = *registerZ*, *00*

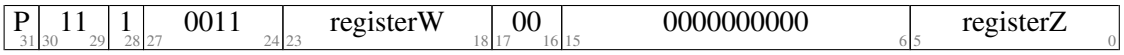
Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.25 NEGW — Negate Word

Encoding Format ALU_WRIW, exucode2 = 0011



Specializes Instruction SBFW, format ALU_WRIW

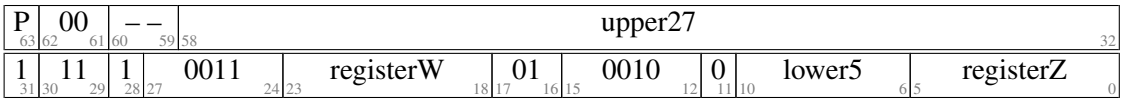
Forced operands *signed10* = 0

Syntax `negw` *registerW* = *registerZ*, *0*

Kalray – all rights reserved

6.11.26 NEGWP — Negate Word Pair

Encoding Format ALU_WPWRR0.M, exucode2 = 0011



Specializes Instruction SBFWP, format ALU_WPWRR0.M

Forced operands *upper27_lower5* = 0, *splat32* = 0

Syntax `negwp` *registerW* = *registerZ*, *00*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Kalray – all rights reserved

6.11.27 NORD

Legacy form of NIORD

Encoding Format ALU_DWRR0, exucode2 = 1011

P	11	1	1011	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction NIORD, format ALU_DWRR0

Forced operands

Syntax `nord` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_DWRR0.M, exucode2 = 1011

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1011	registerW	01	0000	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction NIORD, format ALU_DWRR0.M

Forced operands

Syntax `nord` *registerW* = *registerZ*, *upper27_lower5splat32*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Encoding Format ALU_DWRI, exucode2 = 1011

P	11	0	1011	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction NIORD, format ALU_DWRI

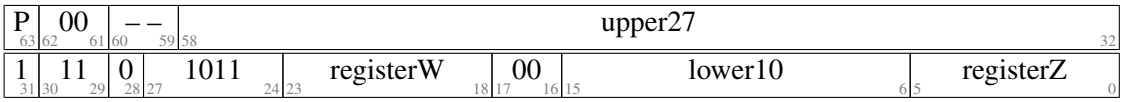
Forced operands

Kalray – all rights reserved

©2025 Kalray

Syntax `nord` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_DWRI.X, exucode2 = 1011

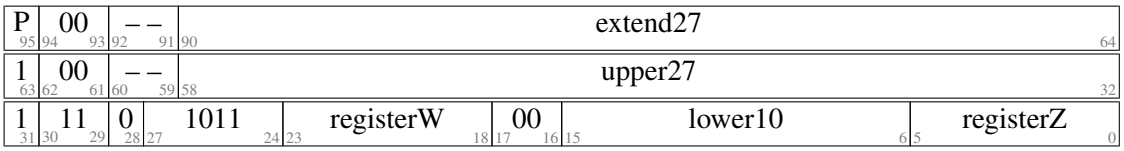


Specializes Instruction NIORD, format ALU_DWRI.X

Forced operands

Syntax `nord` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_DWRI.Y, exucode2 = 1011



Specializes Instruction NIORD, format ALU_DWRI.Y

Forced operands

Syntax `nord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Kalray – all rights reserved

6.11.28 NORW

Legacy form of NIORW

Encoding Format ALU_WRIW, exucode2 = 1011

P	11	1	1011	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction NIORW, format ALU_WRIW

Forced operands

Syntax `norw` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_WRIW.X, exucode2 = 1011

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1011	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction NIORW, format ALU_WRIW.X

Forced operands

Syntax `norw` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_WWRR0, exucode2 = 1011

P	11	1	1011	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction NIORW, format ALU_WWRR0

Forced operands

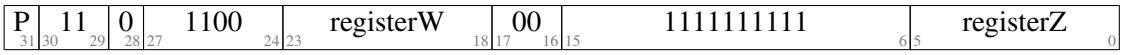
Syntax `norw` *registerW* = *registerZ*, *registerY*

Kalray – all rights reserved

©2025 Kalray

6.11.29 NOTD — Complement Double Word

Encoding Format ALU_DWRI, exucode2 = 1100



Specializes Instruction EORD, format ALU_DWRI

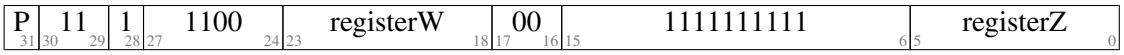
Forced operands *signed10* = -1

Syntax `notd` *registerW* = *registerZ*, *-1*

Kalray – all rights reserved

6.11.30 NOTW — Complement Word

Encoding Format ALU_WRIW, exucode2 = 1100



Specializes Instruction EORW, format ALU_WRIW

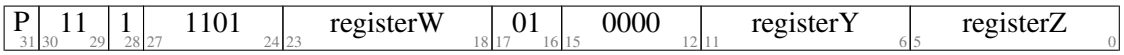
Forced operands *signed10* = -1

Syntax `notw` *registerW* = *registerZ*, *-1*

Kalray – all rights reserved

6.11.31 NXORD — Alias of NEORD

Encoding Format ALU_DWRR0, exucode2 = 1101

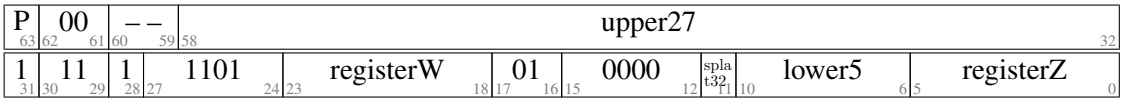


Specializes Instruction NEORD, format ALU_DWRR0

Forced operands

Syntax `nxord` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_DWRR0.M, exucode2 = 1101



Specializes Instruction NEORD, format ALU_DWRR0.M

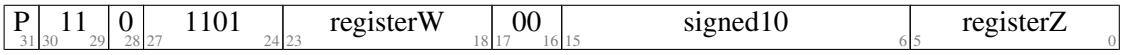
Forced operands

Syntax `nxord` *registerW* = *registerZ*, *upper27_lower5splat32*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Encoding Format ALU_DWRI, exucode2 = 1101



Specializes Instruction NEORD, format ALU_DWRI

Forced operands

Kalray – all rights reserved

Syntax `nxord` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_DWRI.X, exucode2 = 1101

P	00	– –	upper27				
63	62 61	60 59	58 32				
1	11	0	1101	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction NEORD, format ALU_DWRI.X

Forced operands

Syntax `nxord` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_DWRI.Y, exucode2 = 1101

P	00	– –	extend27				
95	94 93	92 91	90 64				
1	00	– –	upper27				
63	62 61	60 59	58 32				
1	11	0	1101	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction NEORD, format ALU_DWRI.Y

Forced operands

Syntax `nxord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Kalray – all rights reserved

©2025 Kalray

6.11.32 NXORW

Alias of NEORW

Encoding Format ALU_WRIW, exucode2 = 1101

P	11	1	1101	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction NEORW, format ALU_WRIW

Forced operands

Syntax `nxorw` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_WRIW.X, exucode2 = 1101

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1101	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction NEORW, format ALU_WRIW.X

Forced operands

Syntax `nxorw` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_WWRR0, exucode2 = 1101

P	11	1	1101	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction NEORW, format ALU_WWRR0

Forced operands

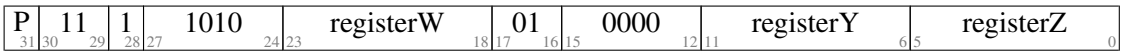
Syntax `nxorw` *registerW* = *registerZ*, *registerY*

Kalray – all rights reserved

6.11.33 ORD

Legacy form of IORD

Encoding Format ALU_DWRR0, exucode2 = 1010

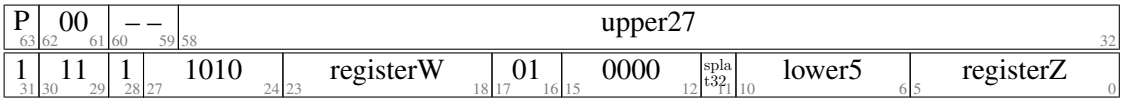


Specializes Instruction IORD, format ALU_DWRR0

Forced operands

Syntax `ord` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_DWRR0.M, exucode2 = 1010



Specializes Instruction IORD, format ALU_DWRR0.M

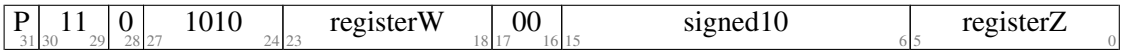
Forced operands

Syntax `ord` *registerW* = *registerZ*, *upper27_lower5splat32*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Encoding Format ALU_DWRI, exucode2 = 1010



Specializes Instruction IORD, format ALU_DWRI

Forced operands

Kalray – all rights reserved

©2025 Kalray

Syntax `ord` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_DWRI.X, exucode2 = 1010

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	1010	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORD, format ALU_DWRI.X

Forced operands

Syntax `ord` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_DWRI.Y, exucode2 = 1010

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	1010	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORD, format ALU_DWRI.Y

Forced operands

Syntax `ord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Kalray – all rights reserved

6.11.34 ORND — Legacy form of IORND

Encoding Format ALU_DWRR0, exucode2 = 1111

P	11	1	1111	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction IORND, format ALU_DWRR0

Forced operands

Syntax `ornd` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_DWRR0.M, exucode2 = 1111

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1111	registerW	01	0000	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction IORND, format ALU_DWRR0.M

Forced operands

Syntax `ornd` *registerW* = *registerZ*, *upper27_lower5splat32*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Encoding Format ALU_DWRI, exucode2 = 1111

P	11	0	1111	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORND, format ALU_DWRI

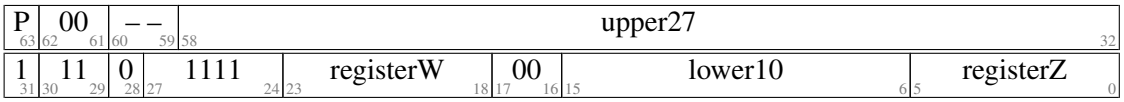
Forced operands

Kalray – all rights reserved

©2025 Kalray

Syntax `ornd` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_DWRI.X, exucode2 = 1111

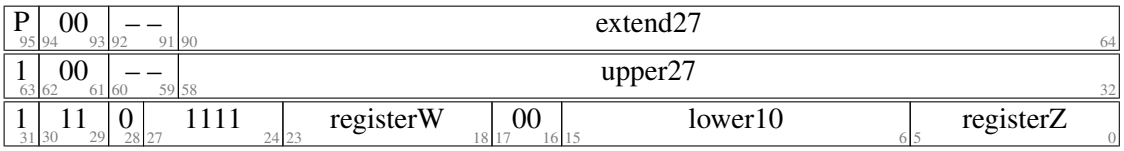


Specializes Instruction IORND, format ALU_DWRI.X

Forced operands

Syntax `ornd` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_DWRI.Y, exucode2 = 1111



Specializes Instruction IORND, format ALU_DWRI.Y

Forced operands

Syntax `ornd` *registerW* = *registerZ*, *extend27_upper27_lower10*

Kalray – all rights reserved

6.11.35 ORNW

Legacy form of IORNW

Encoding Format ALU_WRIW, exucode2 = 1111

P	11	1	1111	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORNW, format ALU_WRIW

Forced operands

Syntax `ornw` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_WRIW.X, exucode2 = 1111

P	00	– –	upper27				
63	62 61	60 59	58 32				
1	11	1	1111	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORNW, format ALU_WRIW.X

Forced operands

Syntax `ornw` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_WWRR0, exucode2 = 1111

P	11	1	1111	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction IORNW, format ALU_WWRR0

Forced operands

Syntax `ornw` *registerW* = *registerZ*, *registerY*

Kalray – all rights reserved

©2025 Kalray

6.11.36 ORRBOD — Legacy form of IORRBOD

Encoding Format ALU_BORWRR, exucode2 = 0001

P	11	1	0001	registerW	11	1100	–	– – –	10	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Specializes Instruction IORRBOD, format ALU_BORWRR

Forced operands

Syntax `orrbod` *registerW* = *registerZ*

Kalray – all rights reserved

6.11.37 ORRHQD — Legacy form of IORRHQD

Encoding Format ALU_HQRWRR, exucode2 = 0001

P	11	1	0001	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Specializes Instruction IORRHQD, format ALU_HQRWRR

Forced operands

Syntax `orrhqd` *registerW* = *registerZ*

Kalray – all rights reserved

©2025 Kalray

6.11.38 ORRWPD — Legacy form of IORRWPD

Encoding Format ALU_WPRWRR, exucode2 = 0001

P	11	1	0001	registerW	11	1100	–	– – –	00	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Specializes Instruction IORRWPD, format ALU_WPRWRR

Forced operands

Syntax `orrwpd` *registerW* = *registerZ*

Kalray – all rights reserved

6.11.39 ORW

Legacy form of IORW

Encoding Format ALU_WRIW, exucode2 = 1010

P	11	1	1010	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORW, format ALU_WRIW

Forced operands

Syntax `orw` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_WRIW.X, exucode2 = 1010

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1010	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction IORW, format ALU_WRIW.X

Forced operands

Syntax `orw` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_WWRR0, exucode2 = 1010

P	11	1	1010	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction IORW, format ALU_WWRR0

Forced operands

Syntax `orw` *registerW* = *registerZ*, *registerY*

Kalray – all rights reserved

©2025 Kalray

6.11.40 SBMM8 Swapped Bit Matrix Multiplication 8×8

Encoding Format ALU_BWRR, exucode2 = 1110

P	11	1	1110	registerW	10	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction SBMM8D, format ALU_BWRR

Forced operands

Syntax `sbmm8` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_BWRR.M, exucode2 = 1110

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1110	registerW	10	0000	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction SBMM8D, format ALU_BWRR.M

Forced operands

Syntax `sbmm8` *registerW* = *registerZ*, *upper27_lower5splat32*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Encoding Format ALU_DBMWRI, exucode2 = 1110

P	11	0	1110	registerW	10	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

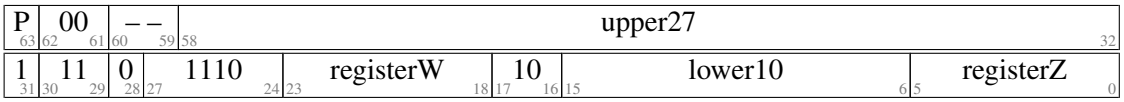
Specializes Instruction SBMM8D, format ALU_DBMWRI

Forced operands

Kalray – all rights reserved

Syntax `sbmm8` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_DBMWRI.X, exucode2 = 1110

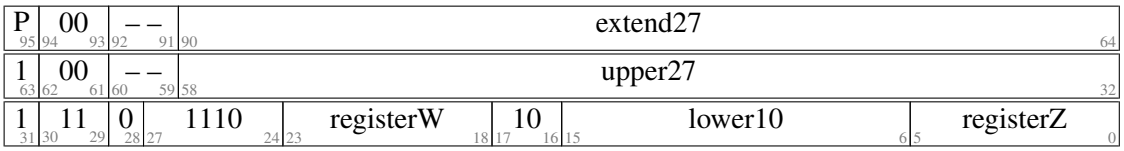


Specializes Instruction SBMM8D, format ALU_DBMWRI.X

Forced operands

Syntax `sbmm8` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_DBMWRI.Y, exucode2 = 1110



Specializes Instruction SBMM8D, format ALU_DBMWRI.Y

Forced operands

Syntax `sbmm8` *registerW* = *registerZ*, *extend27_upper27_lower10*

Kalray – all rights reserved

©2025 Kalray

6.11.41 SBMMT8 Swapped Bit Matrix Multiplication Transposed 8×8

Encoding Format ALU_BWRR, exucode2 = 1111

P	11	1	1111	registerW	10	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction SBMMT8D, format ALU_BWRR

Forced operands

Syntax `sbmmt8` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_BWRR.M, exucode2 = 1111

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1111	registerW	10	0000	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction SBMMT8D, format ALU_BWRR.M

Forced operands

Syntax `sbmmt8` *registerW* = *registerZ*, *upper27_lower5splat32*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Encoding Format ALU_DBMWRI, exucode2 = 1111

P	11	0	1111	registerW	10	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction SBMMT8D, format ALU_DBMWRI

Forced operands

Kalray – all rights reserved

Syntax `sbmmt8` *registerW = registerZ, signed10*

Encoding Format ALU_DBMWRI.X, exucode2 = 1111

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	1111	registerW	10	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction SBMMT8D, format ALU_DBMWRI.X

Forced operands

Syntax `sbmmt8` *registerW = registerZ, upper27_lower10*

Encoding Format ALU_DBMWRI.Y, exucode2 = 1111

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	1111	registerW	10	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction SBMMT8D, format ALU_DBMWRI.Y

Forced operands

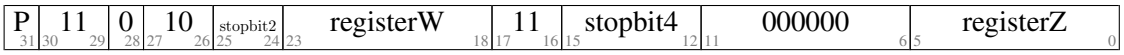
Syntax `sbmmt8` *registerW = registerZ, extend27_upper27_lower10*

Kalray – all rights reserved

©2025 Kalray

6.11.42 SXBD — Sign Extend Byte to Double Word

Encoding Format ALU_WRB, wricode1 = 10



Specializes Instruction EXTFS, format ALU_WRB

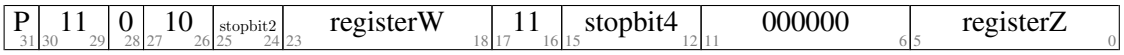
Forced operands *stopbit2_stopbit4* = 7, *startbit* = 0

Syntax `sxbd` *registerW* = *registerZ*, *7*, *0*

Kalray – all rights reserved

6.11.43 SXHD — Sign Extend Half Word to Double Word

Encoding Format ALU_WRB, wricode1 = 10



Specializes Instruction EXTFS, format ALU_WRB

Forced operands *stopbit2_stopbit4* = 15, *startbit* = 0

Syntax `sxhd` *registerW* = *registerZ*, *15*, *0*

Kalray – all rights reserved

©2025 Kalray

6.11.44 SXWD — Sign Extend Word to Double Word

Encoding Format ALU_WRB, wricode1 = 10

P	11	0	10	stopbit2	registerW	11	stopbit4	000000	registerZ
31	30 29	28	27 26	25 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction EXTFS, format ALU_WRB

Forced operands *stopbit2_stopbit4* = 31, *startbit* = 0

Syntax `sxwd` *registerW* = *registerZ*, *31*, *0*

Kalray – all rights reserved

6.11.45 XORD — Alias of EORD

Encoding Format ALU_DWRR0, exucode2 = 1100

P	11	1	1100	registerW	01	0000	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction EORD, format ALU_DWRR0

Forced operands

Syntax `xord` *registerW* = *registerZ*, *registerY*

Encoding Format ALU_DWRR0.M, exucode2 = 1100

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1100	registerW	01	0000	splat32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

Specializes Instruction EORD, format ALU_DWRR0.M

Forced operands

Syntax `xord` *registerW* = *registerZ*, *upper27_lower5splat32*

Modifier splat32:

Name	Case	Value
.	No splat	0
.@	Splat x2	1

Encoding Format ALU_DWRI, exucode2 = 1100

P	11	0	1100	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction EORD, format ALU_DWRI

Forced operands

Kalray – all rights reserved

©2025 Kalray

Syntax `xord` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_DWRI.X, exucode2 = 1100

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	1100	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction EORD, format ALU_DWRI.X

Forced operands

Syntax `xord` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_DWRI.Y, exucode2 = 1100

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	1100	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction EORD, format ALU_DWRI.Y

Forced operands

Syntax `xord` *registerW* = *registerZ*, *extend27_upper27_lower10*

Kalray – all rights reserved

6.11.46 XORRBOD

Alias of EORRBOD

Encoding Format ALU_BORWRR, exucode2 = 0010

P	11	1	0010	registerW	11	1100	–	– – –	10	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Specializes Instruction EORRBOD, format ALU_BORWRR

Forced operands

Syntax `xorrbod` *registerW* = *registerZ*

Kalray – all rights reserved

©2025 Kalray

6.11.47 XORRHQD

Alias of EORRHQD

Encoding Format ALU_HQRWRR, exucode2 = 0010

P	11	1	0010	registerW	11	1100	–	– – –	01	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Specializes Instruction EORRHQD, format ALU_HQRWRR

Forced operands

Syntax `xorrhqd` *registerW* = *registerZ*

Kalray – all rights reserved

6.11.48 XORRWPD

Alias of EORRWPD

Encoding Format ALU_WPRWRR, exucode2 = 0010

P	11	1	0010	registerW	11	1100	–	– – –	00	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 8	7 6	5 0

Specializes Instruction EORRWPD, format ALU_WPRWRR

Forced operands

Syntax `xorrwpd` *registerW* = *registerZ*

Kalray – all rights reserved

©2025 Kalray

6.11.49 XORW

Alias of EORW

Encoding Format ALU_WRIW, exucode2 = 1100

P	11	1	1100	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction EORW, format ALU_WRIW

Forced operands

Syntax `xorw` *registerW* = *registerZ*, *signed10*

Encoding Format ALU_WRIW.X, exucode2 = 1100

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	1100	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction EORW, format ALU_WRIW.X

Forced operands

Syntax `xorw` *registerW* = *registerZ*, *upper27_lower10*

Encoding Format ALU_WWRR0, exucode2 = 1100

P	11	1	1100	registerW	01	0001	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction EORW, format ALU_WWRR0

Forced operands

Syntax `xorw` *registerW* = *registerZ*, *registerY*

Kalray – all rights reserved

6.11.50 ZXBD — Zero Extend Byte to Double Word

Encoding Format ALU_WRIW, exucode2 = 1000

P	11	1	1000	registerW	00	0011111111	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

Specializes Instruction ANDW, format ALU_WRIW

Forced operands *signed10* = 255

Syntax `zxbd` *registerW* = *registerZ*, *255*

Kalray – all rights reserved

©2025 Kalray

6.11.51 ZXHD — Zero Extend Half Word to Double Word

Encoding Format ALU_WRB, wricode1 = 01

P	11	0	01	stopbit2	registerW	11	stopbit4	000000	registerZ
31	30 29	28	27 26	25 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction EXTFZ, format ALU_WRB

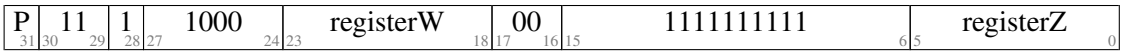
Forced operands *stopbit2_stopbit4* = 15, *startbit* = 0

Syntax `zxhd` *registerW* = *registerZ*, *15*, *0*

Kalray – all rights reserved

6.11.52 ZXWD — Zero Extend Word to Double Word

Encoding Format ALU_WRIW, exucode2 = 1000



Specializes Instruction ANDW, format ALU_WRIW

Forced operands *signed10* = -1

Syntax `zxwd` *registerW* = *registerZ*, *-1*

Kalray – all rights reserved

6.11.53 XMT44D Extension Matrix Transpose 4x4 Double Words

Encoding Format EXT_XCOPYV, exucode4 = 01

P	00	00	111	registerAq	01	01	– – –	1	registerCq	00	– – – – – –
31	30 29	28 27	26 24	23 20	19 18	17 16	15 13	12	11 8	7 6	5 0

Specializes Instruction XCOPYV, format EXT_XCOPYV

Forced operands *shuffleV* = 1

Syntax `xmt44d`*1 registerAq* = *registerCq*

Modifier shuffleV:

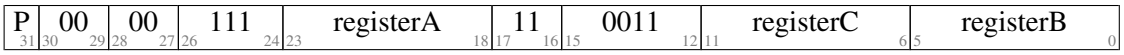
Name	Case	Value
.	No Shuffle	0
.TD	Transpose Double Words 4x4	1

Kalray – all rights reserved

6.11.54 XNORO

Alias of XNIORO

Encoding Format EXT_XBWO, exucode5 = 0011



Specializes Instruction XNIORO, format EXT_XBWO

Forced operands

Syntax `xnoro` *registerA* = *registerB*, *registerC*

Kalray – all rights reserved

©2025 Kalray

6.11.55 XNXORO

Alias of XNEORO

Encoding Format EXT_XBWO, exucode5 = 0101

P	00	00	111	registerA	11	0101	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction XNEORO, format EXT_XBWO

Forced operands

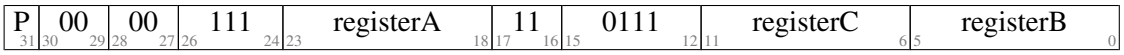
Syntax `xnxoro` *registerA* = *registerB*, *registerC*

Kalray – all rights reserved

6.11.56 XORNO

Alias of XIORNO

Encoding Format EXT_XBWO, exucode5 = 0111



Specializes Instruction XIORNO, format EXT_XBWO

Forced operands

Syntax `xorno` *registerA* = *registerB*, *registerC*

Kalray – all rights reserved

©2025 Kalray

6.11.57 XORO

Legacy form of XIORO

Encoding Format EXT_XBWO, exucode5 = 0010

P	00	00	111	registerA	11	0010	registerC	registerB
31	30 29	28 27	26 24	23 18	17 16	15 12	11 6	5 0

Specializes Instruction XIORO, format EXT_XBWO

Forced operands

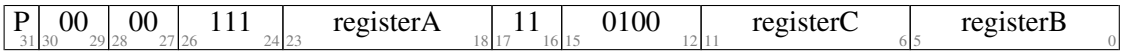
Syntax `xoro` *registerA* = *registerB*, *registerC*

Kalray – all rights reserved

6.11.58 XXORO

Alias of XEORO

Encoding Format EXT_XBWO, exucode5 = 0100



Specializes Instruction XEORO, format EXT_XBWO

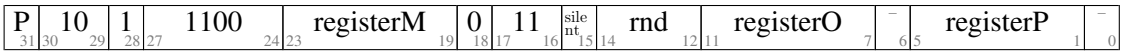
Forced operands

Syntax `xxoro` *registerA* = *registerB*, *registerC*

Kalray – all rights reserved

6.11.59 FADDDC Floating-Point Add From Double Word Complex

Encoding Format MAU_FQQQ0, exucode2 = 1100



Specializes Instruction FADDDP, format MAU_FQQQ0

Forced operands

Syntax `fadddc`*roundingsilent registerM* = *registerP*, *registerO*

Modifier rounding:

Name	Case	Value
.RN	Round to Nearest, ties to Even	0
.RU	Round Upward	1
.RD	Round Downward	2
.RZ	Round toward Zero	3
.R4	Reserved 4	4
.R5	Reserved 5	5
.RO	Reserved 6	6
.	Use CS rounding	7

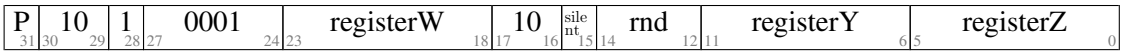
Modifier silent:

Name	Case	Value
.	Effects on CS	0
.S	Silent on CS	1

Kalray – all rights reserved

6.11.60 FADDWC Floating-Point Add From Word Complex

Encoding Format MAU_FDDD1, exucode2 = 0001



Specializes Instruction FADDWP, format MAU_FDDD1

Forced operands

Syntax `faddwc`*roundingsilent registerW = registerZ, registerY*

Modifier rounding:

Name	Case	Value
.RN	Round to Nearest, ties to Even	0
.RU	Round Upward	1
.RD	Round Downward	2
.RZ	Round toward Zero	3
.R4	Reserved 4	4
.R5	Reserved 5	5
.RO	Reserved 6	6
.	Use CS rounding	7

Modifier silent:

Name	Case	Value
.	Effects on CS	0
.S	Silent on CS	1

Kalray – all rights reserved

6.11.61 FADDWCP Floating-Point Add Word Complex Pair

Encoding Format MAU_FQQQ1, exucode2 = 1000

P	10	1	1000	registerM	1	11	silent	rnd	registerO	–	registerP	–
31	30 29	28	27 24	23 19	18	17 16	15	14 12	11 7	6	5 1	0

Specializes Instruction FADDWQ, format MAU_FQQQ1

Forced operands

Syntax `faddwcp`*roundingsilent registerM* = *registerP*, *registerO*

Modifier rounding:

Name	Case	Value
.RN	Round to Nearest, ties to Even	0
.RU	Round Upward	1
.RD	Round Downward	2
.RZ	Round toward Zero	3
.R4	Reserved 4	4
.R5	Reserved 5	5
.RO	Reserved 6	6
.	Use CS rounding	7

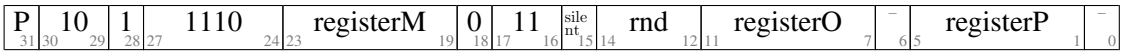
Modifier silent:

Name	Case	Value
.	Effects on CS	0
.S	Silent on CS	1

Kalray – all rights reserved

6.11.62 FSBFDC Floating-Point Subtract Double Word Complex

Encoding Format MAU_FQQQ0, exucode2 = 1110



Specializes Instruction FSBFDP, format MAU_FQQQ0

Forced operands

Syntax `fsbfdc`*roundingsilent registerM* = *registerP*, *registerO*

Modifier rounding:

Name	Case	Value
.RN	Round to Nearest, ties to Even	0
.RU	Round Upward	1
.RD	Round Downward	2
.RZ	Round toward Zero	3
.R4	Reserved 4	4
.R5	Reserved 5	5
.RO	Reserved 6	6
.	Use CS rounding	7

Modifier silent:

Name	Case	Value
.	Effects on CS	0
.S	Silent on CS	1

Kalray – all rights reserved

6.11.63 FSBFWC Floating-Point Subtract From Word Complex

Encoding Format MAU_FDDD1, exucode2 = 0101

P	10	1	0101	registerW	10	silent	rnd	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15	14 12	11 6	5 0

Specializes Instruction FSBFWP, format MAU_FDDD1

Forced operands

Syntax `fsbfwc`*roundingsilent registerW* = *registerZ*, *registerY*

Modifier rounding:

Name	Case	Value
.RN	Round to Nearest, ties to Even	0
.RU	Round Upward	1
.RD	Round Downward	2
.RZ	Round toward Zero	3
.R4	Reserved 4	4
.R5	Reserved 5	5
.RO	Reserved 6	6
.	Use CS rounding	7

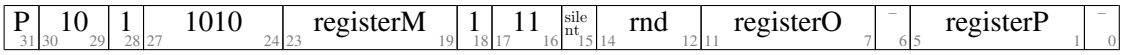
Modifier silent:

Name	Case	Value
.	Effects on CS	0
.S	Silent on CS	1

Kalray – all rights reserved

6.11.64 FSBFWCP Floating-Point Subtract From Word Complex Pair

Encoding Format MAU_FQQQ1, exucode2 = 1010



Specializes Instruction FSBFWQ, format MAU_FQQQ1

Forced operands

Syntax `fsbfwcp`*roundingsilent registerM* = *registerP*, *registerO*

Modifier rounding:

Name	Case	Value
.RN	Round to Nearest, ties to Even	0
.RU	Round Upward	1
.RD	Round Downward	2
.RZ	Round toward Zero	3
.R4	Reserved 4	4
.R5	Reserved 5	5
.RO	Reserved 6	6
.	Use CS rounding	7

Modifier silent:

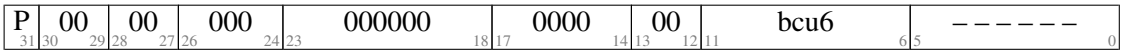
Name	Case	Value
.	Effects on CS	0
.S	Silent on CS	1

Kalray – all rights reserved

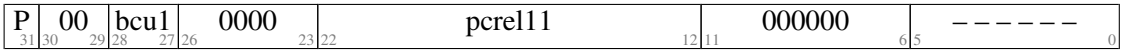
©2025 Kalray

6.12 Format Descriptions

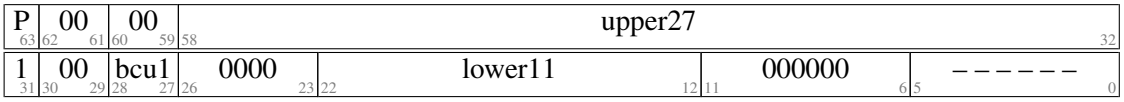
6.12.1 BCU Opcode Error



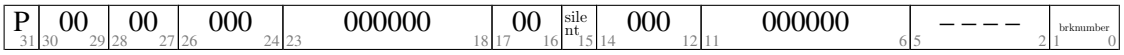
6.12.2 BCU PCREL11 Dummy



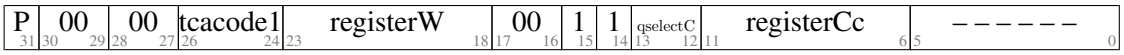
6.12.3 BCU PCREL Dummy



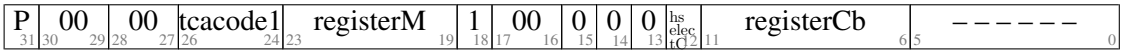
6.12.4 BCU Break



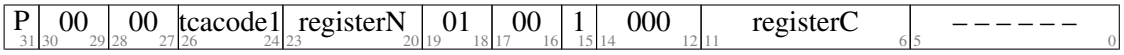
6.12.5 BCU Move From Coprocessor Double Word



6.12.6 BCU Move From Coprocessor Quadruple Word

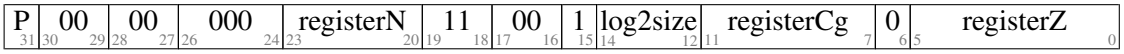


6.12.7 BCU Move From Coprocessor Octuple Word

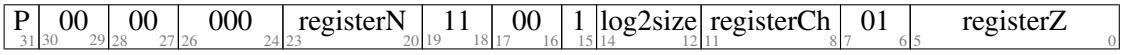


Kalray – all rights reserved

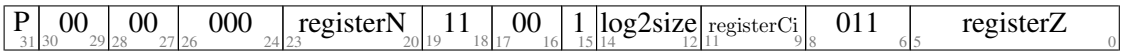
6.12.8 BCU XACCESSO from 2-Register Buffer



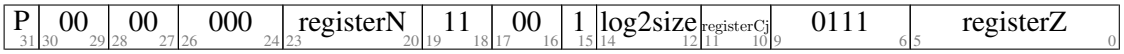
6.12.9 BCU XACCESSO from 4-Register Buffer



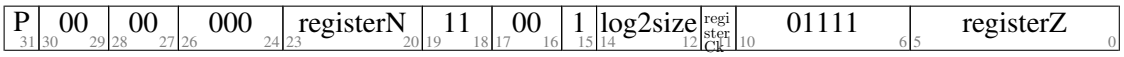
6.12.10 BCU XACCESSO from 8-Register Buffer



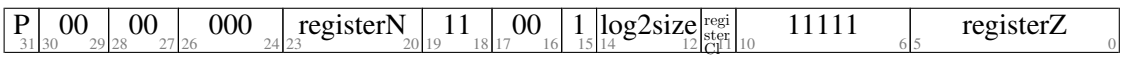
6.12.11 BCU XACCESSO from 16-Register Buffer



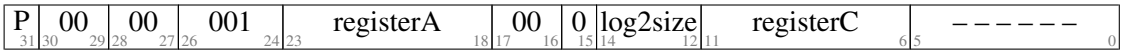
6.12.12 BCU XACCESSO from 32-Register Buffer



6.12.13 BCU XACCESSO from 64-Register Buffer



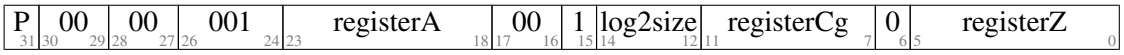
6.12.14 BCU XCOPYO



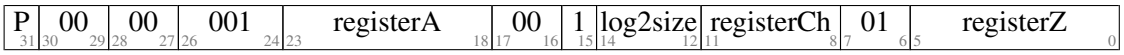
Kalray – all rights reserved

©2025 Kalray

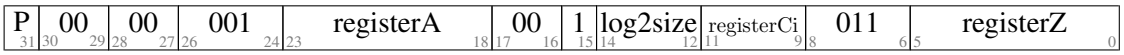
6.12.15 BCU XALIGNO from 2-Register Buffer



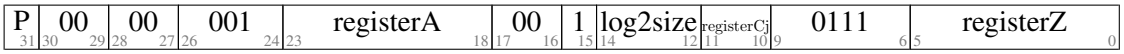
6.12.16 BCU XALIGNO from 4-Register Buffer



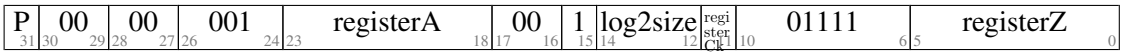
6.12.17 BCU XALIGNO from 8-Register Buffer



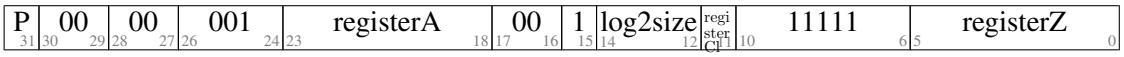
6.12.18 BCU XALIGNO from 16-Register Buffer



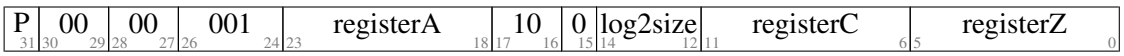
6.12.19 BCU XALIGNO from 32-Register Buffer



6.12.20 BCU XALIGNO from 64-Register Buffer

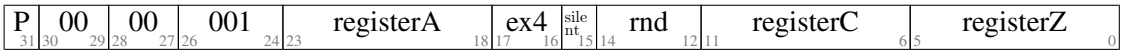


6.12.21 BCU Reshape Vector to Vector

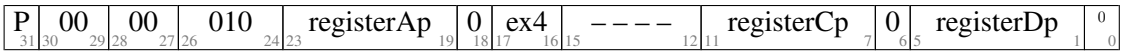


Kalray – all rights reserved

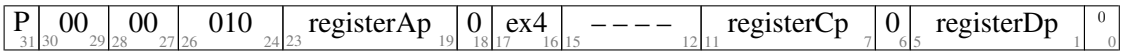
6.12.22 BCU Floating Point Scale Word Octuple



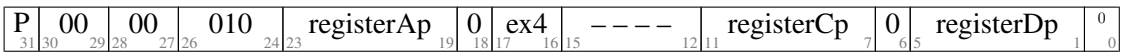
6.12.23 Extension Matrix Multiply Add Bytes to Words 0



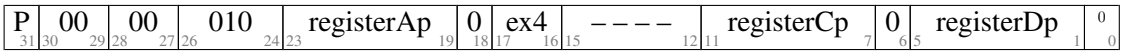
6.12.24 Extension Matrix Multiply Add Bytes to Words 1



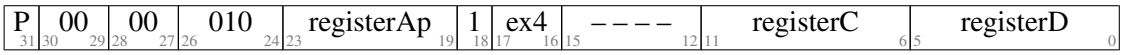
6.12.25 Extension Matrix Multiply Add Bytes to Words 2



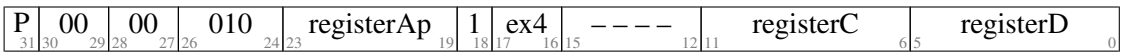
6.12.26 Extension Matrix Multiply Add Bytes to Words 3



6.12.27 Extension Matrix Multiply Add Bytes to Words 0



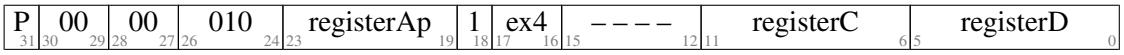
6.12.28 Extension Matrix Multiply Add Bytes to Words 1



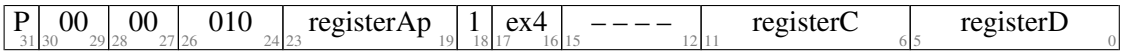
Kalray – all rights reserved

©2025 Kalray

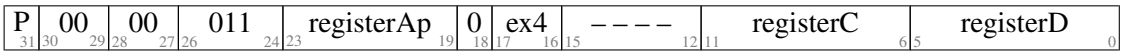
6.12.29 Extension Matrix Multiply Add Bytes to Words 2



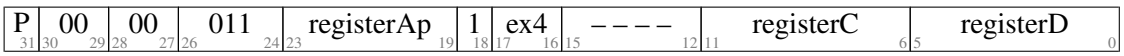
6.12.30 Extension Matrix Multiply Add Bytes to Words 3



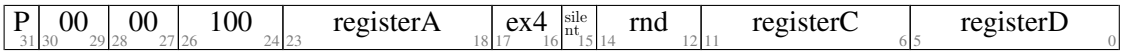
6.12.31 Extension Multiply Add 4x4 Matrix Byte to Word 0



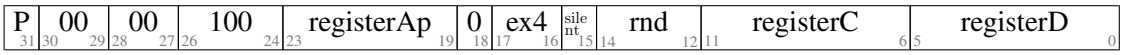
6.12.32 Extension Multiply Add 4x4 Matrix Byte to Word 1



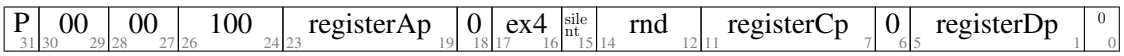
6.12.33 Extension Multiply Add Integer Floating Point Word Octuple



6.12.34 Extension Floating Point Fused Multiply Add Half to Word

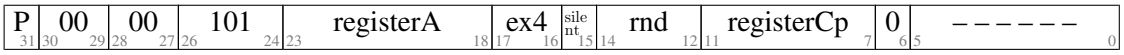


6.12.35 Extension Floating-Point Matrix Multiply Add Half Words to Words

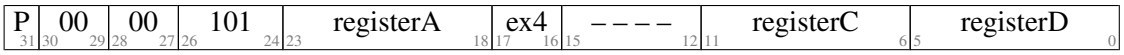


Kalray – all rights reserved

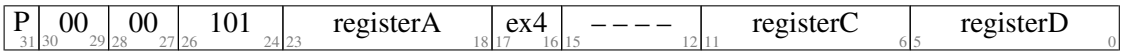
6.12.36 Extension Floating Point Narrow Word to Half Word



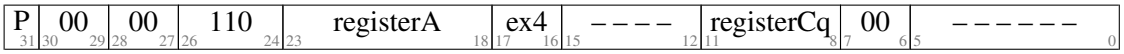
6.12.37 Extension Floating Point Minimum and Maximum



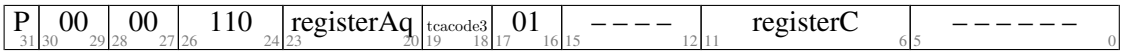
6.12.38 Extension Clamp Word Octuple



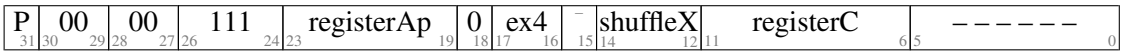
6.12.39 Extension Truncate Word to Byte Vector



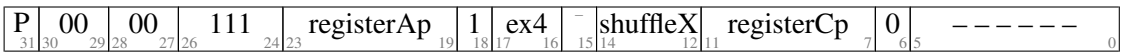
6.12.40 Extension Extend Bytes to Words



6.12.41 Extension Coprocessor Splat Word Octuple to Hexadecuple



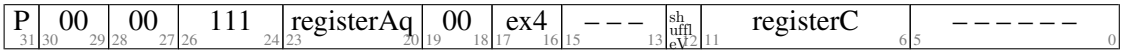
6.12.42 Extension Coprocessor Copy Word Hexadecuple



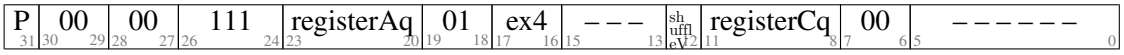
Kalray – all rights reserved

©2025 Kalray

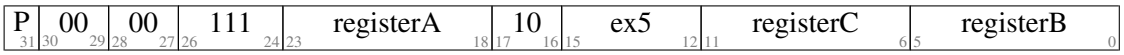
6.12.43 Extension Coprocessor Splat Word Octuple to Vector



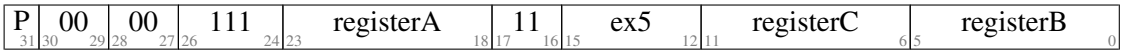
6.12.44 Extension Coprocessor Copy Word Vector



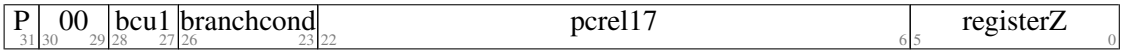
6.12.45 Extension Bit Matrix Multiply



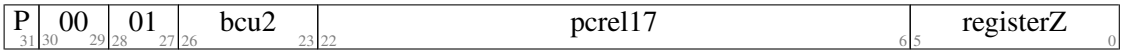
6.12.46 Extension Bitwise Octuple WOrd



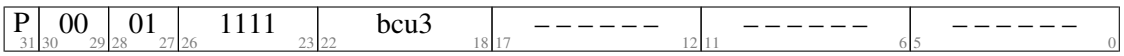
6.12.47 BCU Conditional Branches



6.12.48 BCU Hardware Loop Setup

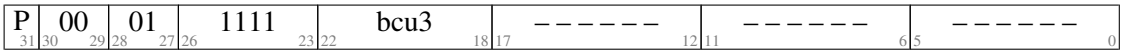


6.12.49 BCU TLB Management

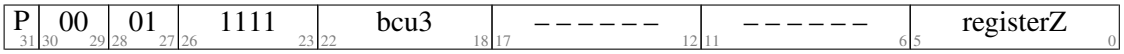


Kalray – all rights reserved

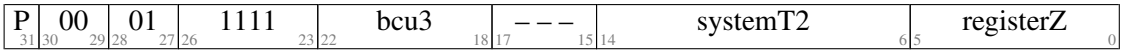
6.12.50 BCU Instruction Pipeline Control



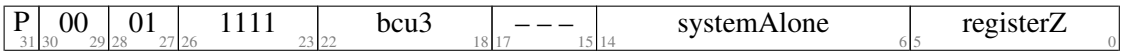
6.12.51 BCU Processing Group Instructions



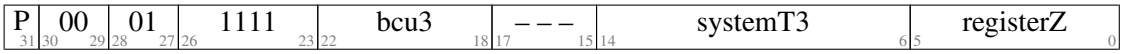
6.12.52 BCU Half-Word Effects



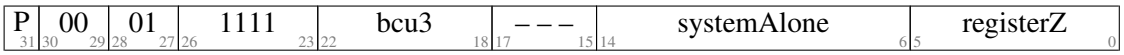
6.12.53 BCU Half-Word Effects Alone



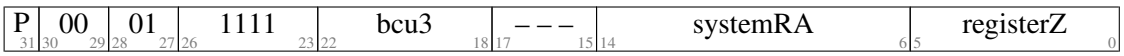
6.12.54 BCU Set System Registers



6.12.55 BCU Set System Register Alone



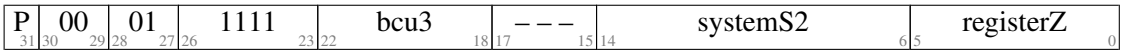
6.12.56 BCU Set System Register RA



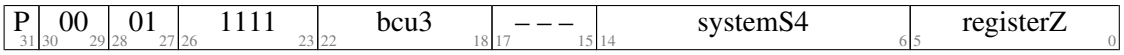
Kalray – all rights reserved

©2025 Kalray

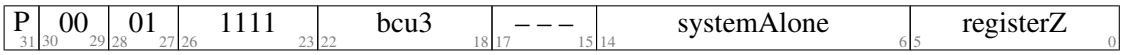
6.12.57 BCU Get System Registers



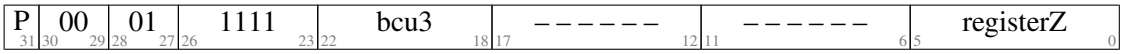
6.12.58 BCU General and System Register Swap



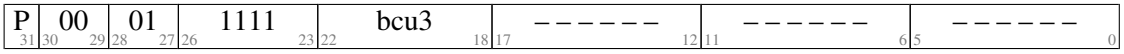
6.12.59 BCU General and System Register Swap Alone



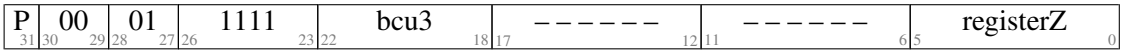
6.12.60 BCU Indirect Get System Registers



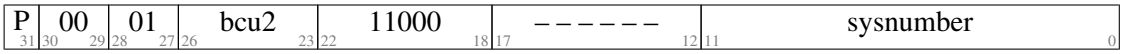
6.12.61 BCU Return from Call and System Calls



6.12.62 BCU Indirect Branch and Call from Register

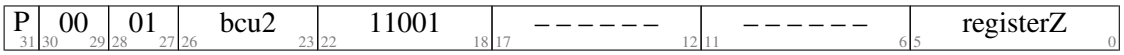


6.12.63 BCU Supervisor Call Immediate

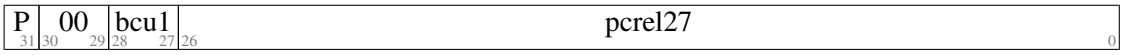


Kalray – all rights reserved

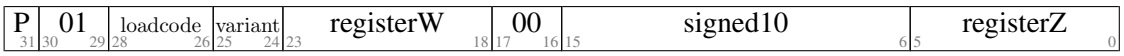
6.12.64 BCU Supervisor Call Register



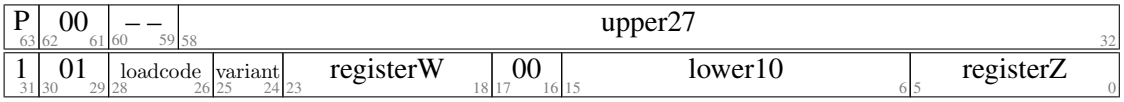
6.12.65 BCU Unconditional Branches



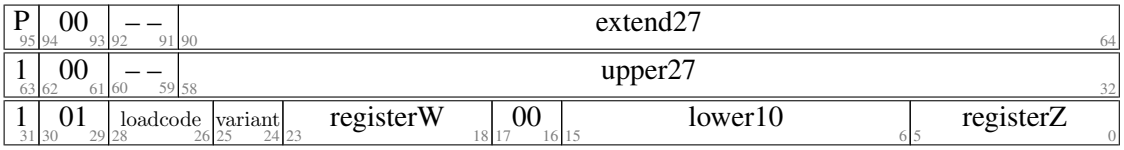
6.12.66 LSU Load Scalar Base Offset



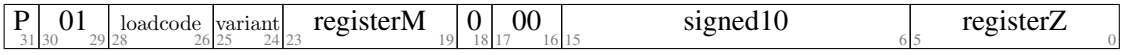
6.12.67 LSU Load Scalar Base Offset Extended



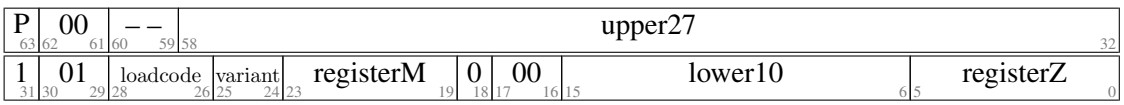
6.12.68 LSU Load Scalar Base Offset Extended Extended



6.12.69 LSU Load Quadruple Base Offset



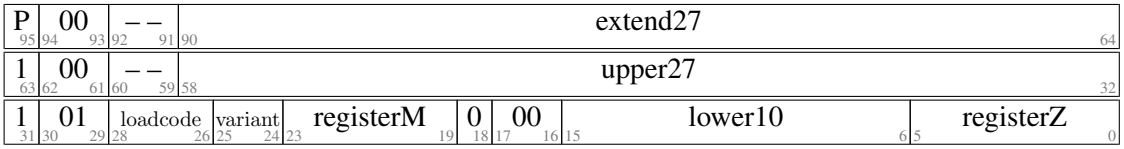
6.12.70 LSU Load Quadruple Base Offset Extended



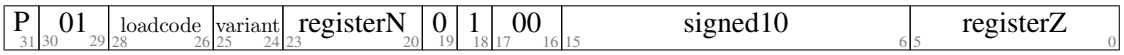
Kalray – all rights reserved

©2025 Kalray

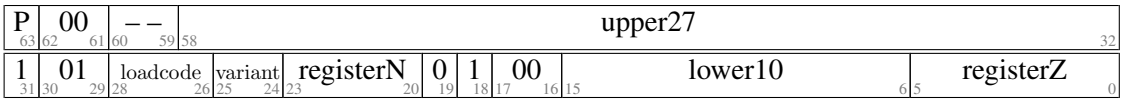
6.12.71 LSU Load Quadruple Base Offset Extended Extended



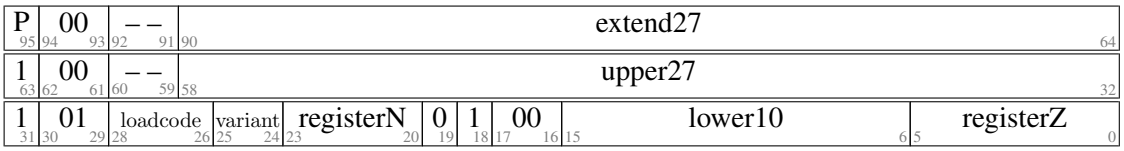
6.12.72 LSU Load Octuple Base Offset



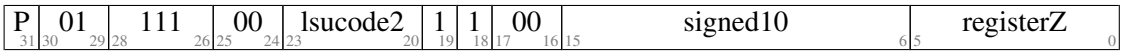
6.12.73 LSU Load Octuple Base Offset Extended



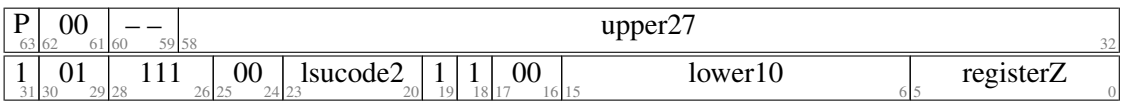
6.12.74 LSU Load Octuple Base Offset Extended Extended



6.12.75 LSU Memory Effects Base Offset

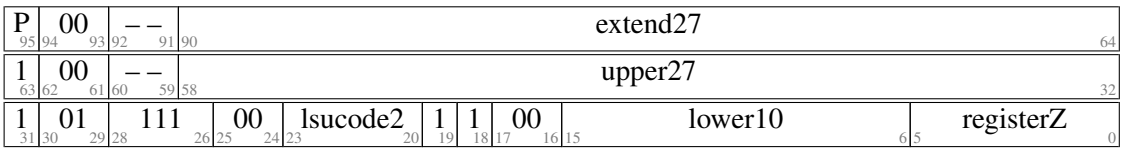


6.12.76 LSU Memory Effects Base Offset Extended

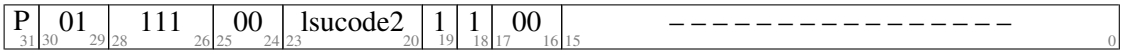


Kalray – all rights reserved

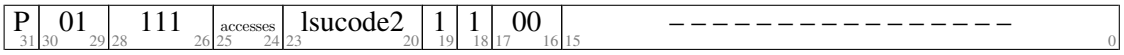
6.12.77 LSU Memory Effects Base Offset Extended Extended



6.12.78 LSU Memory and Cache Control

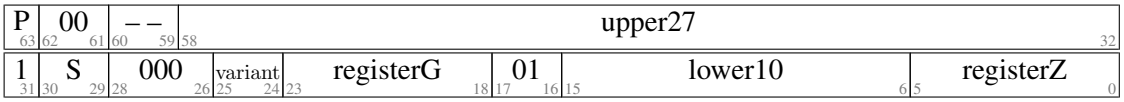


6.12.79 LSU Memory Fence

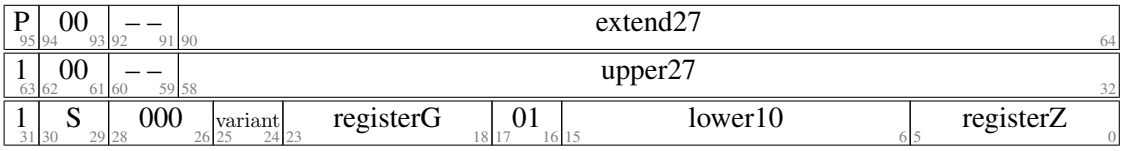


6.12.80 LSU Extension Load Octuple Base Offset

6.12.81 LSU Extension Load Octuple Base Offset Extended



6.12.82 LSU Extension Load Octuple Base Offset Extended Extended

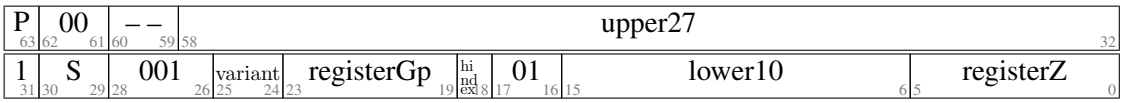


Kalray – all rights reserved

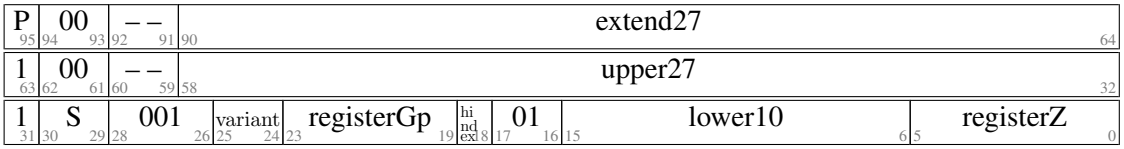
©2025 Kalray

6.12.83 LSU Extension Load Octuple Cross 2 Base Offset

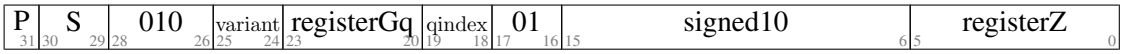
6.12.84 LSU Extension Load Octuple Cross 2 Base Offset Extended



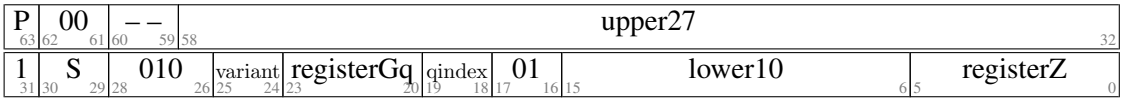
6.12.85 LSU Extension Load Octuple Cross 2 Base Offset Extended Extended



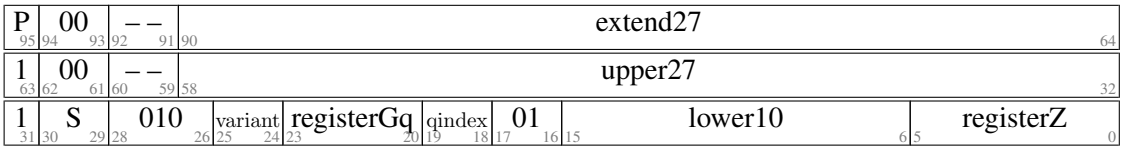
6.12.86 LSU Extension Load Octuple Cross 4 Base Offset



6.12.87 LSU Extension Load Octuple Cross 4 Base Offset Extended



6.12.88 LSU Extension Load Octuple Cross 4 Base Offset Extended Extended



Kalray – all rights reserved

6.12.89 LSU Store Scalar Base Offset

P	01	lsucode0	registerT	01	signed10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

6.12.90 LSU Store Scalar Base Offset Extended

P	00	– –	upper27	
63	62 61	60 59	58	32
1	01	lsucode0	registerT	01	lower10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

6.12.91 LSU Store Scalar Base Offset Extended Extended

P	00	– –	extend27	
95	94 93	92 91	90	64
1	00	– –	upper27	
63	62 61	60 59	58	32
1	01	lsucode0	registerT	01	lower10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

6.12.92 LSU Store Quadruple Base Offset

P	01	lsucode0	registerU	0	01	signed10	registerZ
31	30 29	28 24	23 19	18	17 16	15 6	5 0

6.12.93 LSU Store Quadruple Base Offset Extended

P	00	– –	upper27	
63	62 61	60 59	58	32
1	01	lsucode0	registerU	0	01	lower10	registerZ
31	30 29	28 24	23 19	18	17 16	15 6	5 0

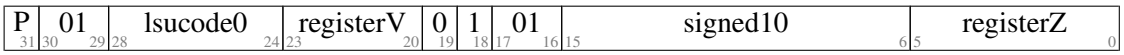
6.12.94 LSU Store Quadruple Base Offset Extended Extended

P	00	– –	extend27	
95	94 93	92 91	90	64
1	00	– –	upper27	
63	62 61	60 59	58	32
1	01	lsucode0	registerU	0	01	lower10	registerZ
31	30 29	28 24	23 19	18	17 16	15 6	5 0

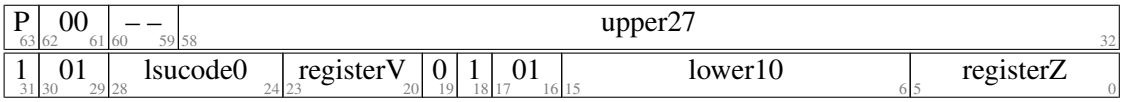
Kalray – all rights reserved

©2025 Kalray

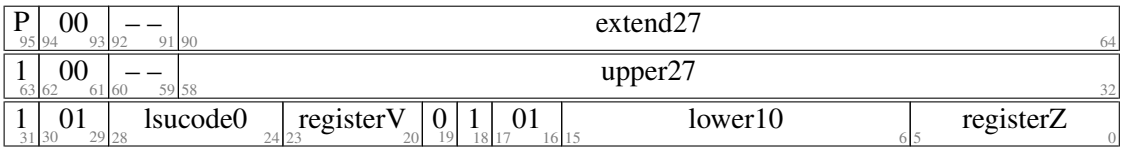
6.12.95 LSU Store Octuple Base Offset



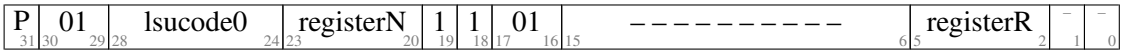
6.12.96 LSU Store Octuple Base Offset Extended



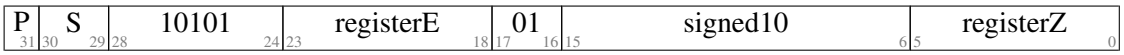
6.12.97 LSU Store Octuple Base Offset Extended Extended



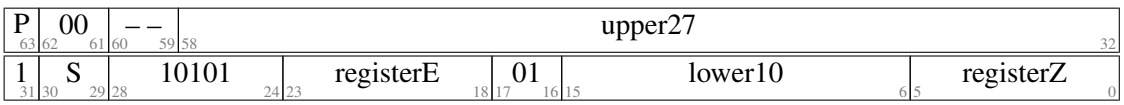
6.12.98 LSU Copy Octuple Word



6.12.99 LSU Extension Store Octuple Base Offset



6.12.100 LSU Extension Store Octuple Base Offset Extended



Kalray – all rights reserved

6.12.101 LSU Extension Store Octuple Base Offset Extended Extended

P	00	– –	extend27
95	94 93	92 91	90 64
1	00	– –	upper27
63	62 61	60 59	58 32
1	S	10101	registerE	01	lower10	registerZ
31	30 29	28 24	23 18	17 16	15 6	5 0

6.12.102 LSU Extension Store Octuple Cross 2 Base Offset

P	S	10110	registerEp	hi nd ex	01	signed10	registerZ
31	30 29	28 24	23 19	8	17 16	15 6	5 0

6.12.103 LSU Extension Store Octuple Cross 2 Base Offset Extended

P	00	– –	upper27
63	62 61	60 59	58 32
1	S	10110	registerEp	hi nd ex	01	lower10	registerZ
31	30 29	28 24	23 19	8	17 16	15 6	5 0

6.12.104 LSU Extension Store Octuple Cross 2 Base Offset Extended Extended

P	00	– –	extend27
95	94 93	92 91	90 64
1	00	– –	upper27
63	62 61	60 59	58 32
1	S	10110	registerEp	hi nd ex	01	lower10	registerZ
31	30 29	28 24	23 19	8	17 16	15 6	5 0

6.12.105 LSU Extension Store Octuple Cross 4 Base Offset

P	S	10111	registerEq	qindex	01	signed10	registerZ
31	30 29	28 24	23 20	19 18	17 16	15 6	5 0

6.12.106 LSU Extension Store Cross 4 Octuple Base Offset Extended

P	00	– –	upper27
63	62 61	60 59	58 32
1	S	10111	registerEq	qindex	01	lower10	registerZ
31	30 29	28 24	23 20	19 18	17 16	15 6	5 0

Kalray – all rights reserved

©2025 Kalray

6.12.107 LSU Extension Store Octuple Cross 4 Base Offset Extended Extended

P	00	– –	extend27
95	94 93	92 91	90 64
1	00	– –	upper27
63	62 61	60 59	58 32

1	S	10111	registerEq	qindex	01	lower10	registerZ
31	30 29	28 24	23 20	19 18	17 16	15 6	5 0

6.12.108 LSU Load Scalar Predicated Base

P	01	loadcode	variant	registerW	10	lsucond	registerY	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

6.12.109 LSU Load Scalar Predicated Base Offset

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	loadcode	variant	registerW	10	lsucond	registerY	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

6.12.110 LSU Load Scalar Predicated Base Double Offset

P	00	– –	extend27
95	94 93	92 91	90 64
1	00	– –	offset27
63	62 61	60 59	58 32

1	01	loadcode	variant	registerW	10	lsucond	registerY	registerZ
31	30 29	28 26	25 24	23 18	17 16	15 12	11 6	5 0

6.12.111 LSU Load Quadruple Predicated Base

P	01	loadcode	variant	registerM	0	10	lsucond	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

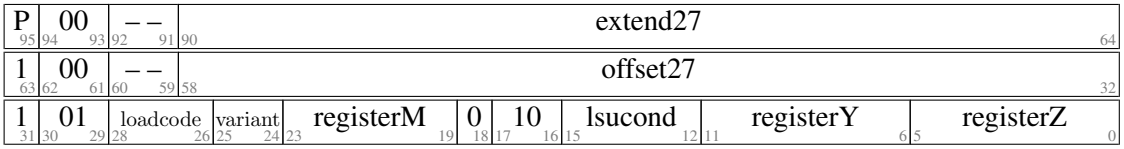
6.12.112 LSU Load Quadruple Predicated Base Offset

P	00	– –	offset27
63	62 61	60 59	58 32

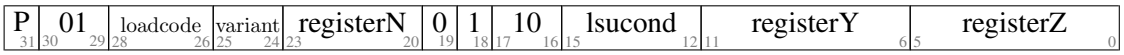
1	01	loadcode	variant	registerM	0	10	lsucond	registerY	registerZ
31	30 29	28 26	25 24	23 19	18	17 16	15 12	11 6	5 0

Kalray – all rights reserved

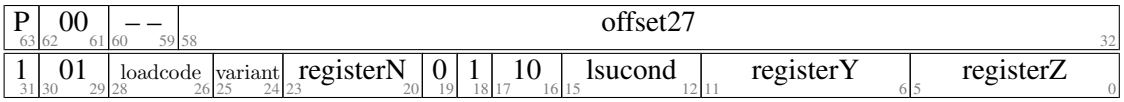
6.12.113 LSU Load Quadruple Predicated Base Double Offset



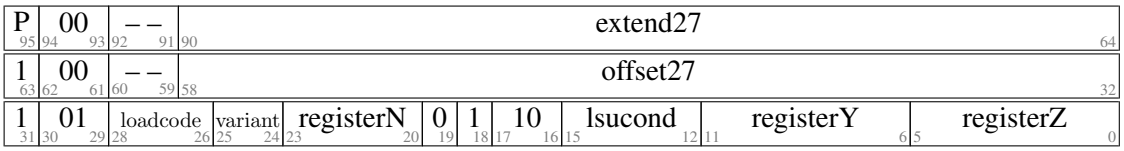
6.12.114 LSU Load Octuple Predicated Base



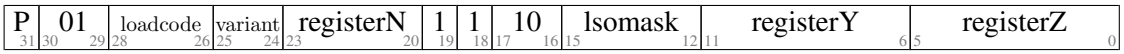
6.12.115 LSU Load Octuple Predicated Base Offset



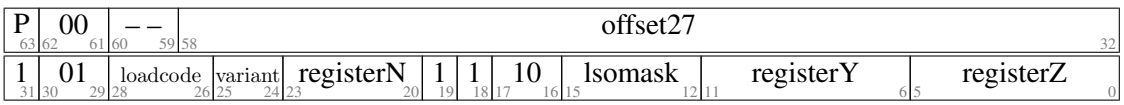
6.12.116 LSU Load Octuple Predicated Base Double Offset



6.12.117 LSU Load Octuple Masked Base



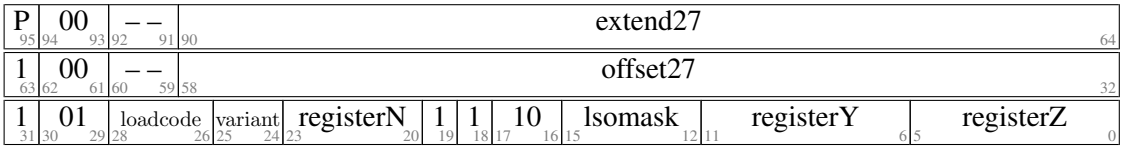
6.12.118 LSU Load Octuple Masked Base Offset



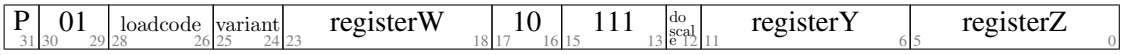
Kalray – all rights reserved

©2025 Kalray

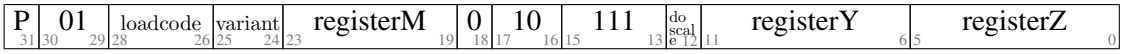
6.12.119 LSU Load Octuple Masked Base Double Offset



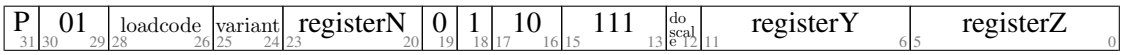
6.12.120 LSU Load Scalar Base + (Index*Scaling)



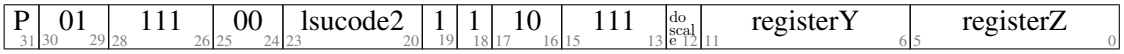
6.12.121 LSU Load Quadruple Base + (Index*Scaling)



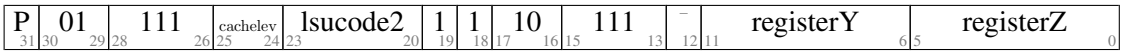
6.12.122 LSU Load Octuple Base + (Index*Scaling)



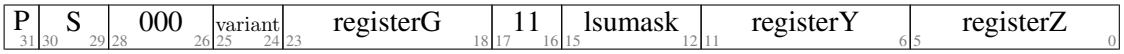
6.12.123 LSU Memory Effects Base + Index



6.12.124 LSU Memory Effects by Set / Way

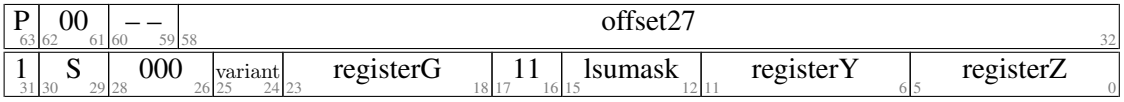


6.12.125 LSU Extension Load Octuple Masked Base

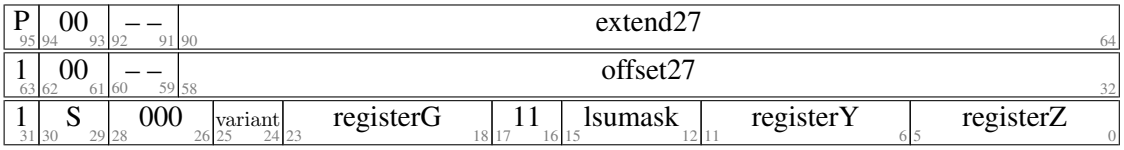


Kalray – all rights reserved

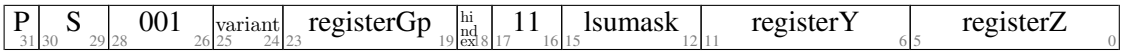
6.12.126 LSU Extension Load Octuple Masked Base Offset



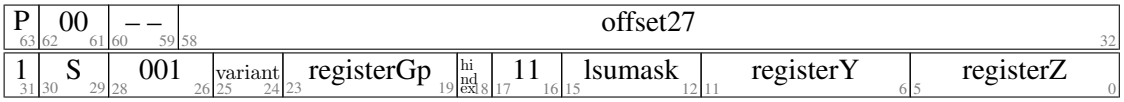
6.12.127 LSU Extension Load Octuple Masked Base Double Offset



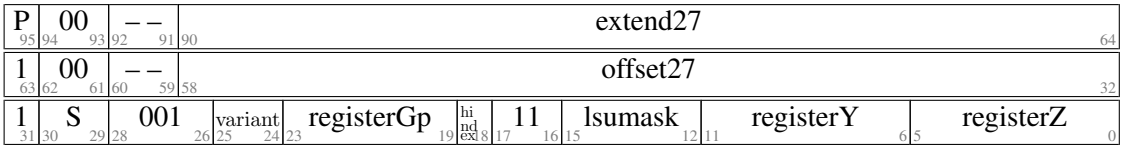
6.12.128 LSU Extension Load Octuple Cross 2 Masked Base



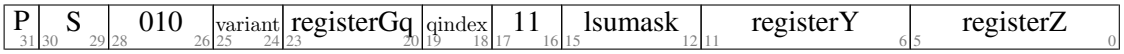
6.12.129 LSU Extension Load Octuple Cross 2 Masked Base Offset



6.12.130 LSU Extension Load Octuple Cross 2 Masked Base Double Offset



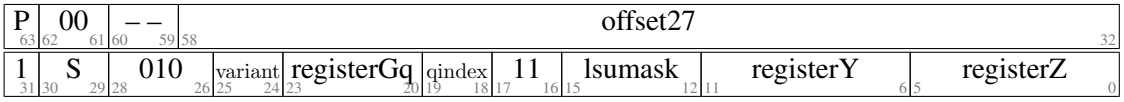
6.12.131 LSU Extension Load Octuple Cross 4 Masked Base



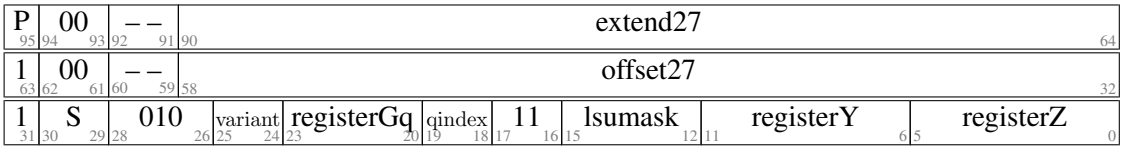
Kalray – all rights reserved

©2025 Kalray

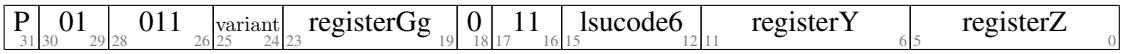
6.12.132 LSU Extension Load Octuple Cross 4 Masked Base Offset



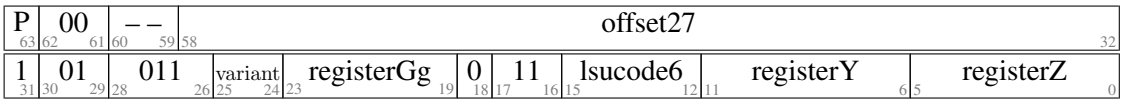
6.12.133 LSU Extension Load Octuple Cross 4 Masked Base Double Offset



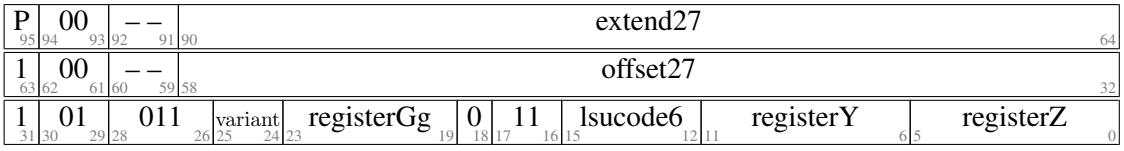
6.12.134 LSU Extension Preload 2-Register Buffer Packed Base



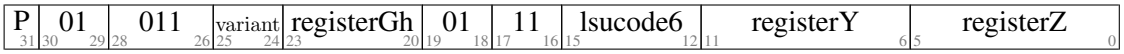
6.12.135 LSU Extension Preload 2-Register Buffer Packed Base Offset



6.12.136 LSU Extension Preload 2-Register Buffer Packed Base Double Offset



6.12.137 LSU Extension Preload 4-Register Buffer Packed Base



Kalray – all rights reserved

6.12.138 LSU Extension Preload 4-Register Buffer Packed Base Offset

P	00	– –	offset27							
63	62 61	60 59	58 32							
1	01	011	variant	registerGh	01	11	lsucode6	registerY	registerZ	
31	30 29	28 26	25 24	23 20	19 18	17 16	15 12	11 6	5 0	

6.12.139 LSU Extension Preload 4-Register Buffer Packed Base Double Offset

P	00	– –	extend27							
95	94 93	92 91	90 64							
1	00	– –	offset27							
63	62 61	60 59	58 32							
1	01	011	variant	registerGh	01	11	lsucode6	registerY	registerZ	
31	30 29	28 26	25 24	23 20	19 18	17 16	15 12	11 6	5 0	

6.12.140 LSU Extension Preload 8-Register Buffer Packed Base

P	01	011	variant	registerGi	011	11	lsucode6	registerY	registerZ	
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0	

6.12.141 LSU Extension Preload 8-Register Buffer Packed Base Offset

P	00	– –	offset27							
63	62 61	60 59	58 32							
1	01	011	variant	registerGi	011	11	lsucode6	registerY	registerZ	
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0	

6.12.142 LSU Extension Preload 8-Register Buffer Packed Base Double Offset

P	00	– –	extend27							
95	94 93	92 91	90 64							
1	00	– –	offset27							
63	62 61	60 59	58 32							
1	01	011	variant	registerGi	011	11	lsucode6	registerY	registerZ	
31	30 29	28 26	25 24	23 21	20 18	17 16	15 12	11 6	5 0	

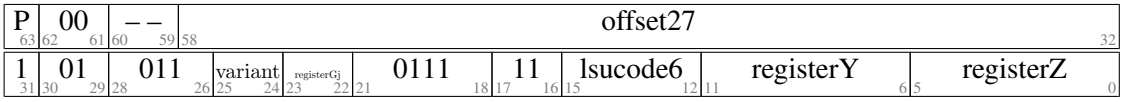
6.12.143 LSU Extension Preload 16-Register Buffer Packed Base

P	01	011	variant	registerGj	0111	11	lsucode6	registerY	registerZ	
31	30 29	28 26	25 24	23 22	21 18	17 16	15 12	11 6	5 0	

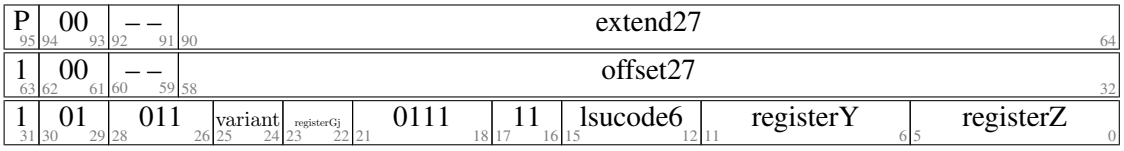
Kalray – all rights reserved

©2025 Kalray

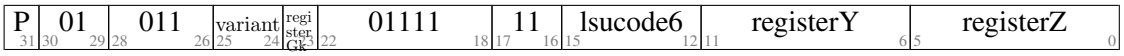
6.12.144 LSU Extension Preload 16-Register Buffer Packed Base Offset



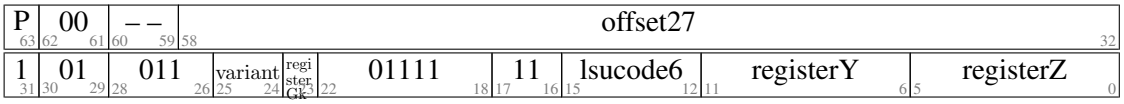
6.12.145 LSU Extension Preload 16-Register Buffer Packed Base Double Offset



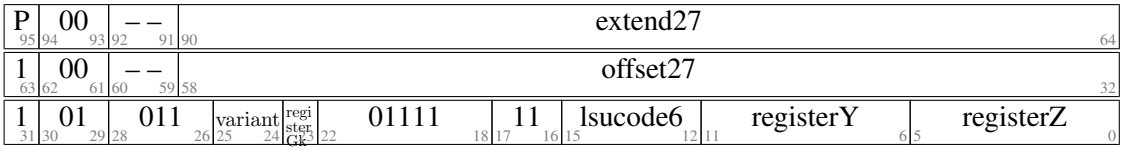
6.12.146 LSU Extension Preload 32-Register Buffer Packed Base



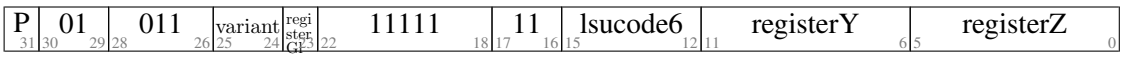
6.12.147 LSU Extension Preload 32-Register Buffer Packed Base Offset



6.12.148 LSU Extension Preload 32-Register Buffer Packed Base Double Offset

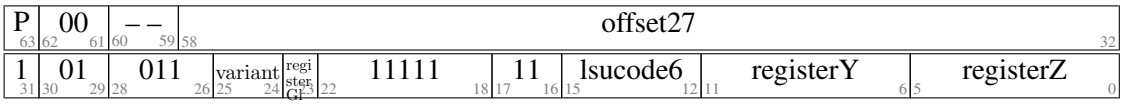


6.12.149 LSU Extension Preload 64-Register Buffer Packed Base

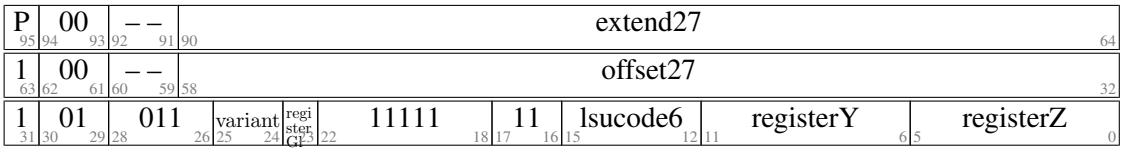


Kalray – all rights reserved

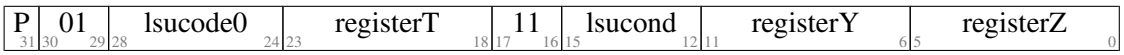
6.12.150 LSU Extension Preload 64-Register Buffer Packed Base Offset



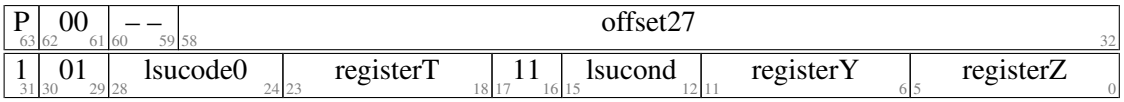
6.12.151 LSU Extension Preload 64-Register Buffer Packed Base Double Offset



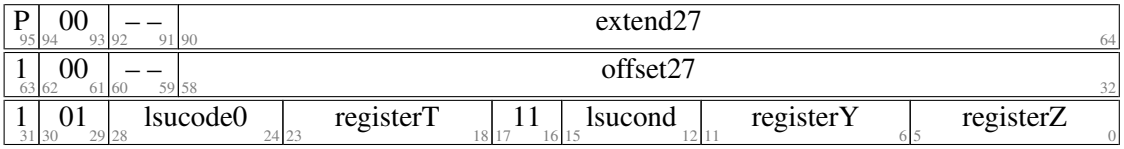
6.12.152 LSU Store Scalar Predicated Base



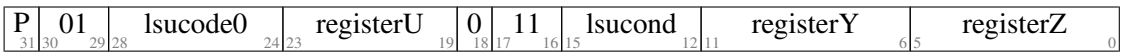
6.12.153 LSU Store Scalar Predicated Base Offset



6.12.154 LSU Store Scalar Predicated Base Double Offset



6.12.155 LSU Store Quadruple Predicated Base



Kalray – all rights reserved

©2025 Kalray

6.12.156 LSU Store Quadruple Predicated Base Offset

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	lsucode0	registerU	0	11	lsucond	registerY	registerZ
31	30 29	28 24	23 19	18	17 16	15 12	11 6	5 0

6.12.157 LSU Store Quadruple Predicated Base Double Offset

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	offset27
63	62 61	60 59	58 32

1	01	lsucode0	registerU	0	11	lsucond	registerY	registerZ
31	30 29	28 24	23 19	18	17 16	15 12	11 6	5 0

6.12.158 LSU Store Octuple Predicated Base

P	01	lsucode0	registerV	0	1	11	lsucond	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

6.12.159 LSU Store Octuple Predicated Base Offset

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	lsucode0	registerV	0	1	11	lsucond	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

6.12.160 LSU Store Octuple Predicated Base Double Offset

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	offset27
63	62 61	60 59	58 32

1	01	lsucode0	registerV	0	1	11	lsucond	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

6.12.161 LSU Store Octuple Masked Base

P	01	lsucode0	registerV	1	1	11	lsomask	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

Kalray – all rights reserved

6.12.162 LSU Store Octuple Masked Base Offset

P	00	– –	offset27
63	62 61	60 59	58 32

1	01	lsucode0	registerV	1	1	11	lsomask	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

6.12.163 LSU Store Octuple Masked Base Double Offset

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	offset27
63	62 61	60 59	58 32

1	01	lsucode0	registerV	1	1	11	lsomask	registerY	registerZ
31	30 29	28 24	23 20	19	18	17 16	15 12	11 6	5 0

6.12.164 LSU Extension Store Octuple Masked Base

P	S	10101	registerE	11	lsumask	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

6.12.165 LSU Extension Store Octuple Masked Base Offset

P	00	– –	offset27
63	62 61	60 59	58 32

1	S	10101	registerE	11	lsumask	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

6.12.166 LSU Extension Store Octuple Masked Base Double Offset

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	offset27
63	62 61	60 59	58 32

1	S	10101	registerE	11	lsumask	registerY	registerZ
31	30 29	28 24	23 18	17 16	15 12	11 6	5 0

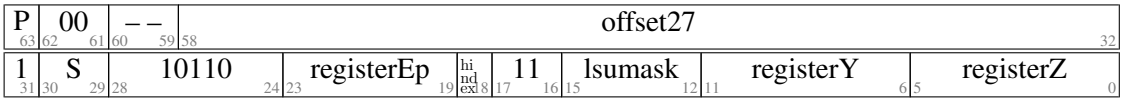
6.12.167 LSU Extension Store Octuple Cross 2 Masked Base

P	S	10110	registerEp	hi nd exl	11	lsumask	registerY	registerZ
31	30 29	28 24	23 19	8	17 16	15 12	11 6	5 0

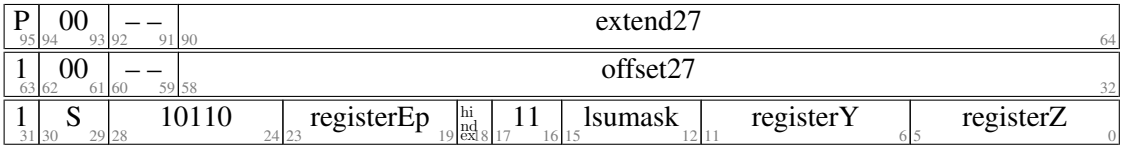
Kalray – all rights reserved

©2025 Kalray

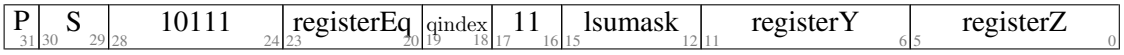
6.12.168 LSU Extension Store Octuple Cross 2 Masked Base Offset



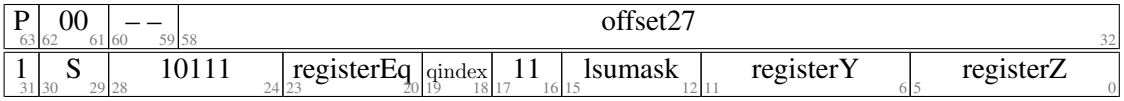
6.12.169 LSU Extension Store Octuple Cross 2 Masked Base Double Offset



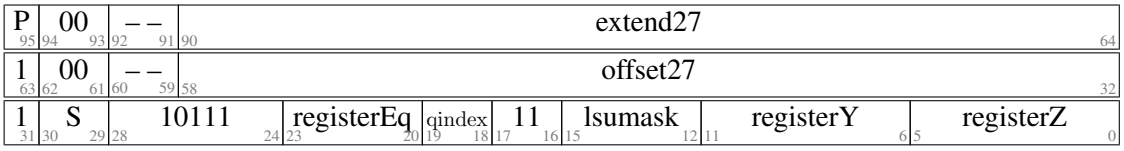
6.12.170 LSU Extension Store Octuple Cross 4 Masked Base



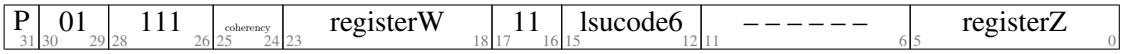
6.12.171 LSU Extension Store Octuple Cross 4 Masked Base Offset



6.12.172 LSU Extension Store Octuple Cross 4 Masked Base Double Offset

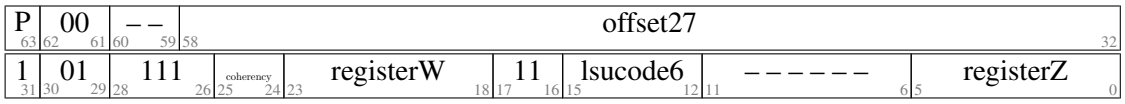


6.12.173 LSU Atomic Load Scalar Base

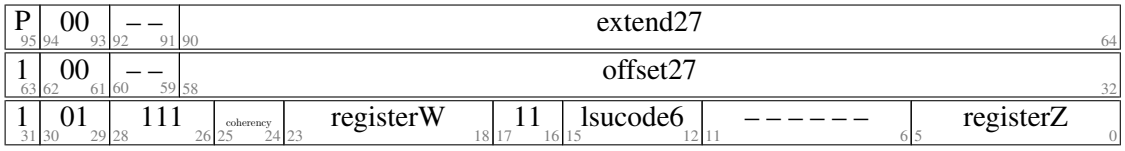


Kalray – all rights reserved

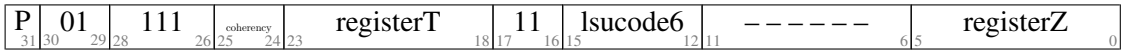
6.12.174 LSU Atomic Load Scalar Base Offset



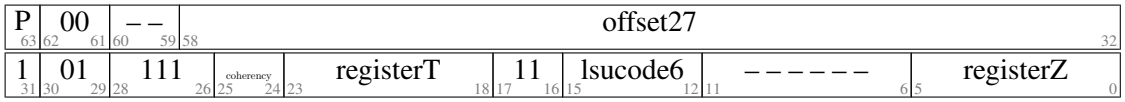
6.12.175 LSU Atomic Load Scalar Base Double Offset



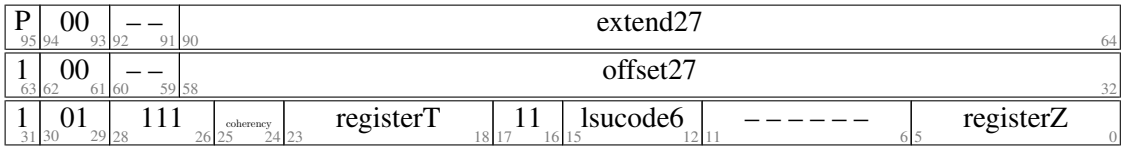
6.12.176 LSU Atomic Store Scalar Base



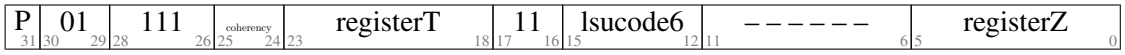
6.12.177 LSU Atomic Store Scalar Base Offset



6.12.178 LSU Atomic Store Scalar Base Double Offset



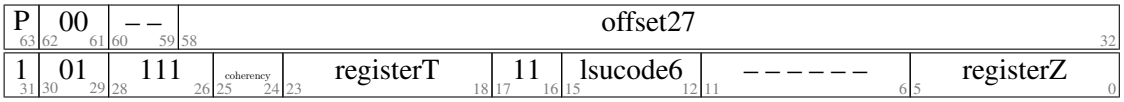
6.12.179 LSU Atomic Load Add Scalar Base



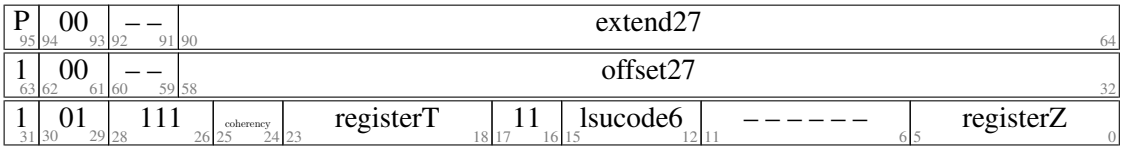
Kalray – all rights reserved

©2025 Kalray

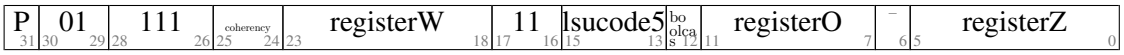
6.12.180 LSU Atomic Load Add Scalar Base Offset



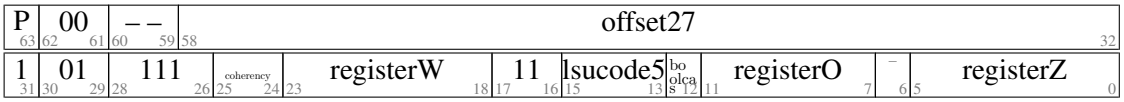
6.12.181 LSU Atomic Load Add Scalar Base Double Offset



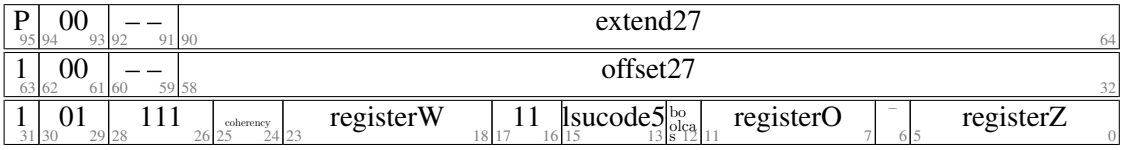
6.12.182 LSU Atomic Compare and Swap Scalar Base



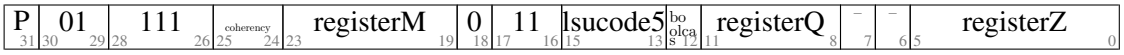
6.12.183 LSU Atomic Compare and Swap Scalar Base Offset



6.12.184 LSU Atomic Compare and Swap Scala Base Double Offset

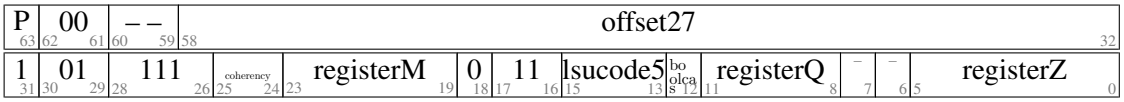


6.12.185 LSU Atomic Compare and Swap Quadruple Base

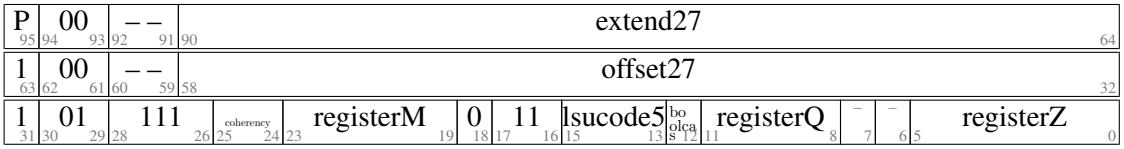


Kalray – all rights reserved

6.12.186 LSU Atomic Compare and Swap Quadruple Base Offset

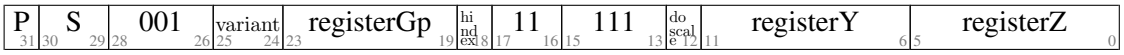


6.12.187 LSU Atomic Compare and Swap Quadruple Base Double Offset

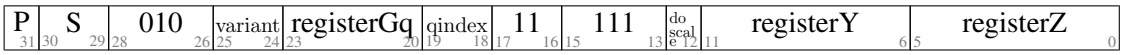


6.12.188 LSU Extension Load Base + (Index*Scaling)

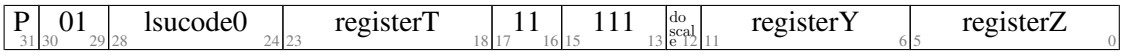
6.12.189 LSU Extension Load Octuple Cross 2 Base + (Index*Scaling)



6.12.190 LSU Extension Load Octuple Cross 4 Base + (Index*Scaling)



6.12.191 LSU Store Scalar Base + (Index*Scaling)

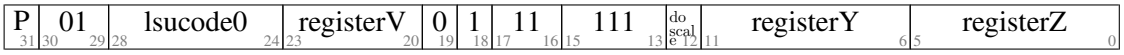


6.12.192 LSU Store Quadruple Base + (Index*Scaling)

Kalray – all rights reserved

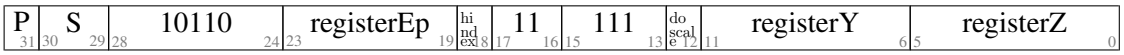
©2025 Kalray

6.12.193 LSU Store Octuple Base + (Index*Scaling)

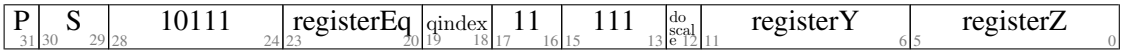


6.12.194 LSU Extension Store Octuple Base + (Index*Scaling)

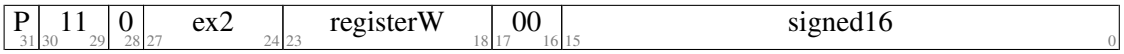
6.12.195 LSU Extension Store Quadruple Cross 2 Base + (Index*Scaling)



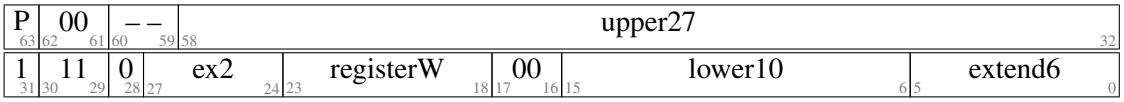
6.12.196 LSU Extension Store Octuple Cross 4 Base + (Index*Scaling)



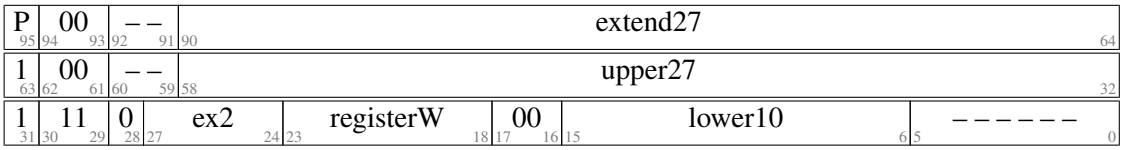
6.12.197 ALU Make from Immediate



6.12.198 ALU Make from Immediate Extended



6.12.199 ALU Make from Immediate Extended Extended



Kalray – all rights reserved

6.12.200 ALU Write Read Double Immediate

P	11	0	ex2	registerW	00	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

6.12.201 ALU Write Read Double Immediate Extended

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	ex2	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

6.12.202 ALU Write Read Double Immediate Extended Extended

P	00	– –	extend27
95	94 93	92 91	90 64

1	00	– –	upper27
63	62 61	60 59	58 32

1	11	0	ex2	registerW	00	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

6.12.203 ALU Compare Write Read Immediate

P	S	0	comparison	registerW	01	signed10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

6.12.204 ALU Compare Write Read Immediate Extended

P	00	– –	upper27
63	62 61	60 59	58 32

1	S	0	comparison	registerW	01	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

6.12.205 ALU Compare Write Read Immediate Extended Extended

P	00	– –	extend27
95	94 93	92 91	90 64

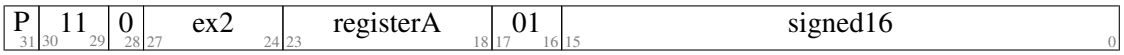
1	00	– –	upper27
63	62 61	60 59	58 32

1	S	0	comparison	registerW	01	lower10	registerZ
31	30 29	28	27 24	23 18	17 16	15 6	5 0

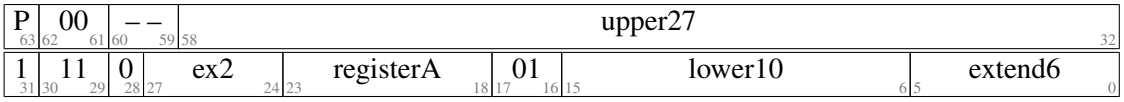
Kalray – all rights reserved

©2025 Kalray

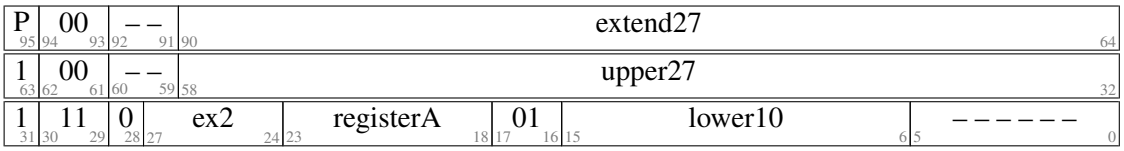
6.12.206 ALU Extension Splat Double To Octuple Word Immediate



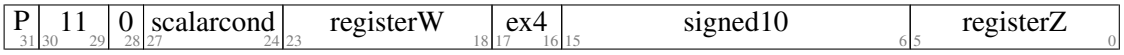
6.12.207 ALU Extension Splat Double To Octuple Word Immediate Extended



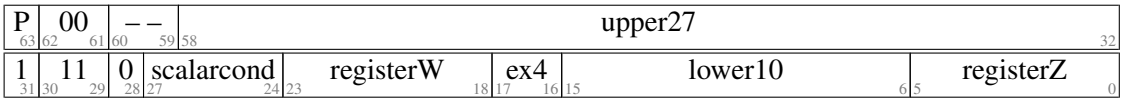
6.12.208 ALU Extension Splat Double To Octuple Word Immediate Extended Extended



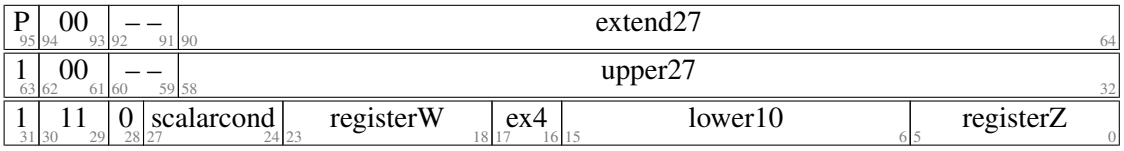
6.12.209 ALU Double Conditional Move Write Read Immediate



6.12.210 ALU Double Conditional Move Write Read Immediate Extended

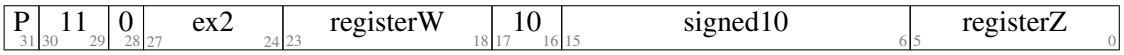


6.12.211 ALU Double Conditional Move Write Read Immediate Extended Extended

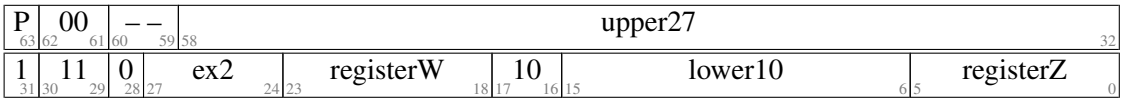


Kalray – all rights reserved

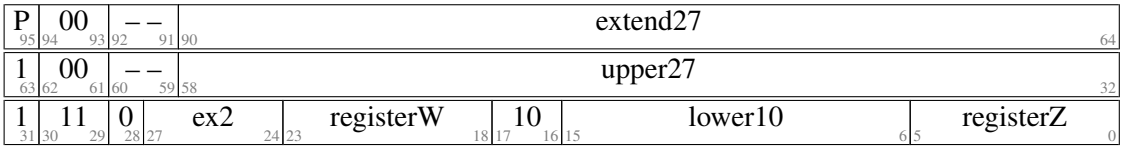
6.12.212 ALU Bit Matrix Write Read Immediate



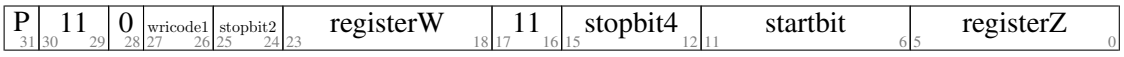
6.12.213 ALU Bit Matrix Write Read Immediate Extended



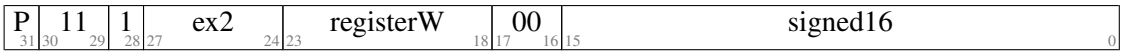
6.12.214 ALU Bit Matrix Write Read Immediate Extended Extended



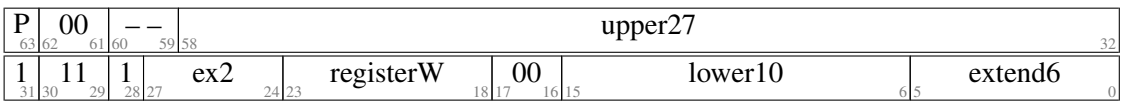
6.12.215 ALU Write Read Bitmask



6.12.216 ALU Add PC to Immediate



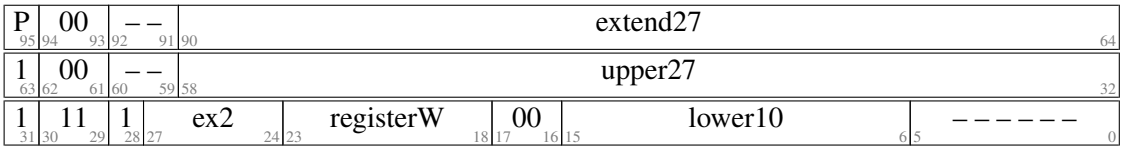
6.12.217 ALU Add PC to Immediate Extended



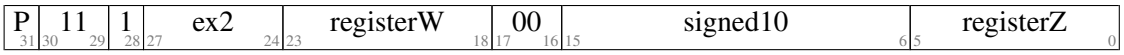
Kalray – all rights reserved

©2025 Kalray

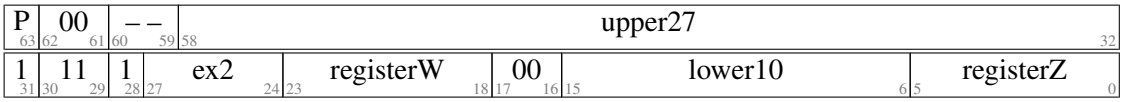
6.12.218 ALU Add PC to Immediate Extended Extended



6.12.219 ALU Write Read Immediate Word

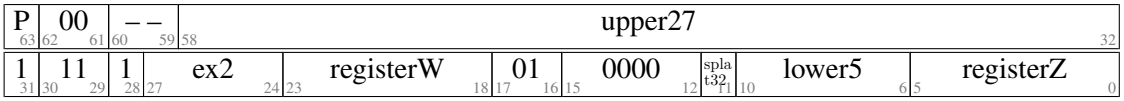


6.12.220 ALU Write Read Immediate Extended Word

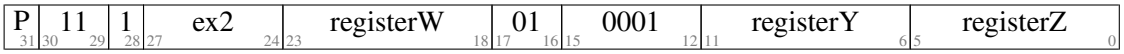


6.12.221 ALU Double Write Read Read 0

6.12.222 ALU Double Write Read Read 0 Magic



6.12.223 ALU Word Write Read Read 0



Kalray – all rights reserved

6.12.224 ALU Word Write Read Read 0 Wrapped

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	ex2	registerW	01	0001	0	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11	10 6	5 0

6.12.225 ALU Word Pair Write Read Read 0

P	11	1	ex2	registerW	01	0010	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

6.12.226 ALU Word Pair Write Read Read 0 Magic

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	ex2	registerW	01	0010	spla t32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	f1	10 6	5 0

6.12.227 ALU Half Quadruple Write Read Read 0

P	11	1	ex2	registerW	01	0011	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

6.12.228 ALU Half Quadruple Write Read Read 0 Magic

P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	ex2	registerW	01	0011	spla t32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	f1	10 6	5 0

6.12.229 ALU Double Write Read Read 1

P	11	1	ex2	registerW	01	0100	registerY	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	11 6	5 0

6.12.230 ALU Double Write Read Read 1 Magic

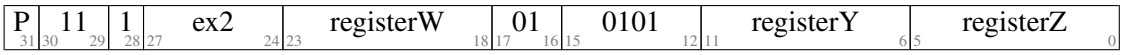
P	00	– –	upper27
63	62 61	60 59	58 32

1	11	1	ex2	registerW	01	0100	spla t32	lower5	registerZ
31	30 29	28	27 24	23 18	17 16	15 12	f1	10 6	5 0

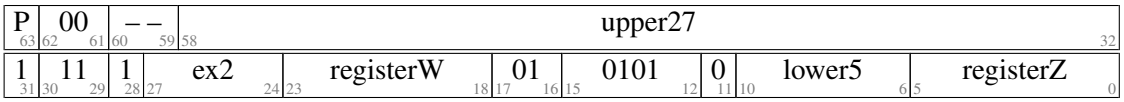
Kalray – all rights reserved

©2025 Kalray

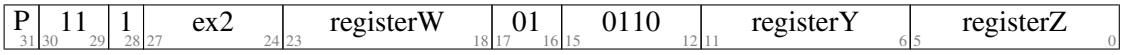
6.12.231 ALU Word Write Read Read 1



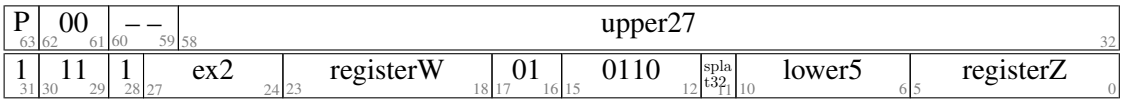
6.12.232 ALU Word Write Read Read 1 Wrapped



6.12.233 ALU Word Pair Write Read Read 1

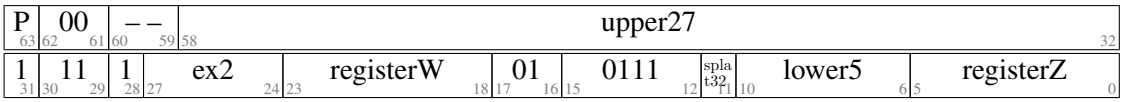


6.12.234 ALU Word Pair Write Read Read 1 Magic

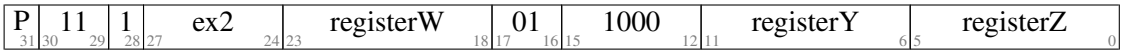


6.12.235 ALU Write Read Read Half Word Quadruple 1

6.12.236 ALU Half Quadruple Write Read Read 1 Magic

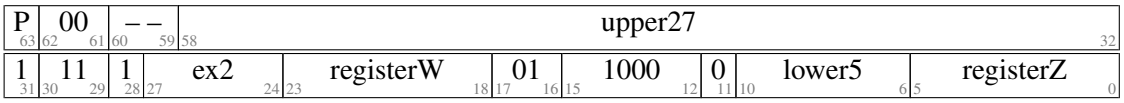


6.12.237 ALU Word to Double Write Read Read 2

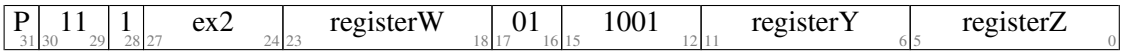


Kalray – all rights reserved

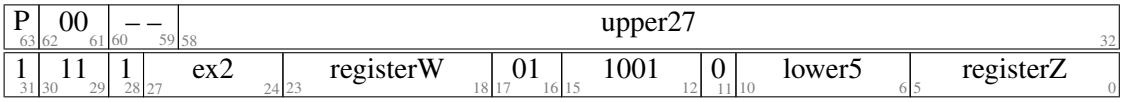
6.12.238 ALU Word to Double Write Read Read 2 Wrapped



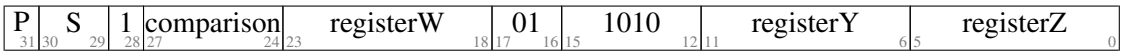
6.12.239 ALU Word to Double Write Read Read 3



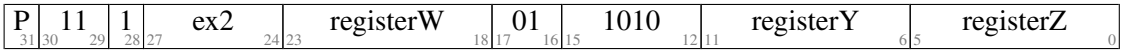
6.12.240 ALU Word to Double Write Read Read 2 Wrapped



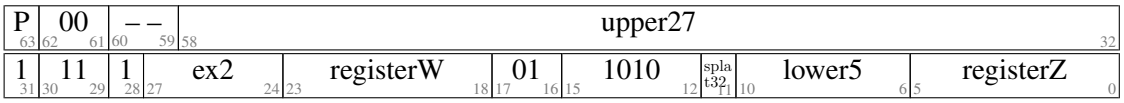
6.12.241 ALU Double Comparison Write Read Read



6.12.242 ALU Double Write Read Read Special



6.12.243 ALU Double Write Read Read Special Magic

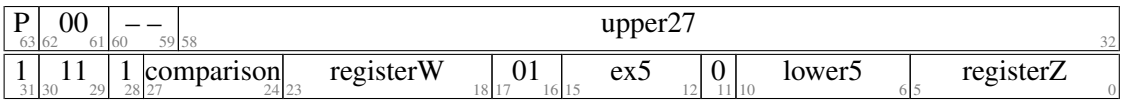


6.12.244 ALU Double Comparison Write Read Read

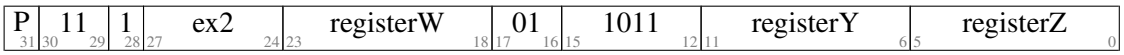
Kalray – all rights reserved

©2025 Kalray

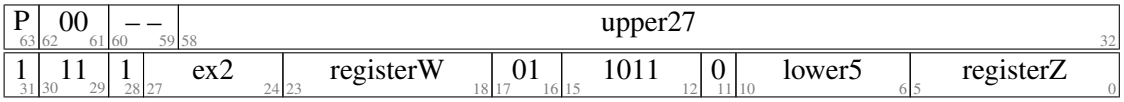
6.12.245 ALU Double Comparison Write Read Read Wrapped



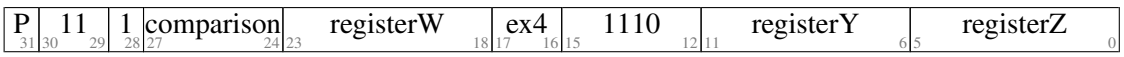
6.12.246 ALU Write Read Read Special Word



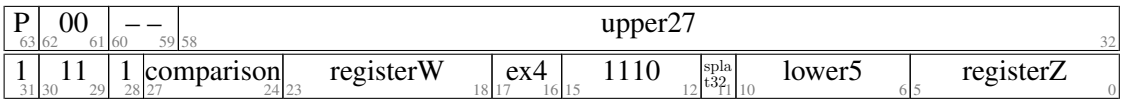
6.12.247 ALU Write Read Read Special Word Wrapped



6.12.248 ALU Comparison Write Read Read Word Pair



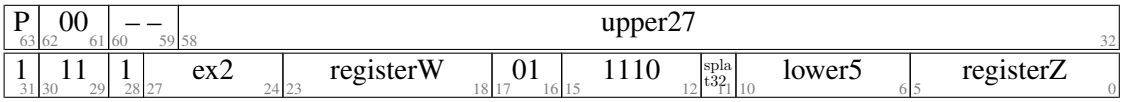
6.12.249 ALU Comparison Write Read Read Word Pair Magic



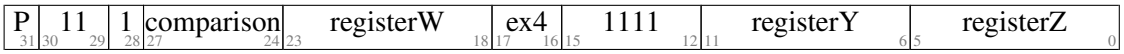
6.12.250 ALU Write Read Read Special Word Pair

Kalray – all rights reserved

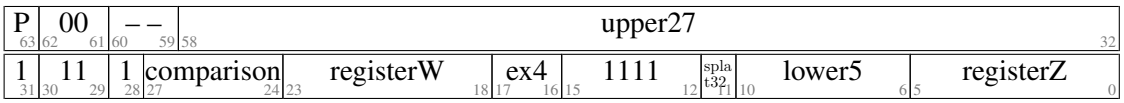
6.12.251 ALU Write Read Read Special Word Pair Magic



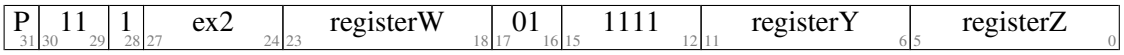
6.12.252 ALU Comparison Write Read Read Half Word Quadruple



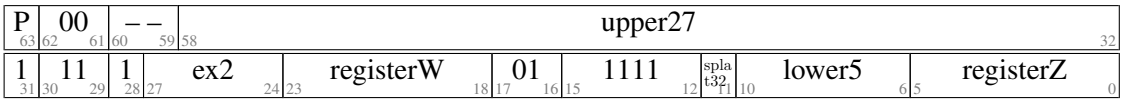
6.12.253 ALU Comparison Write Read Read Half Word Quadruple Magic



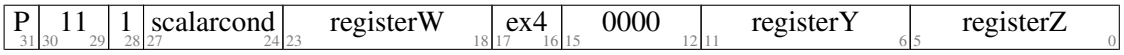
6.12.254 ALU Write Read Read Special Half Quadruple



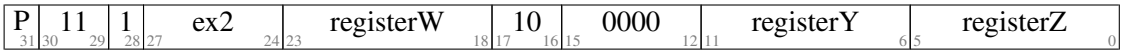
6.12.255 ALU Write Read Read Special Half Quadruple Magic



6.12.256 ALU Double Conditional Move Write Read Read



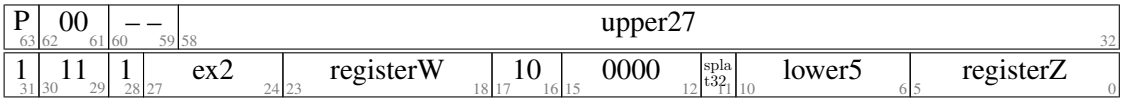
6.12.257 ALU Bit Matrix Write Read Read



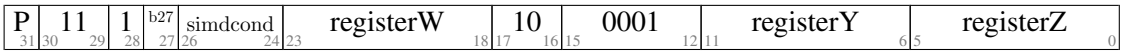
Kalray – all rights reserved

©2025 Kalray

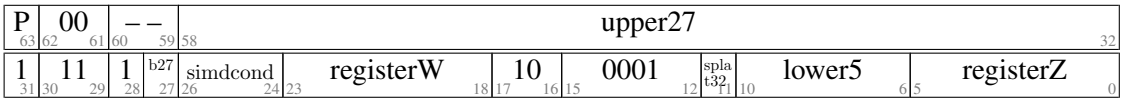
6.12.258 ALU Bit Matrix Write Read Read Magic



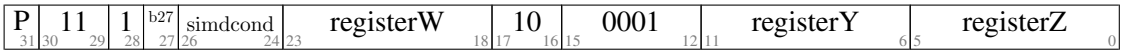
6.12.259 ALU Move Write Read Read Word Pair



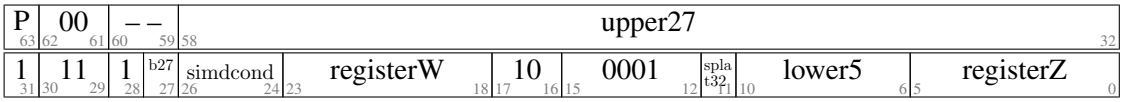
6.12.260 ALU Move Write Read Read Word Pair Magic



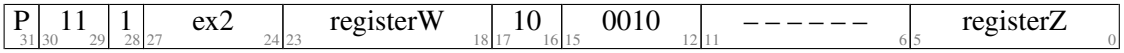
6.12.261 ALU Move Write Read Read Half Word Quadruple



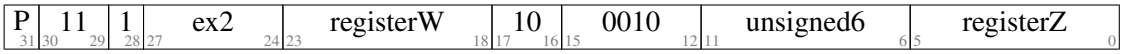
6.12.262 ALU Move Write Read Read Half Word Quadruple Magic



6.12.263 ALU Bit Write Read Double

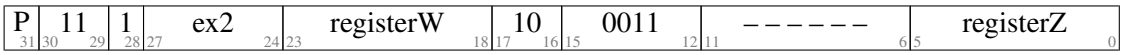


6.12.264 ALU Double Shift Rotate Write Read Immediate

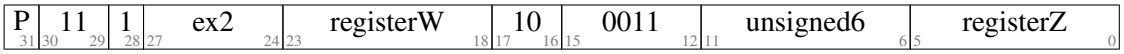


Kalray – all rights reserved

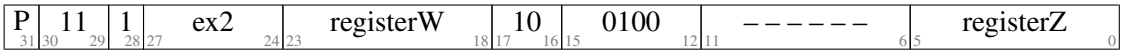
6.12.265 ALU Bit Write Read Word



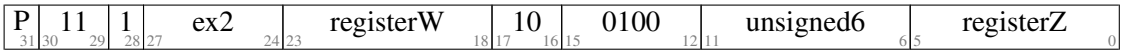
6.12.266 ALU Shift Rotate Write Read Immediate



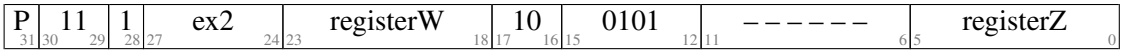
6.12.267 ALU Bit Write Read Word Pair



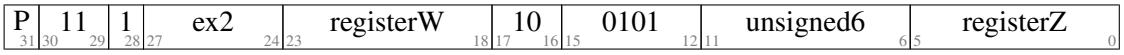
6.12.268 ALU Word Pair Shift Rotate Write Read Immediate



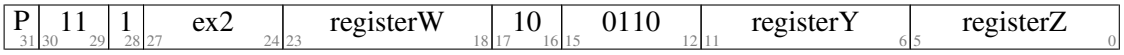
6.12.269 ALU Arithmetic Write Read Half Word Quadruple



6.12.270 ALU Shift Rotate Write Read Immediate Half Word Quadruple



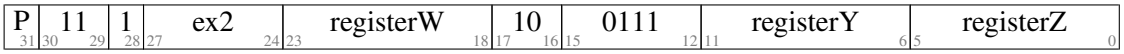
6.12.271 ALU Double Shift Rotate Write Read Read



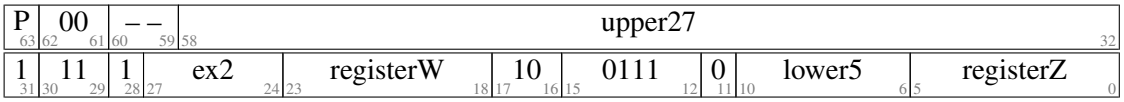
Kalray – all rights reserved

©2025 Kalray

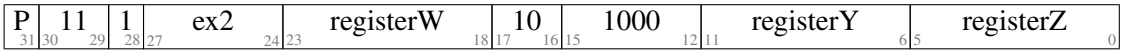
6.12.272 ALU Shift Rotate Write Read Read



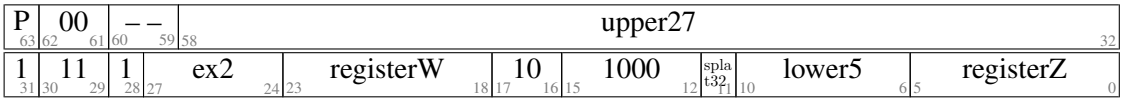
6.12.273 ALU Double Write Read Read 1 Wrapped



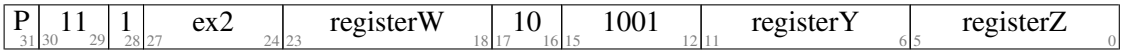
6.12.274 ALU Word Pair Shift Rotate Write Read Read



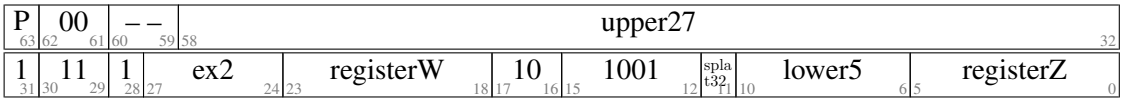
6.12.275 ALU Word Pair Shift Rotate Write Read Read Magic



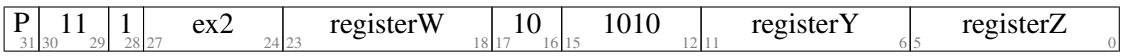
6.12.276 ALU Shift Rotate Write Read Read Half Word Quadruple



6.12.277 ALU Shift Rotate Write Read Read Half Word Quadruple Magic

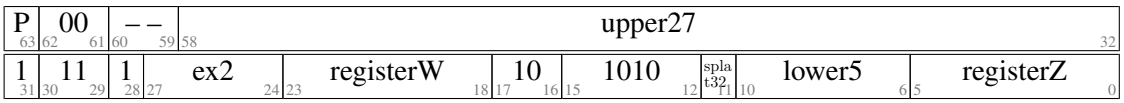


6.12.278 ALU Write Read Read Byte Octuple 0

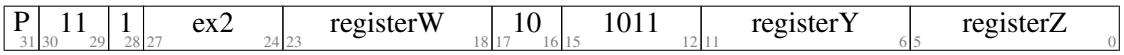


Kalray – all rights reserved

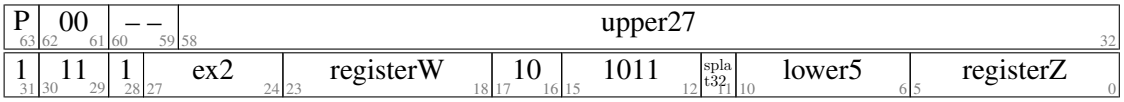
6.12.279 ALU Write Read Read Byte Octuple 0 Magic



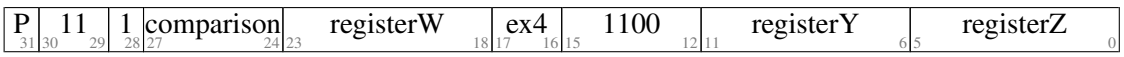
6.12.280 ALU Write Read Read Byte Octuple 1



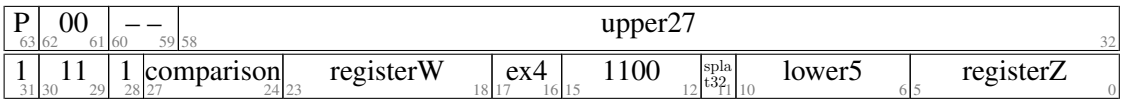
6.12.281 ALU Write Read Read Byte Octuple 1 Magic



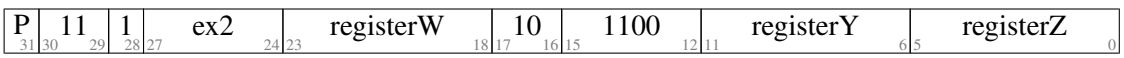
6.12.282 ALU Comparison Write Read Read Byte Octuple



6.12.283 ALU Comparison Write Read Read Byte Octuple Magic



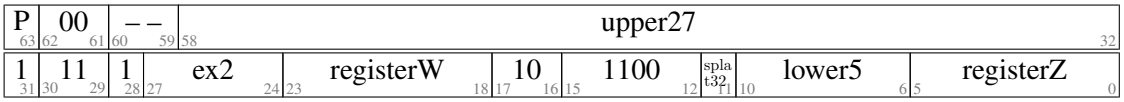
6.12.284 ALU Write Read Read Special Byte Octuple



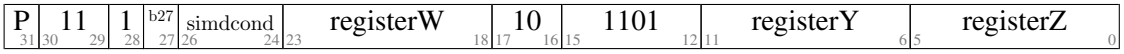
Kalray – all rights reserved

©2025 Kalray

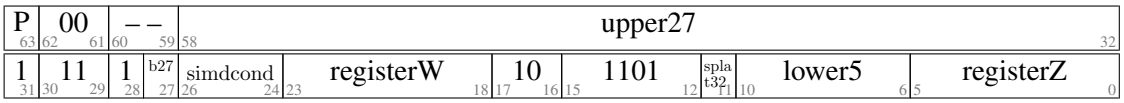
6.12.285 ALU Write Read Read Special Byte Octuple Magic



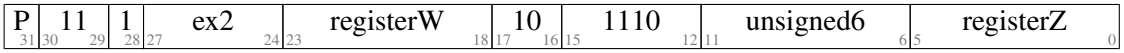
6.12.286 ALU Conditional Move Write Read Read Byte Octuple



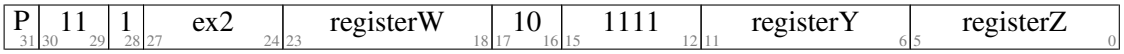
6.12.287 ALU Move Write Read Read Byte Octuple Magic



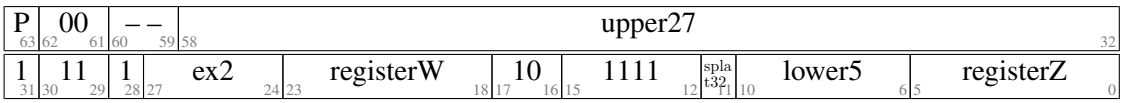
6.12.288 ALU Shift Rotate Write Read Immediate Byte Octuple



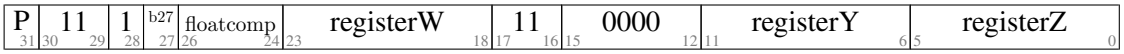
6.12.289 ALU Shift Rotate Write Read Read Byte Octuple



6.12.290 ALU Shift Rotate Write Read Read Byte Octuple Magic

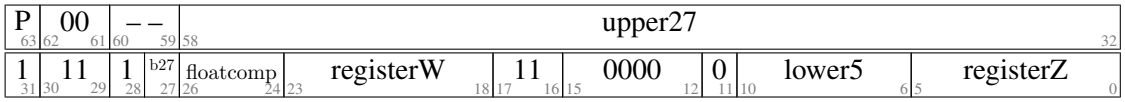


6.12.291 ALU Floating-Point Comparison Write Read Read Scalar

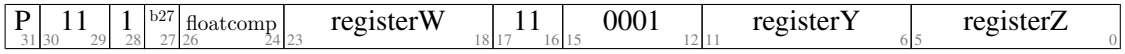


Kalray – all rights reserved

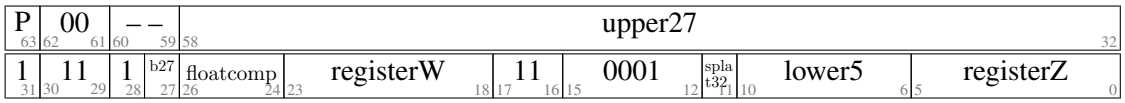
6.12.292 ALU Floating-Point Comparison Write Read Immediate Wrapped



6.12.293 ALU Floating-Point Comparison Write Read Read Word Pair

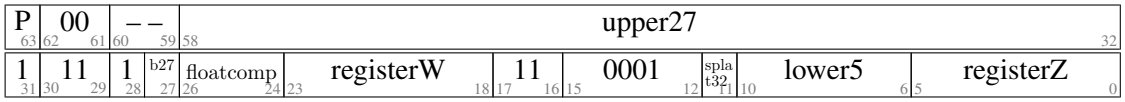


6.12.294 ALU Floating-Point Comparison Write Read Read Word Pair Magic



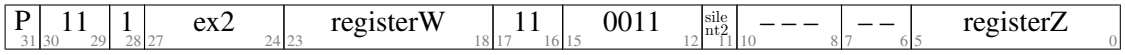
6.12.295 ALU Floating-Point Comparison Write Read Read Half Word Quadruple

6.12.296 ALU Floating-Point Comparison Write Read Read Half Word Quadruple Magic



6.12.297 ALU Double Floating-Point Write Read

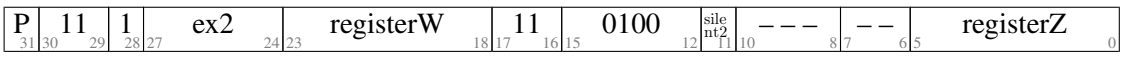
6.12.298 ALU Floating-Point Write Read



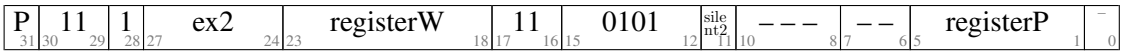
Kalray – all rights reserved

©2025 Kalray

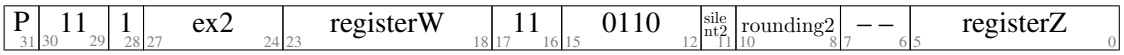
6.12.299 ALU Floating-Point Write Read Half Word Quadruple



6.12.300 ALU Double Floating-Point Write Read Read

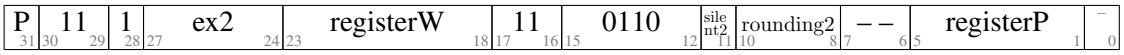


6.12.301 ALU Floating-Point Write Read Word Long Latency



6.12.302 ALU Floating-Point Write Read Word

6.12.303 ALU Floating-Point Write Word Read Word Pair

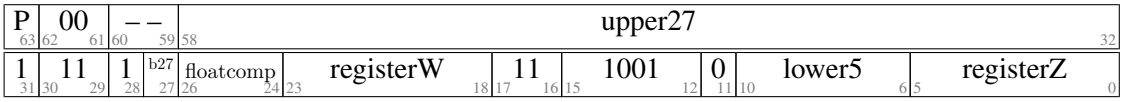


6.12.304 ALU Double Floating-Point Write Read Division

6.12.305 ALU Floating-Point Comparison Negate Write Read Read Scalar

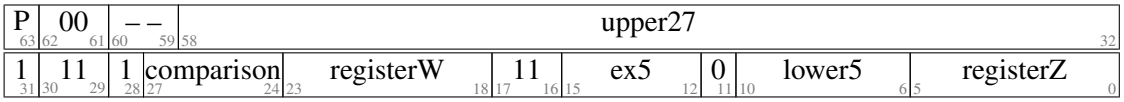
Kalray – all rights reserved

6.12.306 ALU Floating-Point Comparison Negate Write Read Immediate Wrapped



6.12.307 ALU Comparison Negate Write Read Read

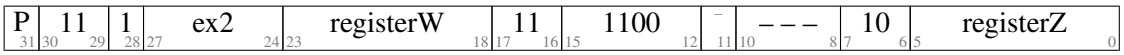
6.12.308 ALU Comparison Negate Write Read Immediate Wrapped



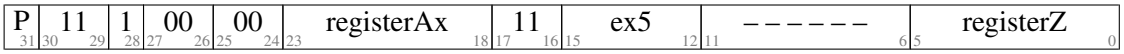
6.12.309 ALU Reduce Write Read Read Word Pair

6.12.310 ALU Word Pair Reduce Write Read Read Half Word Quadruple

6.12.311 ALU Word Pair Reduce Write Read Read Half Word Quadruple



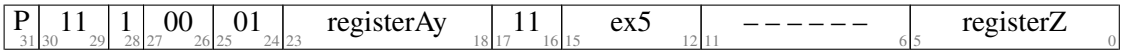
6.12.312 ALU Move To Extension Double Word 0



Kalray – all rights reserved

©2025 Kalray

6.12.313 ALU Move To Extension Double Word 1



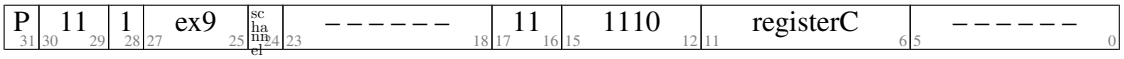
6.12.314 ALU Move To Extension Double Word 2

6.12.315 ALU Move To Extension Double Word 3

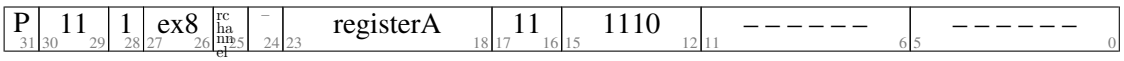
6.12.316 ALU Move To Extension Quadruple Word 0

6.12.317 ALU Move To Extension Quadruple Word 1

6.12.318 ALU Extension Send Octuple Word to Channel

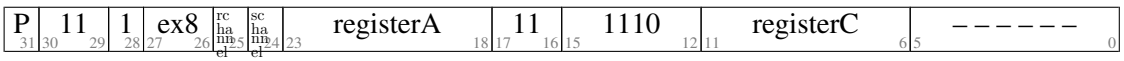


6.12.319 ALU Extension Receive Octuple Word from Channel

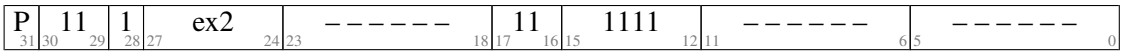


Kalray – all rights reserved

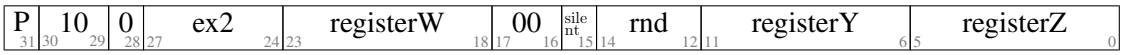
6.12.320 ALU Extension Send and Receive Octuple Word to/from Channels



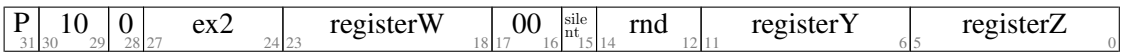
6.12.321 ALU No Operation



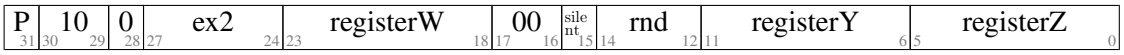
6.12.322 MAU Floating-Point Multiply Add Word



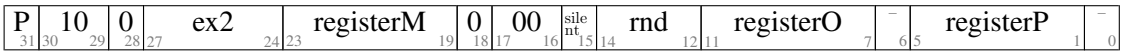
6.12.323 MAU Floating-Point Multiply Add Word Pair



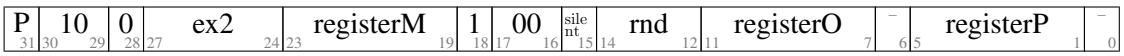
6.12.324 MAU Floating-Point Multiply Add Double



6.12.325 MAU Floating-Point Multiply Add Word Quadruple



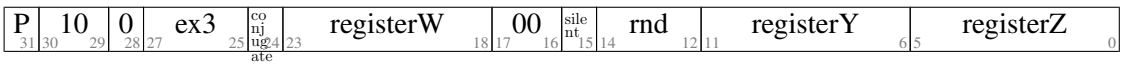
6.12.326 MAU Floating-Point Multiply Add Double Pair



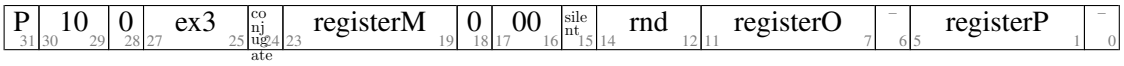
Kalray – all rights reserved

©2025 Kalray

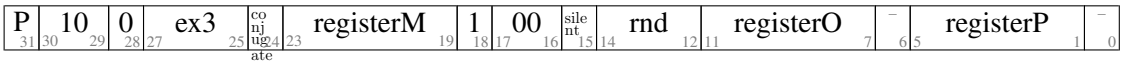
6.12.327 MAU Floating-Point Multiply Complex Word



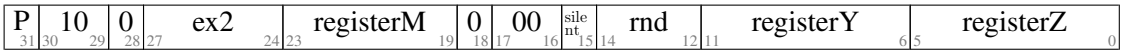
6.12.328 MAU Floating-Point Multiply Word Complex Pair



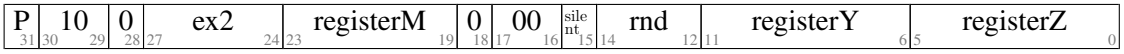
6.12.329 MAU Floating-Point Multiply Complex Double



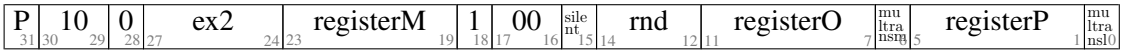
6.12.330 MAU Floating-Point Matrix Multiply Word



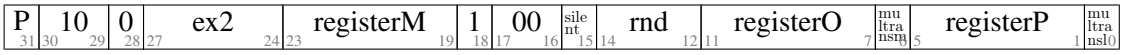
6.12.331 MAU Floating-Point Matrix Multiply Add Word



6.12.332 MAU Floating-Point Matrix Multiply Word Pair

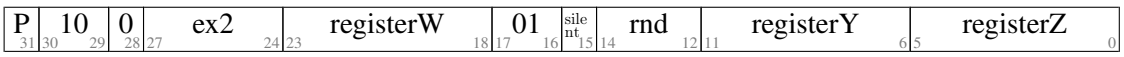


6.12.333 MAU Floating-Point Matrix Multiply Add Word Pair

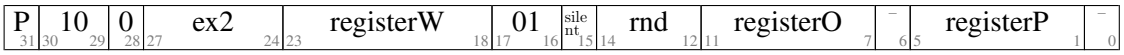


Kalray – all rights reserved

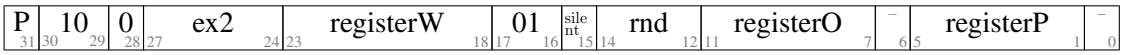
6.12.334 MAU Floating-Point Dual Multiply Add Word



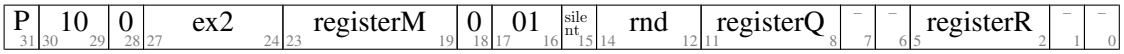
6.12.335 MAU Floating-Point Dual Multiply Add Word Pair



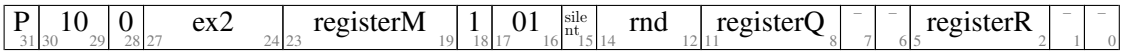
6.12.336 MAU Floating-Point Dual Multiply Add Double



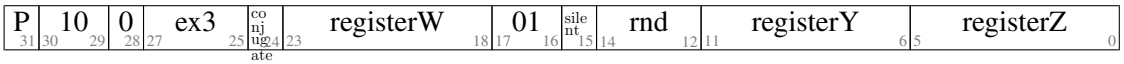
6.12.337 MAU Floating-Point Dual Multiply Add Word Quadruple



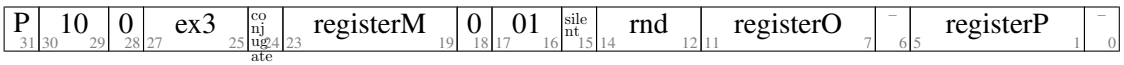
6.12.338 MAU Floating-Point Dual Multiply Add Double Pair



6.12.339 MAU Floating-Point Multiply Add Word Complex



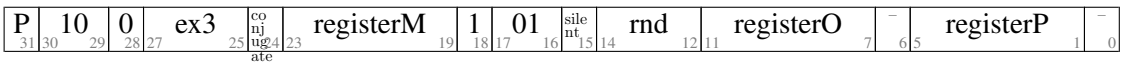
6.12.340 MAU Floating-Point Multiply Add Word Complex Pair



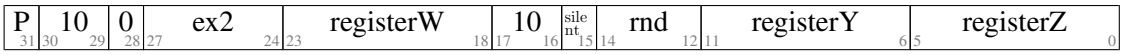
Kalray – all rights reserved

©2025 Kalray

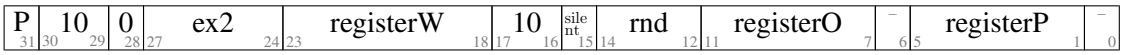
6.12.341 MAU Floating-Point Multiply Add Complex Double



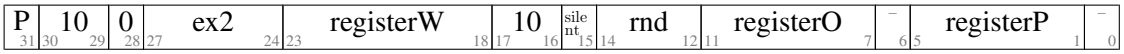
6.12.342 MAU Floating-Point Dual Multiply Dual Add Word



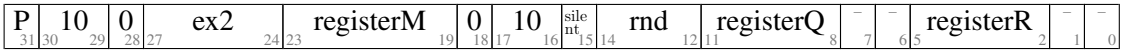
6.12.343 MAU Floating-Point Dual Multiply Dual Add Word Pair



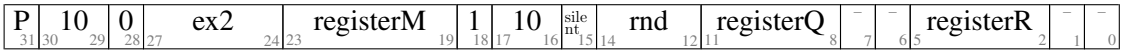
6.12.344 MAU Floating-Point Dual Multiply Dual Add Double



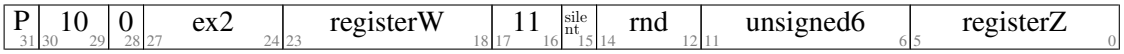
6.12.345 MAU Floating-Point Dual Multiply Dual Add Word Quadruple



6.12.346 MAU Floating-Point Dual Multiply Dual Add Double Pair

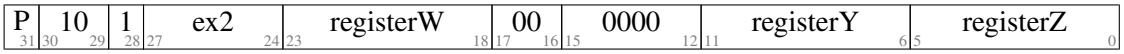


6.12.347 MAU Floating-Point Double Double

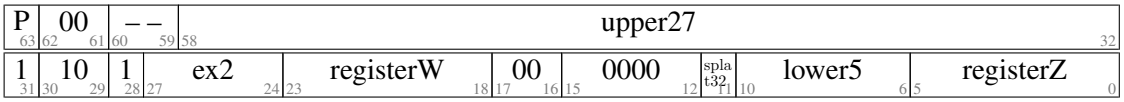


Kalray – all rights reserved

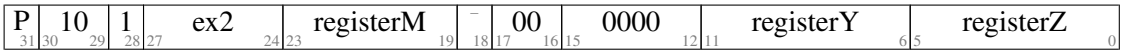
6.12.348 MAU Double Double Double Double 0



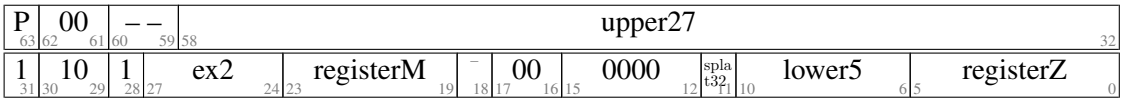
6.12.349 MAU Double Double Double Double 0 Magic



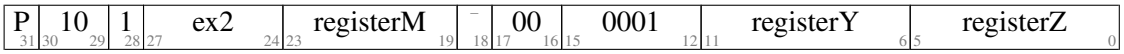
6.12.350 MAU Quadruple Quadruple Double Double



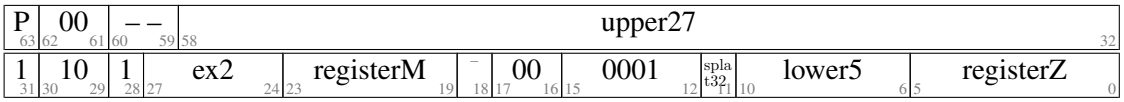
6.12.351 MAU Quadruple Quadruple Double Double Magic



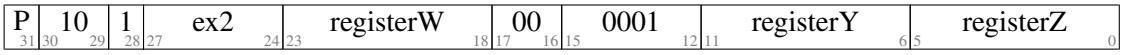
6.12.352 MAU Quadruple Double Double 0



6.12.353 MAU Quadruple Double Double 0 Magic



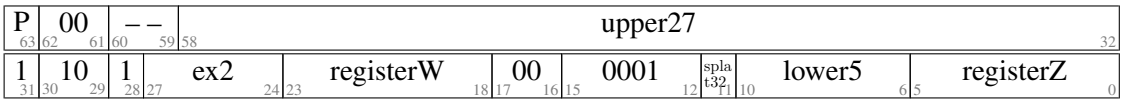
6.12.354 MAU Double Double Double 0



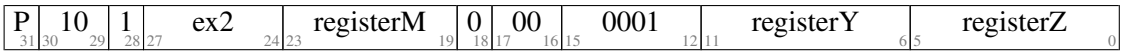
Kalray – all rights reserved

©2025 Kalray

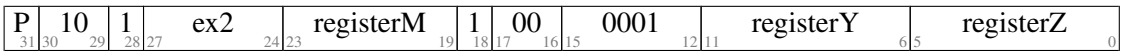
6.12.355 MAU Double Double Double 0 Magic



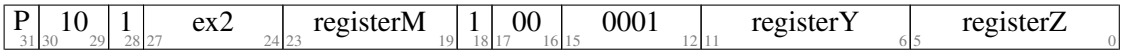
6.12.356 MAU Quadruple Quadruple Double Double 1



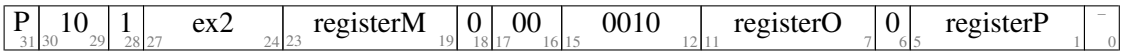
6.12.357 MAU Quadruple Double Double 1



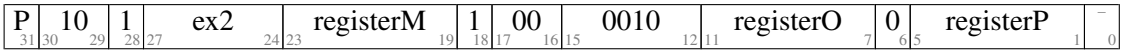
6.12.358 MAU Quadruple Quadruple Double Double 2



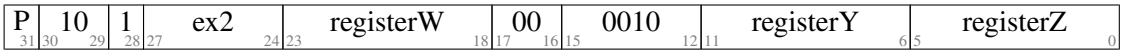
6.12.359 MAU Quadruple Quadruple Quadruple



6.12.360 MAU Quadruple Quadruple Quadruple Quadruple



6.12.361 MAU CRC



Kalray – all rights reserved

6.12.362 MAU CRC Wrapped

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	10	1	ex2						registerW								00		0010			0	lower5					registerZ					
31	30	29	28	27	24	23										18	17	16	15		12	11	10				6	5					0

6.12.363 MAU Double Pair Pair

P	10	1	ex2			registerW							00		0010			registerY						registerZ					
31	30	29	28	27	24	23							18	17	16	15		12	11				6	5					0

6.12.364 MAU Single Single Single Single

P	10	1	ex2			registerW							00		0011			registerY						registerZ					
31	30	29	28	27	24	23							18	17	16	15		12	11				6	5					0

6.12.365 MAU Single Single Single Single Wrapped

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	10	1	ex2						registerW								00		0011			0	lower5					registerZ					
31	30	29	28	27	24	23										18	17	16	15		12	11	10				6	5					0

6.12.366 MAU Single Single Single Single

P	10	1	ex2			registerW							00		0011			registerY						registerZ					
31	30	29	28	27	24	23							18	17	16	15		12	11				6	5					0

6.12.367 MAU Single Single Single Wrapped

P	00	– –	upper27																													
63	62	61	60	59	58																											32
1	10	1	ex2						registerW								00		0011			0	lower5					registerZ					
31	30	29	28	27	24	23										18	17	16	15		12	11	10				6	5					0

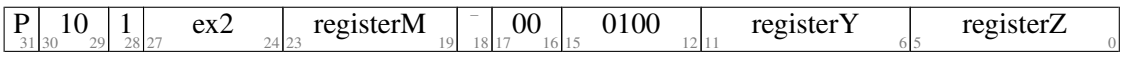
6.12.368 MAU Quadruple Quadruple Double Double 3

P	10	1	ex2			registerM							¯	00		0100			registerY						registerZ					
31	30	29	28	27	24	23						19	18	17	16	15		12	11				6	5					0

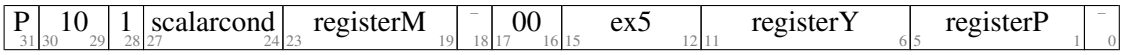
Kalray – all rights reserved

©2025 Kalray

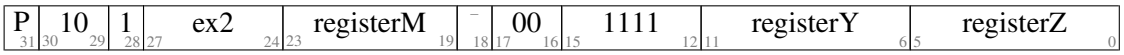
6.12.369 MAU Quadruple Double Double 3



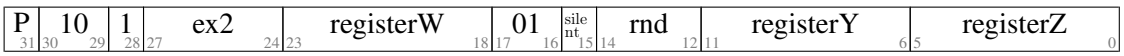
6.12.370 MAU Integer Division and Remainder



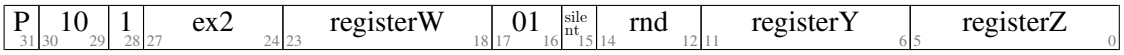
6.12.371 MAU Copy Quadruple Word



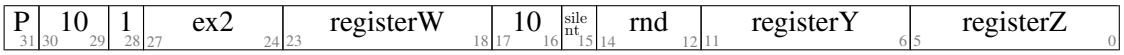
6.12.372 MAU Floating-Point Double Double Double Double



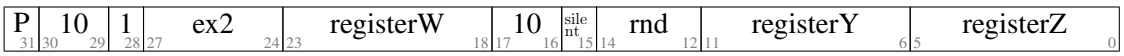
6.12.373 MAU Floating-Point Double Double Double



6.12.374 MAU Floating-Point Double Double Double 1

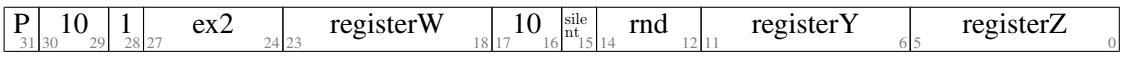


6.12.375 MAU Floating-Point Single Single Single Single

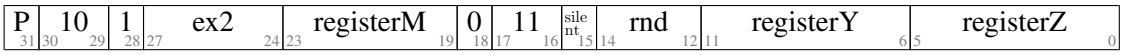


Kalray – all rights reserved

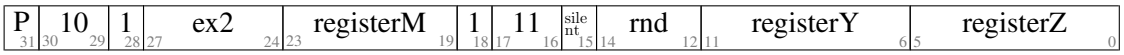
6.12.376 MAU Floating-Point Single Single Single



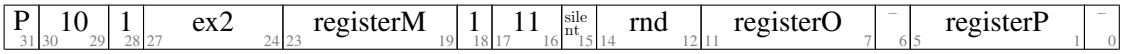
6.12.377 MAU Floating-Point Quadruple Quadruple Double Double



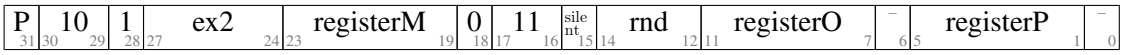
6.12.378 MAU Floating-Point Quadruple Double Double



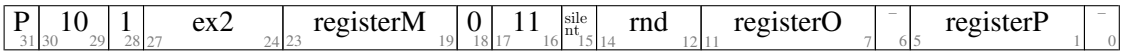
6.12.379 MAU Floating-Point Quadruple Quadruple Quadruple 2



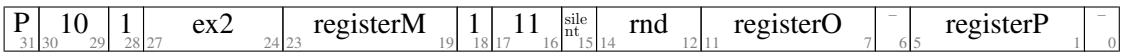
6.12.380 MAU Floating-Point Quadruple Quadruple Quadruple Quadruple



6.12.381 MAU Floating-Point Quadruple Quadruple Quadruple 0



6.12.382 MAU Floating-Point Quadruple Quadruple Quadruple 1



Kalray – all rights reserved

©2025 Kalray

6.13 Instruction Encoding Tables

Note Instructions whose name are suffixed with '*' are those which are not yet described in this document or implemented in the software tools (instruction set simulators, assembler). Moreover, we use the following shortcuts in the table:

Abbreviation	↔	Interpretation
b_7_5	↔	bits_7_5
b_9_8	↔	bits_9_8
comp	↔	comparison
cond	↔	condition
fcomp	↔	floatcomp
N	↔	negate
P	↔	parallel
rnd	↔	rounding
scal	↔	scaling

| | | |

| | | |

©2025 Kalray

| | | |

| | | |

©2025 Kalray

ALU_MAKE	MAKE	P	1	1	0	0	0	0	0	registerW	0	0	signed16						
ALU_DWRI	ABDD	P	1	1	0	0	0	0	1	registerW	0	0	signed10						registerZ
	ADDD	P	1	1	0	0	0	1	0	registerW	0	0	signed10						registerZ
	SBFD	P	1	1	0	0	0	1	1	registerW	0	0	signed10						registerZ
	MIND	P	1	1	0	0	1	0	0	registerW	0	0	signed10						registerZ
	MAXD	P	1	1	0	0	1	0	1	registerW	0	0	signed10						registerZ
	MINUD	P	1	1	0	0	1	1	0	registerW	0	0	signed10						registerZ
	MAXUD	P	1	1	0	0	1	1	1	registerW	0	0	signed10						registerZ
	ANDD	P	1	1	0	1	0	0	0	registerW	0	0	signed10						registerZ
	NANDD	P	1	1	0	1	0	0	1	registerW	0	0	signed10						registerZ
	IORD	P	1	1	0	1	0	1	0	registerW	0	0	signed10						registerZ
	NIORD	P	1	1	0	1	0	1	1	registerW	0	0	signed10						registerZ
	EORD	P	1	1	0	1	1	0	0	registerW	0	0	signed10						registerZ
	NEORD	P	1	1	0	1	1	0	1	registerW	0	0	signed10						registerZ
	ANDND	P	1	1	0	1	1	1	0	registerW	0	0	signed10						registerZ
	IORND	P	1	1	0	1	1	1	1	registerW	0	0	signed10						registerZ
ALU_CWRI	COMPD	P	1	1	0	comp				registerW	0	1	signed10						registerZ
ALU_XSDOI	XSPLATDO	P	1	1	0	1	1	1	0	registerA	0	1	signed16						
ALU_DCMWRI	CMOVED	P	1	1	0	scalarcond				registerW	1	0	signed10						registerZ
ALU_DBMWRI	SBMM8D	P	1	1	0	1	1	1	0	registerW	1	0	signed10						registerZ
	SBMMT8D	P	1	1	0	1	1	1	1	registerW	1	0	signed10						registerZ
ALU_WRB	INSF	P	1	1	0	0	0	stopbit2		registerW	1	1	stopbit4		startbit				registerZ
	EXTFZ	P	1	1	0	0	1	stopbit2		registerW	1	1	stopbit4		startbit				registerZ
	EXTFS	P	1	1	0	1	0	stopbit2		registerW	1	1	stopbit4		startbit				registerZ
	CLRF	P	1	1	0	1	1	stopbit2		registerW	1	1	stopbit4		startbit				registerZ
ALU_PCREL	PCREL	P	1	1	1	0	0	0	0	registerW	0	0	signed16						
ALU_WRIW	ABDW	P	1	1	1	0	0	0	1	registerW	0	0	signed10						registerZ
	ADDW	P	1	1	1	0	0	1	0	registerW	0	0	signed10						registerZ
	SBFW	P	1	1	1	0	0	1	1	registerW	0	0	signed10						registerZ
	MINW	P	1	1	1	0	1	0	0	registerW	0	0	signed10						registerZ
	MAXW	P	1	1	1	0	1	0	1	registerW	0	0	signed10						registerZ
	MINUW	P	1	1	1	0	1	1	0	registerW	0	0	signed10						registerZ
	MAXUW	P	1	1	1	0	1	1	1	registerW	0	0	signed10						registerZ
	ANDW	P	1	1	1	1	0	0	0	registerW	0	0	signed10						registerZ
	NANDW	P	1	1	1	1	0	0	1	registerW	0	0	signed10						registerZ
	IORW	P	1	1	1	1	0	1	0	registerW	0	0	signed10						registerZ
	NIORW	P	1	1	1	1	0	1	1	registerW	0	0	signed10						registerZ
	EORW	P	1	1	1	1	1	0	0	registerW	0	0	signed10						registerZ
	NEORW	P	1	1	1	1	1	0	1	registerW	0	0	signed10						registerZ
	ANDNW	P	1	1	1	1	1	1	0	registerW	0	0	signed10						registerZ
	IORNW	P	1	1	1	1	1	1	1	registerW	0	0	signed10						registerZ
ALU_DWRR0	ABDSD	P	1	1	1	0	0	0	0	registerW	0	1	0	0	0	0	registerY		registerZ
	ABDD	P	1	1	1	0	0	0	1	registerW	0	1	0	0	0	0	registerY		registerZ
	ADDD	P	1	1	1	0	0	1	0	registerW	0	1	0	0	0	0	registerY		registerZ
	SBFD	P	1	1	1	0	0	1	1	registerW	0	1	0	0	0	0	registerY		registerZ
	MIND	P	1	1	1	0	1	0	0	registerW	0	1	0	0	0	0	registerY		registerZ
	MAXD	P	1	1	1	0	1	0	1	registerW	0	1	0	0	0	0	registerY		registerZ
	MINUD	P	1	1	1	0	1	1	0	registerW	0	1	0	0	0	0	registerY		registerZ
	MAXUD	P	1	1	1	0	1	1	1	registerW	0	1	0	0	0	0	registerY		registerZ
	ANDD	P	1	1	1	1	0	0	0	registerW	0	1	0	0	0	0	registerY		registerZ
	NANDD	P	1	1	1	1	0	0	1	registerW	0	1	0	0	0	0	registerY		registerZ
	IORD	P	1	1	1	1	0	1	0	registerW	0	1	0	0	0	0	registerY		registerZ
	NIORD	P	1	1	1	1	0	1	1	registerW	0	1	0	0	0	0	registerY		registerZ
	EORD	P	1	1	1	1	1	0	0	registerW	0	1	0	0	0	0	registerY		registerZ
	NEORD	P	1	1	1	1	1	0	1	registerW	0	1	0	0	0	0	registerY		registerZ
	ANDND	P	1	1	1	1	1	1	0	registerW	0	1	0	0	0	0	registerY		registerZ
	IORND	P	1	1	1	1	1	1	1	registerW	0	1	0	0	0	0	registerY		registerZ
ALU_WWRR0	ABDSW	P	1	1	1	0	0	0	0	registerW	0	1	0	0	0	1	registerY		registerZ
	ABDW	P	1	1	1	0	0	0	1	registerW	0	1	0	0	0	1	registerY		registerZ
	ADDW	P	1	1	1	0	0	1	0	registerW	0	1	0	0	0	1	registerY		registerZ
	SBFW	P	1	1	1	0	0	1	1	registerW	0	1	0	0	0	1	registerY		registerZ
	MINW	P	1	1	1	0	1	0	0	registerW	0	1	0	0	0	1	registerY		registerZ
	MAXW	P	1	1	1	0	1	0	1	registerW	0	1	0	0	0	1	registerY		registerZ
	MINUW	P	1	1	1	0	1	1	0	registerW	0	1	0	0	0	1	registerY		registerZ
	MAXUW	P	1	1	1	0	1	1	1	registerW	0	1	0	0	0	1	registerY		registerZ
	ANDW	P	1	1	1	1	0	0	0	registerW	0	1	0	0	0	1	registerY		registerZ
	NANDW	P	1	1	1	1	0	0	1	registerW	0	1	0	0	0	1	registerY		registerZ
	IORW	P	1	1	1	1	0	1	0	registerW	0	1	0	0	0	1	registerY		registerZ
	NIORW	P	1	1	1	1	0	1	1	registerW	0	1	0	0	0	1	registerY		registerZ
	EORW	P	1	1	1	1	1	0	0	registerW	0	1	0	0	0	1	registerY		registerZ
	NEORW	P	1	1	1	1	1	0	1	registerW	0	1	0	0	0	1	registerY		registerZ
	ANDNW	P	1	1	1	1	1	1	0	registerW	0	1	0	0	0	1	registerY		registerZ
	IORNW	P	1	1	1	1	1	1	1	registerW	0	1	0	0	0	1	registerY		registerZ
ALU_WPWRR0	ABDSWP	P	1	1	1	0	0	0	0	registerW	0	1	0	0	1	0	registerY		registerZ
	ABDWP	P	1	1	1	0	0	0	1	registerW	0	1	0	0	1	0	registerY		registerZ
	ADDWP	P	1	1	1	0	0	1	0	registerW	0	1	0	0	1	0	registerY		registerZ
	SBFWP	P	1	1	1	0	0	1	1	registerW	0	1	0	0	1	0	registerY		registerZ
	MINWP	P	1	1	1	0	1	0	0	registerW	0	1	0	0	1	0	registerY		registerZ
	MAXWP	P	1	1	1	0	1	0	1	registerW	0	1	0	0	1	0	registerY		registerZ
	MINUWP	P	1	1	1	0	1	1	0	registerW	0	1	0	0	1	0	registerY		registerZ
	MAXUWP	P	1	1	1	0	1	1	1	registerW	0	1	0	0	1	0	registerY		registerZ

Kalray – all rights reserved

ALU_HQWRR0	ABDSHQ	P	1	1	1	0	0	0	0	registerW	0	1	0	0	1	1	registerY	registerZ
	ABDHQ	P	1	1	1	0	0	0	1	registerW	0	1	0	0	1	1	registerY	registerZ
	ADDHQ	P	1	1	1	0	0	1	0	registerW	0	1	0	0	1	1	registerY	registerZ
	SBFHQ	P	1	1	1	0	0	1	1	registerW	0	1	0	0	1	1	registerY	registerZ
	MINHQ	P	1	1	1	0	1	0	0	registerW	0	1	0	0	1	1	registerY	registerZ
	MAXHQ	P	1	1	1	0	1	0	1	registerW	0	1	0	0	1	1	registerY	registerZ
	MINUHQ	P	1	1	1	0	1	1	0	registerW	0	1	0	0	1	1	registerY	registerZ
	MAXUHQ	P	1	1	1	0	1	1	1	registerW	0	1	0	0	1	1	registerY	registerZ
ALU_DWRR1	ADDX2D	P	1	1	1	0	0	0	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFX2D	P	1	1	1	0	0	0	1	registerW	0	1	0	1	0	0	registerY	registerZ
	ADDX4D	P	1	1	1	0	0	1	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFX4D	P	1	1	1	0	0	1	1	registerW	0	1	0	1	0	0	registerY	registerZ
	ADDX8D	P	1	1	1	0	1	0	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFX8D	P	1	1	1	0	1	0	1	registerW	0	1	0	1	0	0	registerY	registerZ
	ADDX16D	P	1	1	1	0	1	1	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFX16D	P	1	1	1	0	1	1	1	registerW	0	1	0	1	0	0	registerY	registerZ
	ADDX32D	P	1	1	1	1	0	0	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFX32D	P	1	1	1	1	0	0	1	registerW	0	1	0	1	0	0	registerY	registerZ
	ADDX64D	P	1	1	1	1	0	1	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFX64D	P	1	1	1	1	0	1	1	registerW	0	1	0	1	0	0	registerY	registerZ
	ADDSD	P	1	1	1	1	1	0	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFSD	P	1	1	1	1	1	0	1	registerW	0	1	0	1	0	0	registerY	registerZ
	ADDUSD	P	1	1	1	1	1	1	0	registerW	0	1	0	1	0	0	registerY	registerZ
	SBFUSD	P	1	1	1	1	1	1	1	registerW	0	1	0	1	0	0	registerY	registerZ
ALU_WWRR1	ADDX2W	P	1	1	1	0	0	0	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFX2W	P	1	1	1	0	0	0	1	registerW	0	1	0	1	0	1	registerY	registerZ
	ADDX4W	P	1	1	1	0	0	1	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFX4W	P	1	1	1	0	0	1	1	registerW	0	1	0	1	0	1	registerY	registerZ
	ADDX8W	P	1	1	1	0	1	0	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFX8W	P	1	1	1	0	1	0	1	registerW	0	1	0	1	0	1	registerY	registerZ
	ADDX16W	P	1	1	1	0	1	1	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFX16W	P	1	1	1	0	1	1	1	registerW	0	1	0	1	0	1	registerY	registerZ
	ADDX32W	P	1	1	1	1	0	0	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFX32W	P	1	1	1	1	0	0	1	registerW	0	1	0	1	0	1	registerY	registerZ
	ADDX64W	P	1	1	1	1	0	1	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFX64W	P	1	1	1	1	0	1	1	registerW	0	1	0	1	0	1	registerY	registerZ
	ADDSW	P	1	1	1	1	1	0	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFSW	P	1	1	1	1	1	0	1	registerW	0	1	0	1	0	1	registerY	registerZ
	ADDUSW	P	1	1	1	1	1	1	0	registerW	0	1	0	1	0	1	registerY	registerZ
	SBFUSW	P	1	1	1	1	1	1	1	registerW	0	1	0	1	0	1	registerY	registerZ
ALU_WPWRR1	ADDX2WP	P	1	1	1	0	0	0	0	registerW	0	1	0	1	1	0	registerY	registerZ
	SBFX2WP	P	1	1	1	0	0	0	1	registerW	0	1	0	1	1	0	registerY	registerZ
	ADDX4WP	P	1	1	1	0	0	1	0	registerW	0	1	0	1	1	0	registerY	registerZ
	SBFX4WP	P	1	1	1	0	0	1	1	registerW	0	1	0	1	1	0	registerY	registerZ
	ADDX8WP	P	1	1	1	0	1	0	0	registerW	0	1	0	1	1	0	registerY	registerZ
	SBFX8WP	P	1	1	1	0	1	0	1	registerW	0	1	0	1	1	0	registerY	registerZ
	ADDX16WP	P	1	1	1	0	1	1	0	registerW	0	1	0	1	1	0	registerY	registerZ
	SBFX16WP	P	1	1	1	0	1	1	1	registerW	0	1	0	1	1	0	registerY	registerZ
	ADDSWP	P	1	1	1	1	1	0	0	registerW	0	1	0	1	1	0	registerY	registerZ
	SBFSWP	P	1	1	1	1	1	0	1	registerW	0	1	0	1	1	0	registerY	registerZ
	ADDUSWP	P	1	1	1	1	1	1	0	registerW	0	1	0	1	1	0	registerY	registerZ
	SBFUSWP	P	1	1	1	1	1	1	1	registerW	0	1	0	1	1	0	registerY	registerZ
ALU_HQWRR1	ADDX2HQ	P	1	1	1	0	0	0	0	registerW	0	1	0	1	1	1	registerY	registerZ
	SBFX2HQ	P	1	1	1	0	0	0	1	registerW	0	1	0	1	1	1	registerY	registerZ
	ADDX4HQ	P	1	1	1	0	0	1	0	registerW	0	1	0	1	1	1	registerY	registerZ
	SBFX4HQ	P	1	1	1	0	0	1	1	registerW	0	1	0	1	1	1	registerY	registerZ
	ADDX8HQ	P	1	1	1	0	1	0	0	registerW	0	1	0	1	1	1	registerY	registerZ
	SBFX8HQ	P	1	1	1	0	1	0	1	registerW	0	1	0	1	1	1	registerY	registerZ
	ADDX16HQ	P	1	1	1	0	1	1	0	registerW	0	1	0	1	1	1	registerY	registerZ
	SBFX16HQ	P	1	1	1	0	1	1	1	registerW	0	1	0	1	1	1	registerY	registerZ
	ADDSHQ	P	1	1	1	1	1	0	0	registerW	0	1	0	1	1	1	registerY	registerZ
	SBFSHQ	P	1	1	1	1	1	0	1	registerW	0	1	0	1	1	1	registerY	registerZ
	ADDUSHQ	P	1	1	1	1	1	1	0	registerW	0	1	0	1	1	1	registerY	registerZ
	SBFUSHQ	P	1	1	1	1	1	1	1	registerW	0	1	0	1	1	1	registerY	registerZ
ALU_WDWRR2	ADDX2WD	P	1	1	1	0	0	0	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFX2WD	P	1	1	1	0	0	0	1	registerW	0	1	1	0	0	0	registerY	registerZ
	ADDX4WD	P	1	1	1	0	0	1	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFX4WD	P	1	1	1	0	0	1	1	registerW	0	1	1	0	0	0	registerY	registerZ
	ADDX8WD	P	1	1	1	0	1	0	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFX8WD	P	1	1	1	0	1	0	1	registerW	0	1	1	0	0	0	registerY	registerZ
	ADDX16WD	P	1	1	1	0	1	1	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFX16WD	P	1	1	1	0	1	1	1	registerW	0	1	1	0	0	0	registerY	registerZ
	ADDX32WD	P	1	1	1	1	0	0	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFX32WD	P	1	1	1	1	0	0	1	registerW	0	1	1	0	0	0	registerY	registerZ
	ADDX64WD	P	1	1	1	1	0	1	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFX64WD	P	1	1	1	1	0	1	1	registerW	0	1	1	0	0	0	registerY	registerZ
	ADDWD	P	1	1	1	1	1	0	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFWD	P	1	1	1	1	1	0	1	registerW	0	1	1	0	0	0	registerY	registerZ
	ADDCD	P	1	1	1	1	1	1	0	registerW	0	1	1	0	0	0	registerY	registerZ
	SBFCD	P	1	1	1	1	1	1	1	registerW	0	1	1	0	0	0	registerY	registerZ
ALU_WDWRR3	ADDX2UWD	P	1	1	1	0	0	0	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFX2UWD	P	1	1	1	0	0	0	1	registerW	0	1	1	0	0	1	registerY	registerZ
	ADDX4UWD	P	1	1	1	0	0	1	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFX4UWD	P	1	1	1	0	0	1	1	registerW	0	1	1	0	0	1	registerY	registerZ
	ADDX8UWD	P	1	1	1	0	1	0	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFX8UWD	P	1	1	1	0	1	0	1	registerW	0	1	1	0	0	1	registerY	registerZ
	ADDX16UWD	P	1	1	1	0	1	1	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFX16UWD	P	1	1	1	0	1	1	1	registerW	0	1	1	0	0	1	registerY	registerZ
	ADDX32UWD	P	1	1	1	1	0	0	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFX32UWD	P	1	1	1	1	0	0	1	registerW	0	1	1	0	0	1	registerY	registerZ
	ADDX64UWD	P	1	1	1	1	0	1	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFX64UWD	P	1	1	1	1	0	1	1	registerW	0	1	1	0	0	1	registerY	registerZ
	ADDUWD	P	1	1	1	1	1	0	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFUWD	P	1	1	1	1	1	0	1	registerW	0	1	1	0	0	1	registerY	registerZ
	ADDCD.I	P	1	1	1	1	1	1	0	registerW	0	1	1	0	0	1	registerY	registerZ
	SBFCD.I	P	1	1	1	1	1	1	1	registerW	0	1	1	0	0	1	registerY	registerZ

Kalray – all rights reserved

©2025 Kalray

| | | |

ALU_WSWRR	LANDW	P	1	1	1	0	0	0	0	registerW	1	0	0	1	1	1	registerY	registerZ
	LNANDW	P	1	1	1	0	0	0	1	registerW	1	0	0	1	1	1	registerY	registerZ
	LIORW	P	1	1	1	0	0	1	0	registerW	1	0	0	1	1	1	registerY	registerZ
	LNIORW	P	1	1	1	0	0	1	1	registerW	1	0	0	1	1	1	registerY	registerZ
	AVGW	P	1	1	1	0	1	0	0	registerW	1	0	0	1	1	1	registerY	registerZ
	AVGRW	P	1	1	1	0	1	0	1	registerW	1	0	0	1	1	1	registerY	registerZ
	AVGUW	P	1	1	1	0	1	1	0	registerW	1	0	0	1	1	1	registerY	registerZ
	AVGRUW	P	1	1	1	0	1	1	1	registerW	1	0	0	1	1	1	registerY	registerZ
	SRSW	P	1	1	1	1	0	0	0	registerW	1	0	0	1	1	1	registerY	registerZ
	SLLW	P	1	1	1	1	0	0	1	registerW	1	0	0	1	1	1	registerY	registerZ
	SRAW	P	1	1	1	1	0	1	0	registerW	1	0	0	1	1	1	registerY	registerZ
	SRLW	P	1	1	1	1	0	1	1	registerW	1	0	0	1	1	1	registerY	registerZ
	SLSW	P	1	1	1	1	1	0	0	registerW	1	0	0	1	1	1	registerY	registerZ
	SLUSW	P	1	1	1	1	1	0	1	registerW	1	0	0	1	1	1	registerY	registerZ
	ROLW	P	1	1	1	1	1	1	0	registerW	1	0	0	1	1	1	registerY	registerZ
	RORW	P	1	1	1	1	1	1	1	registerW	1	0	0	1	1	1	registerY	registerZ
ALU_WPSWRR	AVGWP	P	1	1	1	0	1	0	0	registerW	1	0	1	0	0	0	registerY	registerZ
	AVGRWP	P	1	1	1	0	1	0	1	registerW	1	0	1	0	0	0	registerY	registerZ
	AVGUWP	P	1	1	1	0	1	1	0	registerW	1	0	1	0	0	0	registerY	registerZ
	AVGRUWP	P	1	1	1	0	1	1	1	registerW	1	0	1	0	0	0	registerY	registerZ
	SRSWPS	P	1	1	1	1	0	0	0	registerW	1	0	1	0	0	0	registerY	registerZ
	SLLWPS	P	1	1	1	1	0	0	1	registerW	1	0	1	0	0	0	registerY	registerZ
	SRAWPS	P	1	1	1	1	0	1	0	registerW	1	0	1	0	0	0	registerY	registerZ
	SRLWPS	P	1	1	1	1	0	1	1	registerW	1	0	1	0	0	0	registerY	registerZ
	SLSWPS	P	1	1	1	1	1	0	0	registerW	1	0	1	0	0	0	registerY	registerZ
	SLUSWPS	P	1	1	1	1	1	0	1	registerW	1	0	1	0	0	0	registerY	registerZ
	ROLWPS	P	1	1	1	1	1	1	0	registerW	1	0	1	0	0	0	registerY	registerZ
	RORWPS	P	1	1	1	1	1	1	1	registerW	1	0	1	0	0	0	registerY	registerZ
ALU_HQSWRR	AVGHQ	P	1	1	1	0	1	0	0	registerW	1	0	1	0	0	1	registerY	registerZ
	AVGRHQ	P	1	1	1	0	1	0	1	registerW	1	0	1	0	0	1	registerY	registerZ
	AVGUHQ	P	1	1	1	0	1	1	0	registerW	1	0	1	0	0	1	registerY	registerZ
	AVGRUHQ	P	1	1	1	0	1	1	1	registerW	1	0	1	0	0	1	registerY	registerZ
	SRSHQS	P	1	1	1	1	0	0	0	registerW	1	0	1	0	0	1	registerY	registerZ
	SLLHQS	P	1	1	1	1	0	0	1	registerW	1	0	1	0	0	1	registerY	registerZ
	SRAHQS	P	1	1	1	1	0	1	0	registerW	1	0	1	0	0	1	registerY	registerZ
	SRLHQS	P	1	1	1	1	0	1	1	registerW	1	0	1	0	0	1	registerY	registerZ
	SLSHQS	P	1	1	1	1	1	0	0	registerW	1	0	1	0	0	1	registerY	registerZ
	SLUSHQS	P	1	1	1	1	1	0	1	registerW	1	0	1	0	0	1	registerY	registerZ
ALU_BOWRR0	ADDBO	P	1	1	1	0	0	0	0	registerW	1	0	1	0	1	0	registerY	registerZ
	SBFBO	P	1	1	1	0	0	0	1	registerW	1	0	1	0	1	0	registerY	registerZ
	ABDBO	P	1	1	1	0	0	1	0	registerW	1	0	1	0	1	0	registerY	registerZ
	ABDSBO	P	1	1	1	0	0	1	1	registerW	1	0	1	0	1	0	registerY	registerZ
	MINBO	P	1	1	1	0	1	0	0	registerW	1	0	1	0	1	0	registerY	registerZ
	MAXBO	P	1	1	1	0	1	0	1	registerW	1	0	1	0	1	0	registerY	registerZ
	MINUBO	P	1	1	1	0	1	1	0	registerW	1	0	1	0	1	0	registerY	registerZ
	MAXUBO	P	1	1	1	0	1	1	1	registerW	1	0	1	0	1	0	registerY	registerZ
ALU_BOWRR1	ADDX2BO	P	1	1	1	0	0	0	0	registerW	1	0	1	0	1	1	registerY	registerZ
	SBFX2BO	P	1	1	1	0	0	0	1	registerW	1	0	1	0	1	1	registerY	registerZ
	ADDX4BO	P	1	1	1	0	0	1	0	registerW	1	0	1	0	1	1	registerY	registerZ
	SBFX4BO	P	1	1	1	0	0	1	1	registerW	1	0	1	0	1	1	registerY	registerZ
	ADDX8BO	P	1	1	1	0	1	0	0	registerW	1	0	1	0	1	1	registerY	registerZ
	SBFX8BO	P	1	1	1	0	1	0	1	registerW	1	0	1	0	1	1	registerY	registerZ
	ADDX16BO	P	1	1	1	0	1	1	0	registerW	1	0	1	0	1	1	registerY	registerZ
	SBFX16BO	P	1	1	1	0	1	1	1	registerW	1	0	1	0	1	1	registerY	registerZ
	ADDSBO	P	1	1	1	1	1	0	0	registerW	1	0	1	0	1	1	registerY	registerZ
	SBFSBO	P	1	1	1	1	1	0	1	registerW	1	0	1	0	1	1	registerY	registerZ
	ADDUSBO	P	1	1	1	1	1	1	0	registerW	1	0	1	0	1	1	registerY	registerZ
	SBFUSBO	P	1	1	1	1	1	1	1	registerW	1	0	1	0	1	1	registerY	registerZ
ALU_BOCNWRR	COMPNBO	P	1	1	1	comp				registerW	1	0	1	1	0	0	registerY	registerZ
ALU_BOWRRS	ABDUBO	P	1	1	1	1	1	1	1	registerW	1	0	1	1	0	0	registerY	registerZ
ALU_BOCMWRR	CMOVEBO	P	1	1	1	0	simdcond			registerW	1	0	1	1	0	1	registerY	registerZ
ALU_BOSWRI	SRSBOS	P	1	1	1	1	0	0	0	registerW	1	0	1	1	1	0	unsigned6	registerZ
	SLLBOS	P	1	1	1	1	0	0	1	registerW	1	0	1	1	1	0	unsigned6	registerZ
	SRABOS	P	1	1	1	1	0	1	0	registerW	1	0	1	1	1	0	unsigned6	registerZ
	SRLBOS	P	1	1	1	1	0	1	1	registerW	1	0	1	1	1	0	unsigned6	registerZ
	SLSBOS	P	1	1	1	1	1	0	0	registerW	1	0	1	1	1	0	unsigned6	registerZ
	SLUSBOS	P	1	1	1	1	1	0	1	registerW	1	0	1	1	1	0	unsigned6	registerZ
ALU_BOSWRR	AVGBO	P	1	1	1	0	1	0	0	registerW	1	0	1	1	1	1	registerY	registerZ
	AVGRBO	P	1	1	1	0	1	0	1	registerW	1	0	1	1	1	1	registerY	registerZ
	AVGUBO	P	1	1	1	0	1	1	0	registerW	1	0	1	1	1	1	registerY	registerZ
	AVGRUBO	P	1	1	1	0	1	1	1	registerW	1	0	1	1	1	1	registerY	registerZ
	SRSBOS	P	1	1	1	1	0	0	0	registerW	1	0	1	1	1	1	registerY	registerZ
	SLLBOS	P	1	1	1	1	0	0	1	registerW	1	0	1	1	1	1	registerY	registerZ
	SRABOS	P	1	1	1	1	0	1	0	registerW	1	0	1	1	1	1	registerY	registerZ
	SRLBOS	P	1	1	1	1	0	1	1	registerW	1	0	1	1	1	1	registerY	registerZ
	SLSBOS	P	1	1	1	1	1	0	0	registerW	1	0	1	1	1	1	registerY	registerZ
	SLUSBOS	P	1	1	1	1	1	0	1	registerW	1	0	1	1	1	1	registerY	registerZ
ALU_FCWRRS	FCOMPW	P	1	1	1	0	fcomp			registerW	1	1	0	0	0	0	registerY	registerZ
	FCOMPD	P	1	1	1	1	fcomp			registerW	1	1	0	0	0	0	registerY	registerZ
ALU_WPFCWRR	FCOMPNWP	P	1	1	1	0	fcomp			registerW	1	1	0	0	0	1	registerY	registerZ
ALU_HQFCWRR	FCOMPNHQ	P	1	1	1	1	fcomp			registerW	1	1	0	0	0	1	registerY	registerZ

Kalray – all rights reserved

©2025 Kalray

ALU_DFWR	FNEGD	P	1	1	1	0	0	0	0	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FABSD	P	1	1	1	0	0	0	1	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FNEGW	P	1	1	1	0	0	1	0	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FABSW	P	1	1	1	0	0	1	1	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FNEGWP	P	1	1	1	0	1	0	0	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FABSWP	P	1	1	1	0	1	0	1	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FNEGHQ	P	1	1	1	0	1	1	0	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FABSHQ	P	1	1	1	0	1	1	1	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FSRSRD	P	1	1	1	1	0	0	0	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FSRSRW	P	1	1	1	1	0	1	0	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
	FSRSRWP	P	1	1	1	1	1	0	0	registerW	1	1	0	0	1	0	–	–	–	–	–	registerZ							
ALU_FWR	FWIDENLWD	P	1	1	1	1	0	0	0	registerW	1	1	0	0	1	1	silent2	–	–	–	–	registerZ							
	FWIDENMWD	P	1	1	1	1	0	0	1	registerW	1	1	0	0	1	1	silent2	–	–	–	–	registerZ							
	FWIDENLHW	P	1	1	1	1	0	1	0	registerW	1	1	0	0	1	1	silent2	–	–	–	–	registerZ							
	FWIDENMHW	P	1	1	1	1	0	1	1	registerW	1	1	0	0	1	1	silent2	–	–	–	–	registerZ							
	FWIDENLHWP	P	1	1	1	1	1	0	0	registerW	1	1	0	0	1	1	silent2	–	–	–	–	registerZ							
	FWIDENMHWP	P	1	1	1	1	1	0	1	registerW	1	1	0	0	1	1	silent2	–	–	–	–	registerZ							
ALU_HQFWR	FSRECD	P	1	1	1	0	0	0	0	registerW	1	1	0	1	0	0	silent2	–	–	–	–	registerZ							
	FSRECW	P	1	1	1	0	0	1	0	registerW	1	1	0	1	0	0	silent2	–	–	–	–	registerZ							
	FSRECWP	P	1	1	1	0	1	0	0	registerW	1	1	0	1	0	0	silent2	–	–	–	–	registerZ							
ALU_DFWRR	FSDIVD	P	1	1	1	0	0	0	0	registerW	1	1	0	1	0	1	silent2	–	–	–	–	registerP	–						
	FCDIVD	P	1	1	1	0	0	0	1	registerW	1	1	0	1	0	1	silent2	–	–	–	–	registerP	–						
	FSDIVW	P	1	1	1	0	0	1	0	registerW	1	1	0	1	0	1	silent2	–	–	–	–	registerP	–						
	FCDIVW	P	1	1	1	0	0	1	1	registerW	1	1	0	1	0	1	silent2	–	–	–	–	registerP	–						
	FSDIVWP	P	1	1	1	0	1	0	0	registerW	1	1	0	1	0	1	silent2	–	–	–	–	registerP	–						
	FCDIVWP	P	1	1	1	0	1	0	1	registerW	1	1	0	1	0	1	silent2	–	–	–	–	registerP	–						
ALU_FWRWLL	FRECW	P	1	1	1	0	0	1	0	registerW	1	1	0	1	1	0	silent2rnd2			–	–	registerZ							
	FRSRW	P	1	1	1	0	0	1	1	registerW	1	1	0	1	1	0	silent2rnd2			–	–	registerZ							
ALU_FWRW	FNARROWDW	P	1	1	1	1	0	0	0	registerW	1	1	0	1	1	0	silent2rnd2			–	–	registerZ							
	FNARROWWH	P	1	1	1	1	0	1	0	registerW	1	1	0	1	1	0	silent2rnd2			–	–	registerZ							
ALU_WPFWR	FNARROWDWP	P	1	1	1	1	1	0	0	registerW	1	1	0	1	1	0	silent2rnd2			–	–	registerP	–						
	FNARROWWHP	P	1	1	1	1	1	1	0	registerW	1	1	0	1	1	0	silent2rnd2			–	–	registerP	–						
ALU_DFWRD	FMIND	P	1	1	1	0	0	0	0	registerW	1	1	1	0	0	0	registerY					registerZ							
	FMAXD	P	1	1	1	0	0	0	1	registerW	1	1	1	0	0	0	registerY					registerZ							
	FMINW	P	1	1	1	0	0	1	0	registerW	1	1	1	0	0	0	registerY					registerZ							
	FMAXW	P	1	1	1	0	0	1	1	registerW	1	1	1	0	0	0	registerY					registerZ							
	FMINWP	P	1	1	1	0	1	0	0	registerW	1	1	1	0	0	0	registerY					registerZ							
	FMAXWP	P	1	1	1	0	1	0	1	registerW	1	1	1	0	0	0	registerY					registerZ							
	FMINHQ	P	1	1	1	0	1	1	0	registerW	1	1	1	0	0	0	registerY					registerZ							
	FMAXHQ	P	1	1	1	0	1	1	1	registerW	1	1	1	0	0	0	registerY					registerZ							
ALU_FCNWRRS	FCOMPNW	P	1	1	1	0	fcomp			registerW	1	1	1	0	0	1	registerY					registerZ							
	FCOMPND	P	1	1	1	1	fcomp			registerW	1	1	1	0	0	1	registerY					registerZ							
ALU_CNWRR	COMPNW	P	1	1	1	comp				registerW	1	1	1	0	1	0	registerY					registerZ							
	COMPND	P	1	1	1	comp				registerW	1	1	1	0	1	1	registerY					registerZ							
ALU_WPRWRR	ANDRWPD	P	1	1	1	0	0	0	0	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	IORRWPD	P	1	1	1	0	0	0	1	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	EORRWPD	P	1	1	1	0	0	1	0	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	MINRWPD	P	1	1	1	0	1	0	0	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	MAXRWPD	P	1	1	1	0	1	0	1	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	ADDRWPD	P	1	1	1	0	1	1	0	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	MINURWPD	P	1	1	1	1	0	0	0	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	MAXURWPD	P	1	1	1	1	0	0	1	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
	ADDURWPD	P	1	1	1	1	0	1	0	registerW	1	1	1	1	0	0	–	–	–	–	0	0	registerZ						
ALU_HQRWRR	ANDRHQD	P	1	1	1	0	0	0	0	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	IORRHQD	P	1	1	1	0	0	0	1	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	EORRHQD	P	1	1	1	0	0	1	0	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	MINRHQD	P	1	1	1	0	1	0	0	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	MAXRHQD	P	1	1	1	0	1	0	1	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	ADDRHQD	P	1	1	1	0	1	1	0	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	MINURHQD	P	1	1	1	1	0	0	0	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	MAXURHQD	P	1	1	1	1	0	0	1	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
	ADDURHQD	P	1	1	1	1	0	1	0	registerW	1	1	1	1	0	0	–	–	–	–	0	1	registerZ						
ALU_BORWRR	ANDRBOD	P	1	1	1	0	0	0	0	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	IORRBOD	P	1	1	1	0	0	0	1	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	EORRBOD	P	1	1	1	0	0	1	0	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	MINRBOD	P	1	1	1	0	1	0	0	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	MAXRBOD	P	1	1	1	0	1	0	1	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	ADDRBOD	P	1	1	1	0	1	1	0	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	MINURBOD	P	1	1	1	1	0	0	0	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	MAXURBOD	P	1	1	1	1	0	0	1	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
	ADDURBOD	P	1	1	1	1	0	1	0	registerW	1	1	1	1	0	0	–	–	–	–	1	0	registerZ						
ALU_XMTD0	XMOVETD	P	1	1	1	0	0	0	0	registerAx	1	1	1	1	1	0	–	–	–	–	–	–	registerZ						
ALU_XMTD1	XMOVETD	P	1	1	1	0	0	0	1	registerAy	1	1	1	1	1	0	–	–	–	–	–	–	registerZ						
ALU_XMTD2	XMOVETD	P	1	1	1	0	0	1	0	registerAz	1	1	1	1	1	0	–	–	–	–	–	–	registerZ						
ALU_XMTD3	XMOVETD	P	1	1	1	0	0	1	1	registerAt	1	1	1	1	1	0	–	–	–	–	–	–	registerZ						
ALU_XMTQ0	XMOVETQ	P	1	1	1	0	1	0	0	registerAE	1	1	1	1	1	0	registerY					registerZ							
ALU_XMTQ1	XMOVETQ	P	1	1	1	0	1	0	1	registerAO	1	1	1	1	1	0	registerY					registerZ							
ALU_CSC	XSENDO	P	1	1	1	0	1	1	schannel	–	–	–	–	–	1	1	1	1	1	0	registerC					–	–	–	–	–
ALU_CRC	XRECVO	P	1	1	1	1	0	rchannel		registerA	1	1	1	1	1	0	–	–	–	–	–	–	–	–	–	–	–	–	
ALU_CSRC	XSENDRECVO	P	1	1	1	1	1	rchannel	schannel	registerA	1	1	1	1	1	0	registerC					–	–	–	–	–			
ALU_NOP	NOP	P	1	1	1	1	1	1	1	–	–	–	–	–	1	1	1	1	1	1	–	–	–	–	–	–	–	–	–

Kalray – all rights reserved

| | | |

©2025 Kalray

MAU_QDD0	MULWDP	P	1	0	1	0	0	0	0	registerM	–	0	0	0	0	0	1	registerY		registerZ	
	MULUWDP	P	1	0	1	0	0	0	1	registerM	–	0	0	0	0	0	1	registerY		registerZ	
	MULSUWDP	P	1	0	1	0	0	1	0	registerM	–	0	0	0	0	0	1	registerY		registerZ	
	MM212W	P	1	0	1	0	0	1	1	registerM	–	0	0	0	0	0	1	registerY		registerZ	
MAU_DDD0	MULD	P	1	0	1	0	1	0	0	registerW		0	0	0	0	0	1	registerY		registerZ	
	MULWP	P	1	0	1	0	1	0	1	registerW		0	0	0	0	0	1	registerY		registerZ	
	MULHQ	P	1	0	1	0	1	1	0	registerW		0	0	0	0	0	1	registerY		registerZ	
MAU_QQDD1	MADDDT	P	1	0	1	1	0	0	0	registerM	0	0	0	0	0	0	1	registerY		registerZ	
	MADDUDT	P	1	0	1	1	0	0	1	registerM	0	0	0	0	0	0	1	registerY		registerZ	
	MADDSUDT	P	1	0	1	1	0	1	0	registerM	0	0	0	0	0	0	1	registerY		registerZ	
	MADDUZDT	P	1	0	1	1	0	1	1	registerM	0	0	0	0	0	0	1	registerY		registerZ	
	MSBFDT	P	1	0	1	1	1	0	0	registerM	0	0	0	0	0	0	1	registerY		registerZ	
	MSBFUDT	P	1	0	1	1	1	0	1	registerM	0	0	0	0	0	0	1	registerY		registerZ	
	MSBFSUDT	P	1	0	1	1	1	1	0	registerM	0	0	0	0	0	0	1	registerY		registerZ	
	MSBFUZDT	P	1	0	1	1	1	1	1	registerM	0	0	0	0	0	0	1	registerY		registerZ	
MAU_QDD1	MULDT	P	1	0	1	1	0	0	0	registerM	1	0	0	0	0	0	1	registerY		registerZ	
	MULUDT	P	1	0	1	1	0	0	1	registerM	1	0	0	0	0	0	1	registerY		registerZ	
	MULSUDT	P	1	0	1	1	0	1	0	registerM	1	0	0	0	0	0	1	registerY		registerZ	
	CMULDT	P	1	0	1	1	0	1	1	registerM	1	0	0	0	0	0	1	registerY		registerZ	
MAU_QQDD2	CMULXDT	P	1	0	1	1	1	0	0	registerM	1	0	0	0	0	0	1	registerY		registerZ	
	CMULGLXDT	P	1	0	1	1	1	0	1	registerM	1	0	0	0	0	0	1	registerY		registerZ	
	CMULGMXDT	P	1	0	1	1	1	1	0	registerM	1	0	0	0	0	0	1	registerY		registerZ	
	CMULGHXDT	P	1	0	1	1	1	1	1	registerM	1	0	0	0	0	0	1	registerY		registerZ	
MAU_QQQ	DOT2WDP	P	1	0	1	0	0	0	0	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
	DOT2UWDP	P	1	0	1	0	0	0	1	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
	DOT2SUWDP	P	1	0	1	0	0	1	0	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
	DOT2WZP	P	1	0	1	0	0	1	1	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
	MULWQ	P	1	0	1	0	1	0	0	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
	MULMWQ	P	1	0	1	0	1	0	1	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
	MULUMWQ	P	1	0	1	0	1	1	0	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
	MULSUMWQ	P	1	0	1	0	1	1	1	registerM	0	0	0	0	0	1	0	registerO	0	registerP	–
MAU_QQQQ	MADDWQ	P	1	0	1	0	0	0	0	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
	MADDMWQ	P	1	0	1	0	0	0	1	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
	MADDUMWQ	P	1	0	1	0	0	1	0	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
	MADDSUMWQ	P	1	0	1	0	0	1	1	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
	MSBFWQ	P	1	0	1	0	1	0	0	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
	MSBFMWQ	P	1	0	1	0	1	0	1	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
	MSBFUMWQ	P	1	0	1	0	1	1	0	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
	MSBFSUMWQ	P	1	0	1	0	1	1	1	registerM	1	0	0	0	0	1	0	registerO	0	registerP	–
MAU_CRC	CRCBELMW	P	1	0	1	1	0	0	0	registerW		0	0	0	0	1	0	registerY		registerZ	
	CRCBELLW	P	1	0	1	1	0	0	1	registerW		0	0	0	0	1	0	registerY		registerZ	
	CRCLELMW	P	1	0	1	1	0	1	0	registerW		0	0	0	0	1	0	registerY		registerZ	
	CRCLELLW	P	1	0	1	1	0	1	1	registerW		0	0	0	0	1	0	registerY		registerZ	
MAU_DPP	DOT2WD	P	1	0	1	1	1	0	0	registerW		0	0	0	0	1	0	registerY		registerZ	
	DOT2UWD	P	1	0	1	1	1	0	1	registerW		0	0	0	0	1	0	registerY		registerZ	
	DOT2SUWD	P	1	0	1	1	1	1	0	registerW		0	0	0	0	1	0	registerY		registerZ	
	DOT2W	P	1	0	1	1	1	1	1	registerW		0	0	0	0	1	0	registerY		registerZ	
MAU_SSSS	MADDWD	P	1	0	1	0	0	0	0	registerW		0	0	0	0	1	1	registerY		registerZ	
	MADDUWD	P	1	0	1	0	0	0	1	registerW		0	0	0	0	1	1	registerY		registerZ	
	MADDSUWD	P	1	0	1	0	0	1	0	registerW		0	0	0	0	1	1	registerY		registerZ	
	MADDW	P	1	0	1	0	0	1	1	registerW		0	0	0	0	1	1	registerY		registerZ	
	MSBFWD	P	1	0	1	0	1	0	0	registerW		0	0	0	0	1	1	registerY		registerZ	
	MSBFUWD	P	1	0	1	0	1	0	1	registerW		0	0	0	0	1	1	registerY		registerZ	
	MSBFSUWD	P	1	0	1	0	1	1	0	registerW		0	0	0	0	1	1	registerY		registerZ	
	MSBFW	P	1	0	1	0	1	1	1	registerW		0	0	0	0	1	1	registerY		registerZ	
MAU_SSS	MULWD	P	1	0	1	1	0	0	0	registerW		0	0	0	0	1	1	registerY		registerZ	
	MULUWD	P	1	0	1	1	0	0	1	registerW		0	0	0	0	1	1	registerY		registerZ	
	MULSUWD	P	1	0	1	1	0	1	0	registerW		0	0	0	0	1	1	registerY		registerZ	
	MULW	P	1	0	1	1	0	1	1	registerW		0	0	0	0	1	1	registerY		registerZ	
MAU_QQDD3	MADDHWQ	P	1	0	1	0	0	0	0	registerM	–	0	0	0	1	0	0	registerY		registerZ	
	MADDUHWQ	P	1	0	1	0	0	0	1	registerM	–	0	0	0	1	0	0	registerY		registerZ	
	MADDSUHWQ	P	1	0	1	0	0	1	0	registerM	–	0	0	0	1	0	0	registerY		registerZ	
	MSBFHWQ	P	1	0	1	0	1	0	0	registerM	–	0	0	0	1	0	0	registerY		registerZ	
	MSBFUHWQ	P	1	0	1	0	1	0	1	registerM	–	0	0	0	1	0	0	registerY		registerZ	
	MSBFSUHWQ	P	1	0	1	0	1	1	0	registerM	–	0	0	0	1	0	0	registerY		registerZ	
MAU_QDD3	MULHWQ	P	1	0	1	1	0	0	0	registerM	–	0	0	0	1	0	0	registerY		registerZ	
	MULUHWQ	P	1	0	1	1	0	0	1	registerM	–	0	0	0	1	0	0	registerY		registerZ	
	MULSUHWQ	P	1	0	1	1	0	1	0	registerM	–	0	0	0	1	0	0	registerY		registerZ	
MAU_DIVREM	DIVREMD*	P	1	0	1	scalarcond				registerM	–	0	0	1	0	0	0	registerY		registerP	–
	DIVREMUD*	P	1	0	1	scalarcond				registerM	–	0	0	1	0	0	1	registerY		registerP	–
	DIVREMW*	P	1	0	1	scalarcond				registerM	–	0	0	1	0	1	0	registerY		registerP	–
	DIVREMUW*	P	1	0	1	scalarcond				registerM	–	0	0	1	0	1	1	registerY		registerP	–
MAU_COPYQ	COPYQ	P	1	0	1	1	1	1	1	registerM	–	0	0	1	1	1	1	registerY		registerZ	
MAU_FDDDD	FFMAD*	P	1	0	1	0	0	0	0	registerW		0	1	silent	rnd			registerY		registerZ	
	FFMAWD	P	1	0	1	0	0	0	1	registerW		0	1	silent	rnd			registerY		registerZ	
	FFMAWP*	P	1	0	1	0	0	1	0	registerW		0	1	silent	rnd			registerY		registerZ	
	FFMAHQ	P	1	0	1	0	0	1	1	registerW		0	1	silent	rnd			registerY		registerZ	
	FFMSD*	P	1	0	1	0	1	0	0	registerW		0	1	silent	rnd			registerY		registerZ	
	FFMSWD	P	1	0	1	0	1	0	1	registerW		0	1	silent	rnd			registerY		registerZ	
	FFMSWP*	P	1	0	1	0	1	1	0	registerW		0	1	silent	rnd			registerY		registerZ	
	FFMSHQ	P	1	0	1	0	1	1	1	registerW		0	1	silent	rnd			registerY		registerZ	

Kalray – all rights reserved

MAU_FDDD	FMULD	P	1	0	1	1	0	0	0	registerW		0	1	silent	rnd	registerY		registerZ	
	FMULWD	P	1	0	1	1	0	0	1	registerW		0	1	silent	rnd	registerY		registerZ	
	FMULWP	P	1	0	1	1	0	1	0	registerW		0	1	silent	rnd	registerY		registerZ	
	FMULHQ	P	1	0	1	1	0	1	1	registerW		0	1	silent	rnd	registerY		registerZ	
	FDOT2W	P	1	0	1	1	1	0	0	registerW		0	1	silent	rnd	registerY		registerZ	
	FDOT2WD	P	1	0	1	1	1	0	1	registerW		0	1	silent	rnd	registerY		registerZ	
	FMULWC*	P	1	0	1	1	1	1	0	registerW		0	1	silent	rnd	registerY		registerZ	
	FMULWC.C*	P	1	0	1	1	1	1	1	registerW		0	1	silent	rnd	registerY		registerZ	
MAU_FDDD1	FADDD	P	1	0	1	0	0	0	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FADDWP	P	1	0	1	0	0	0	1	registerW		1	0	silent	rnd	registerY		registerZ	
	FADDHQ	P	1	0	1	0	0	1	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FADDWC.C	P	1	0	1	0	0	1	1	registerW		1	0	silent	rnd	registerY		registerZ	
	FSBFD	P	1	0	1	0	1	0	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FSBFWP	P	1	0	1	0	1	0	1	registerW		1	0	silent	rnd	registerY		registerZ	
	FSBFHQ	P	1	0	1	0	1	1	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FSBFWC.C	P	1	0	1	0	1	1	1	registerW		1	0	silent	rnd	registerY		registerZ	
MAU_FSSSS	FFMAHW	P	1	0	1	1	0	0	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FFMAW*	P	1	0	1	1	0	0	1	registerW		1	0	silent	rnd	registerY		registerZ	
	FFMSHW	P	1	0	1	1	0	1	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FFMSW*	P	1	0	1	1	0	1	1	registerW		1	0	silent	rnd	registerY		registerZ	
MAU_FSSS	FADDW	P	1	0	1	1	1	0	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FSBFW	P	1	0	1	1	1	0	1	registerW		1	0	silent	rnd	registerY		registerZ	
	FMULW	P	1	0	1	1	1	1	0	registerW		1	0	silent	rnd	registerY		registerZ	
	FMULHW	P	1	0	1	1	1	1	1	registerW		1	0	silent	rnd	registerY		registerZ	
MAU_FQQDD	FFMAWDP	P	1	0	1	0	0	0	0	registerM	0	1	1	silent	rnd	registerY		registerZ	
	FFMAHWQ	P	1	0	1	0	0	0	1	registerM	0	1	1	silent	rnd	registerY		registerZ	
	FFMSWDP	P	1	0	1	0	0	1	0	registerM	0	1	1	silent	rnd	registerY		registerZ	
	FFMSHWQ	P	1	0	1	0	0	1	1	registerM	0	1	1	silent	rnd	registerY		registerZ	
	FMMA212W*	P	1	0	1	0	1	0	0	registerM	0	1	1	silent	rnd	registerY		registerZ	
	FMMS212W*	P	1	0	1	0	1	1	0	registerM	0	1	1	silent	rnd	registerY		registerZ	
MAU_FQDD	FMULWDP	P	1	0	1	0	0	0	0	registerM	1	1	1	silent	rnd	registerY		registerZ	
	FMULHWQ	P	1	0	1	0	0	0	1	registerM	1	1	1	silent	rnd	registerY		registerZ	
	FMULWDC*	P	1	0	1	0	0	1	0	registerM	1	1	1	silent	rnd	registerY		registerZ	
	FMULWDC.C*	P	1	0	1	0	0	1	1	registerM	1	1	1	silent	rnd	registerY		registerZ	
	FMM212W*	P	1	0	1	0	1	0	0	registerM	1	1	1	silent	rnd	registerY		registerZ	
MAU_FQQQ2	FMULHO	P	1	0	1	0	1	0	1	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FADDHO	P	1	0	1	0	1	1	0	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FSBFHO	P	1	0	1	0	1	1	1	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
MAU_FQQQQ	FFMAWQ*	P	1	0	1	1	0	0	0	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
	FFMSWQ*	P	1	0	1	1	0	0	1	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
	FFMAHO	P	1	0	1	1	0	1	0	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
	FFMSHO	P	1	0	1	1	0	1	1	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
MAU_FQQQ0	FADDDP	P	1	0	1	1	1	0	0	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
	FADDDC.C	P	1	0	1	1	1	0	1	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
	FSBFDP	P	1	0	1	1	1	1	0	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
	FSBFDC.C	P	1	0	1	1	1	1	1	registerM	0	1	1	silent	rnd	registerO	–	registerP	–
MAU_FQQQ1	FADDWQ	P	1	0	1	1	0	0	0	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FADDWCP.C	P	1	0	1	1	0	0	1	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FSBFWQ	P	1	0	1	1	0	1	0	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FSBFWCP.C	P	1	0	1	1	0	1	1	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FDOT2WDP	P	1	0	1	1	1	0	0	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FDOT2WZP	P	1	0	1	1	1	0	1	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FMULWQ	P	1	0	1	1	1	1	0	registerM	1	1	1	silent	rnd	registerO	–	registerP	–
	FMULDP*	P	1	0	1	1	1	1	1	registerM	1	1	1	silent	rnd	registerO	–	registerP	–

Kalray – all rights reserved

©2025 Kalray

BCU_PCREL11.XPCREL38*	P	0 0	0 0	upper27	1	0 0	0 0	0 0 0 0	lower11	0 0 0 0 0 0	– – – – – –

Kalray – all rights reserved

| | | |

| | | |

| | | |

©2025 Kalray

| | | |

| | | |

©2025 Kalray

| | | |

| | | |

| | | |

| | | |

©2025 Kalray

Resource	Availability
ISSUE	8
TINY	4
LITE	2
FULL	1
LSU	1
MAU	1
BCU	1
EXT	1
AUXR	1
AUXW	1
XFER	1
MEMW	1
SR12	1
SR13	1
SR14	1
SR15	4

Table 14: Kalray-1 processor execution resources.

6.14 Instruction Reservation Tables

The instruction scheduling constraints of the Kalray kv3 core with regards to execution resources are exactly represented by *reservation tables*. A reservation table has one row per core execution resource, and one column per execution cycle relative to the instruction issue date. Each entry in a reservation table records the amount of execution resource required. Parallel and pipelined instruction execution is allowed as long as the sum of required execution resources in all the instruction reservation tables does not exceed the execution resources available in the core, see Table 14.

6.14.1 Reservation Table ALL

Instruction must be alone in instruction bundle.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	8
TINY	
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.2 Reservation Table ALU_FULL

Instruction must issue on the ALU0.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	1
FULL	1
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.3 Reservation Table ALU_FULL.X

Instruction with one immediate extension, must issue on the ALU0.

Kalray – all rights reserved

©2025 Kalray

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	1
FULL	1
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.4 Reservation Table ALU_FULL.Y

Instruction with two immediate extensions, must issue on the ALU0.

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	1
FULL	1
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.5 Reservation Table ALU_LITE

Instruction must issue on one of the two LITE units.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	1
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.6 Reservation Table ALU_LITE.X

Instruction with one immediate extension, must issue on one of the two LITE units.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	1
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.7 Reservation Table ALU_LITE.Y

Instruction with two immediate extensions, must issue on one of the two LITE units.

Kalray – all rights reserved

©2025 Kalray

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	1
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.8 Reservation Table ALU_LITE_CRWH

Instruction must issue on one of the two LITE units and writes to high part of coprocessor register.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	1
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	1
SR15	
Stalls	

Kalray – all rights reserved

6.14.9 Reservation Table ALU_LITE_CRWL

Instruction must issue on one of the two LITE units and writes to low part of coprocessor register.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	1
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	1
SR14	
SR15	
Stalls	

6.14.10 Reservation Table ALU_NOP

Instruction is a no-operation.

Resource	Cycle 0
ISSUE	1
TINY	
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	1
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.11 Reservation Table ALU_TINY

Instruction must issue on one of the four TINY units.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.12 Reservation Table ALU_TINY.X

Instruction with one immediate extension, must issue on one of the four TINY units.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

6.14.13 Reservation Table ALU_TINY.Y

Instruction with two immediate extensions, must issue on one of the four TINY units.

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.14 Reservation Table ALU_TINY_CRRP

Instruction must issue on one of the four TINY units and uses the coprocessor read ports.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	1
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.15 Reservation Table ALU_TINY_CRRP_CRWL_CRWH

Instruction must issue on one of the four TINY units and uses the coprocessor read and write ports.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	1
SR13	1
SR14	1
SR15	
Stalls	

6.14.16 Reservation Table ALU_TINY_CRWH

Instruction must issue on one of the four TINY units and uses the coprocessor write ports.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	1
SR15	
Stalls	

Kalray – all rights reserved

6.14.17 Reservation Table ALU_TINY_CRWL

Instruction must issue on one of the four TINY units and uses the coprocessor write ports.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	1
SR14	
SR15	
Stalls	

6.14.18 Reservation Table ALU_TINY_CRWL_CRWH

Instruction must issue on one of the four TINY units and uses the coprocessor write ports.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	1
SR14	1
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.19 Reservation Table ALU_TINY_CRWL_CRWH.X

Instruction with one immediate extension, must issue on one of the four TINY units and uses the coprocessor write ports.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	1
SR14	1
SR15	
Stalls	

6.14.20 Reservation Table ALU_TINY_CRWL_CRWH.Y

Instruction with two immediate extensions, must issue on one of the four TINY units and uses the coprocessor write ports.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	1
SR14	1
SR15	
Stalls	

6.14.21 Reservation Table BCU

Instruction must issue on the BCU.

Resource	Cycle 0
ISSUE	1
TINY	
LITE	
FULL	
LSU	
MAU	
BCU	1
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.22 Reservation Table BCU_CRRP_CRWL_CRWH

Instruction must issue on the BCU and uses the coprocessor read and write ports.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	1
TINY	
LITE	
FULL	
LSU	
MAU	
BCU	1
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	1
SR13	1
SR14	1
SR15	
Stalls	

6.14.23 Reservation Table BCU_TINY_AUXW_CRRP

Instruction must issue on the BCU, uses the auxiliary write port and uses the coprocessor read ports.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	
BCU	1
EXT	
AUXR	
AUXW	1
XFER	
MEMW	
SR12	1
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

6.14.24 Reservation Table BCU_TINY_TINY_MAU_XNOP

Instruction must issue on the BCU and uses the MAU write port, exclude NOPs.

Resource	Cycle 0
ISSUE	1
TINY	2
LITE	
FULL	
LSU	
MAU	1
BCU	1
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	4
Stalls	

6.14.25 Reservation Table BCU_XFER

Instruction must issue on the BCU and unconditionally transfers control.

Resource	Cycle 0
ISSUE	1
TINY	
LITE	
FULL	
LSU	
MAU	
BCU	1
EXT	
AUXR	
AUXW	
XFER	1
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.26 Reservation Table EXT

Instruction must issue on the EXT.

Resource	Cycle 0
ISSUE	1
TINY	
LITE	
FULL	
LSU	
MAU	
BCU	
EXT	1
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.27 Reservation Table LSU

Instruction must issue on the LSU.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

6.14.28 Reservation Table LSU.X

Instruction with one immediate extension, must issue on the LSU.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.29 Reservation Table LSU.Y

Instruction with two immediate extensions, must issue on the LSU.

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.30 Reservation Table LSU_AUXR

Instruction must issue on the LSU, and requires the auxiliary read port.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	1
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.31 Reservation Table LSU_AUXR.X

Instruction with one immediate extension, must issue on the LSU, and requires the auxiliary read port.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	1
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

6.14.32 Reservation Table LSU_AUXR.Y

Instruction with two immediate extensions, must issue on the LSU, and requires the auxiliary read port.

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	1
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.33 Reservation Table LSU_AUXR_AUXW

Instruction must issue on the LSU, and requires the auxiliary read and write ports.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	1
AUXW	1
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.34 Reservation Table LSU_AUXR_AUXW.X

Instruction with one immediate extension, must issue on the LSU, and requires the auxiliary read and write ports.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	1
AUXW	1
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.35 Reservation Table LSU_AUXR_AUXW.Y

Instruction with two immediate extensions, must issue on the LSU, and requires the auxiliary read and write ports.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	1
AUXW	1
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.36 Reservation Table LSU_AUXW

Instruction must issue on the LSU, and requires the auxiliary write port.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	1
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.37 Reservation Table LSU_AUXW.X

Instruction with one immediate extension, must issue on the LSU, and requires the auxiliary write port.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	1
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.38 Reservation Table LSU_AUXW.Y

Instruction with two immediate extensions, must issue on the LSU, and requires the auxiliary write port.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	1
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.39 Reservation Table LSU_CRRP

Instruction must issue on the LSU and reads a coprocessor register pair.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	1
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.40 Reservation Table LSU_CRRP.X

Instruction with one immediate extension, must issue on the LSU and reads a coprocessor register pair.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	1
SR13	
SR14	
SR15	
Stalls	

6.14.41 Reservation Table LSU_CRRP.Y

Instruction with two immediate extensions, must issue on the LSU and reads a coprocessor register pair.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	1
MAU	
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	1
SR13	
SR14	
SR15	
Stalls	

6.14.42 Reservation Table MAU

Instruction must issue on the MAU.

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	1
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.43 Reservation Table MAU.X

Instruction with one immediate extension, must issue on the MAU.

Kalray – all rights reserved

©2025 Kalray

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	
MAU	1
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.44 Reservation Table MAU.Y

Instruction with two immediate extensions, must issue on the MAU.

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	
MAU	1
BCU	
EXT	
AUXR	
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.45 Reservation Table MAU_AUXR

Instruction must issue on the MAU, and requires the auxiliary read port.

Kalray – all rights reserved

Resource	Cycle 0
ISSUE	1
TINY	1
LITE	
FULL	
LSU	
MAU	1
BCU	
EXT	
AUXR	1
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

6.14.46 Reservation Table MAU_AUXR.X

Instruction with one immediate extension, must issue on the MAU, and requires the auxiliary read port.

Resource	Cycle 0
ISSUE	2
TINY	1
LITE	
FULL	
LSU	
MAU	1
BCU	
EXT	
AUXR	1
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

©2025 Kalray

6.14.47 Reservation Table MAU_AUXR.Y

Instruction with two immediate extensions, must issue on the MAU, and requires the auxiliary read port.

Resource	Cycle 0
ISSUE	3
TINY	1
LITE	
FULL	
LSU	
MAU	1
BCU	
EXT	
AUXR	1
AUXW	
XFER	
MEMW	
SR12	
SR13	
SR14	
SR15	
Stalls	

Kalray – all rights reserved

6.15 Instruction Constraints

The Kalray kv3 core bundling constraints are implied by the instruction scheduling constraints on dependence latencies and execution resources. In particular, a bundle is valid if the sum of the first column of each reservation table of the bundled instructions does not exceed the availability of execution resources given in Table 14.

6.15.1 Instruction Bundling Constraints

Rule 1 Two data dependent instructions may not be scheduled within a single bundle.

Rule 2 Two instructions of the same bundle must not target the same architectural resource, otherwise the update value of this resource is unpredictable.

Rule 3 (Specialization of Rule 2) An ALU instruction that produces a carry must not be bundled with a `SET` or `WFXL` instruction on CS.

Rule 4 (Specialization of Rule 2) An ALU or MAU instruction that produces an IEEE 754 floating point flag must not be bundled with a `SET` or `WFX*` instruction on CS.

Rule 5 If a `GET`, `IGET` or `WAITIT` instruction is issued within a bundle, then the BCU and the tiny ALU unit inside the MAU are used. As a consequence:

- No MAU instruction is allowed in this bundle
- Only 2 tiny ALU instructions may be used in this bundle (one on each ALU).

This rule is implicit and covered by the instruction reservation tables.

6.15.2 Instruction Scheduling Constraints

Rule 6 – Mandatory PC must not be the target of a `hfx*` or a `set` instruction, else a TRAP.OPCODE is generated. A simple branch may be used to modify the PC if needed.

Rule 7 – Mandatory The following instructions may only be issued alone in the bundle:

- `RFE`
- `SCALL`
- `AWAIT`
- `SLEEP`
- `STOP`
- `BARRIER`

Kalray – all rights reserved

©2025 Kalray

- LOOPDO
- TLBREAD
- TLBWRITE
- TLBIINVAL
- TLBDINVAL
- TLBPROBE
- SET/WFXL on PS/MMC/*OW

Rule 8 – Mandatory LS, LE and LC must not be modified within the hardware loop.

Rule 9 – Mandatory The instructions that enable or disable hardware loops must not be used inside the corresponding loop.

Rule 10 – Mandatory To avoid any semantic ambiguity and simplify the design of the hardware, branching instructions, including those triggering implicit refetches, are not allowed at a hardware loop last bundle (generates OPCODE traps). Those are:

- CALL
- ICALL
- CB
- GOTO
- IGOTO
- RET
- RFE
- SCALL
- BARRIER
- LOOPDO
- TLBREAD
- TLBWRITE
- TLBIINVAL
- TLBDINVAL
- TLBPROBE
- SET/WFXL on PS

Kalray – all rights reserved

Rule 11 – Mandatory In order to ensure a correct behavior while executing an hardware loop, a re-fetch must be generated after a set instruction involving LS, LE or LC registers. This re-fetch can be done through a `BARRIER` instruction, but also through the RFE or RET instructions.

Rule 12 – Mandatory Similarly, a refetch has to be generated after an `IINVAL*` instruction. This re-fetch can be done through a `BARRIER` instruction, but also through the `RFE` or RET instructions.

Rule 13 – Mandatory After changing the value of the rounding mode (RM) field of CS, a `BARRIER` instruction must be issued before any MAU/FPU instruction, to guarantee that these instructions will see the new value of RM.

Rule 14 – Mandatory Similarly, after changing the value of PMC, a `BARRIER` is needed before a `GET` instruction on the concerned registers among PM0, PM1, PM2 and PM3, to make sure that the correct value will be read.

Rule 15 – Performance For hardware simplicity, there are no bypasses on system function registers. Then, in case of RAW dependency, the core will stall until the write completion. However, two specific cases are optimized:

- A `RET` instruction after a RA update will be executed seeing immediately the new RA value, without stalls.
- An instruction which involves the carry flag after a CS update will be executed seeing immediately the new carry value, without stalls.

Kalray – all rights reserved

7 Application Binary Interface

7.1 Introduction

The Kalray kv3 core Application Binary Interface (ABI) is a set of conventions designed for inter-operability between the different code-generation tools and libraries.

This ABI is little-endian only for bytes and bits. Dynamic linking assumes that position-independent code and data are accessed through pointers materialized in the called function.

The 32-bit addressing code generation uses a convention called ILP32, in which integers, long integers, and pointers are 32-bit quantities. The 64-bit addressing code generation uses the LP64 convention, in which integers are 32-bit quantities, while long integers and pointers are 64-bit quantities. In both cases, `char` is a signed type and `long long` is a 64-bit integer.

7.2 Memory Addressing Model

A program that executes on the Kalray kv3 core views the addressable memory as a collection of *segments*. A segment is an area of memory that has specific uses within an application and has no fixed address relationship to any other segment. Thus, relative distances between any two items belonging to the same segment are constant once the program has been linked, but the distance between two items in different segments is not fixed. The program view of the addressable memory as a collection of segments is independent from the definition of address ranges with associated attributes in the MMU TLB entries. It is expected, however, that the MMU be configured to enforce the correct attributes of the segments once their addresses are resolved by the linker or the loader. The following memory segments are defined:

Text Contains the executable code, and exception tables.

Data Contains static storage variables with initialization values.

Heap Contains storage managed by dynamic memory allocation.

Stack Contains the stack frames of the executing functions.

Device Memory cells whose access triggers side-effects on hardware.

The memory segments except the device segment are populated by the linker or the loader by concatenating the contributions from the program units that compose the executable program. These contributions are contained into *section*s, whose mapping of to segments is as follows:

Section	Segment	Contents
`.text`	Text	Executable code
`.data`	Data	Initialized static data
`.bss`	Data	Cleared on start static data
`.rodata`	Data	Read-only data
`.tdata`	Data	Initialized thread-local data
`.tbss`	Data	Cleared on start thread-local data

Kalray – all rights reserved

7.3 Data Layout Rules

7.3.1 Native C Types

Native types of the C language are *naturally aligned*, that is, their address must be a multiple of their size as computed by the `sizeof` operator in C.

While the natural alignment of native types is not required by the Kalray kv3 architecture, it promotes more efficient code at little cost in storage.

7.3.2 C Aggregate Types

C aggregate types comprise arrays, complex types, GNU C vector types, `struct` types, and `union` types:

- An array is aligned on the same boundary as its element type.
- Complex types and GNU C vector types inherit the alignment or their elements. However, for performance reasons, they may be aligned on a multiple of their size in data sections and stack frames.
- C `struct` and `union` types inherit the alignment of their most strictly aligned member.
- The size of any aggregate type is always a multiple of its alignment.
- Each aggregate type member is assigned to the lowest available offset with the appropriate alignment. This may require internal padding, depending on the previous member.
- An aggregate size is increased, if necessary, to make it a multiple of the alignment. This may require tail padding, depending on the last member type.
- There are no minimal size requirements for a `struct` or `union` type.

The content of any padding is undefined.

7.3.3 Bit-Field Layout

C `struct` and `union` types may have bit-fields that define integral objects with a specified number of bits. The Kalray kv3 run-time architecture does not specify the signedness of bit-field types. This is specified by the ISO C99 language standard for all bit-field types except for plain `int`. The signedness of plain int bit-fields should be specified by the compiler implementation. Bit-fields obey the same size and alignment rules as other `struct` and `union` members, with the following additions:

- Bit-fields are allocated least significant bit to most significant bit (from right to left).
- A named bit-field must entirely reside in a storage unit appropriate for its declared type. For example, a bit-field of type short must never cross a half-word boundary. (This bit-field alignment rule is inherited from the historic PCC compiler.)

Kalray – all rights reserved

- The type of unnamed bit-fields does not affect the alignment of the enclosing `struct` or `union` type. Zero-length unnamed bit-fields force the alignment of subsequent members to the boundary corresponding to the size of the bit-field type.

7.4 Stack Frame Organization

7.4.1 Stack Frame Addressing

A function stack frame is an area of memory in the stack segment which is dedicated to the activation of a called function. The stack grows downwards, and is delimited by the value of the stack pointer (SP) register that points to its lowest address. An application may not write to memory below the stack pointer. Any memory below the stack pointer may be destroyed at any time, for instance by interrupt handlers.

The stack pointer must always be aligned on a 32-byte boundary. This implies that all stack frames must be a multiple of 32 bytes in size. When composing a function stack frame, small object sizes and alignment are rounded up to 8 bytes. Objects that have stricter alignment than 8 bytes are assigned a slot that ensures this stricter alignment, typically by allocating more stack slots at compile time then rounding the address at execution time.

The frame grows downwards, meaning that local variable slots are addressed by a negative offset from a virtual frame pointer. The first local variable slot is thus at offset -8 from the virtual frame pointer. The virtual frame pointer is an abstraction and is different from the address maintained into the frame pointer (FP) register, also known as hard frame pointer. The FP register points to the first address above the Dynamic Area region, described next.

7.4.2 Stack Frame Regions

A function stack frame comprises up to six regions, listed from the highest address to the lowest address in the stack frame (Figure 7):

Anonymous Arguments This region contains space for storing the incoming anonymous arguments passed in registers in case the function is variadic. Anonymous arguments, also known as unnamed arguments, are those passed at position starting at ellipsis '`...`' in the prototype of a variadic function.

Local Variables A function may store local variables and compiler temporaries in this region. In case a static chain pointer is required, it uses the first local variable slot. The static chain pointer is used in languages that support nested function definitions, in order to access the local variables of the enclosing functions. Nested functions are supported as an extension in GNU C.

Registers Save A function may store callee-saved (preserved) registers in this region before reusing them. A function may also store caller-saved (scratch) registers in this region before calling a function.

Frame Marker This region contains information required for unwinding the stack at run-time, and is pointed to by the FP register. It contains a pointer to the previous frame marker

Kalray – all rights reserved

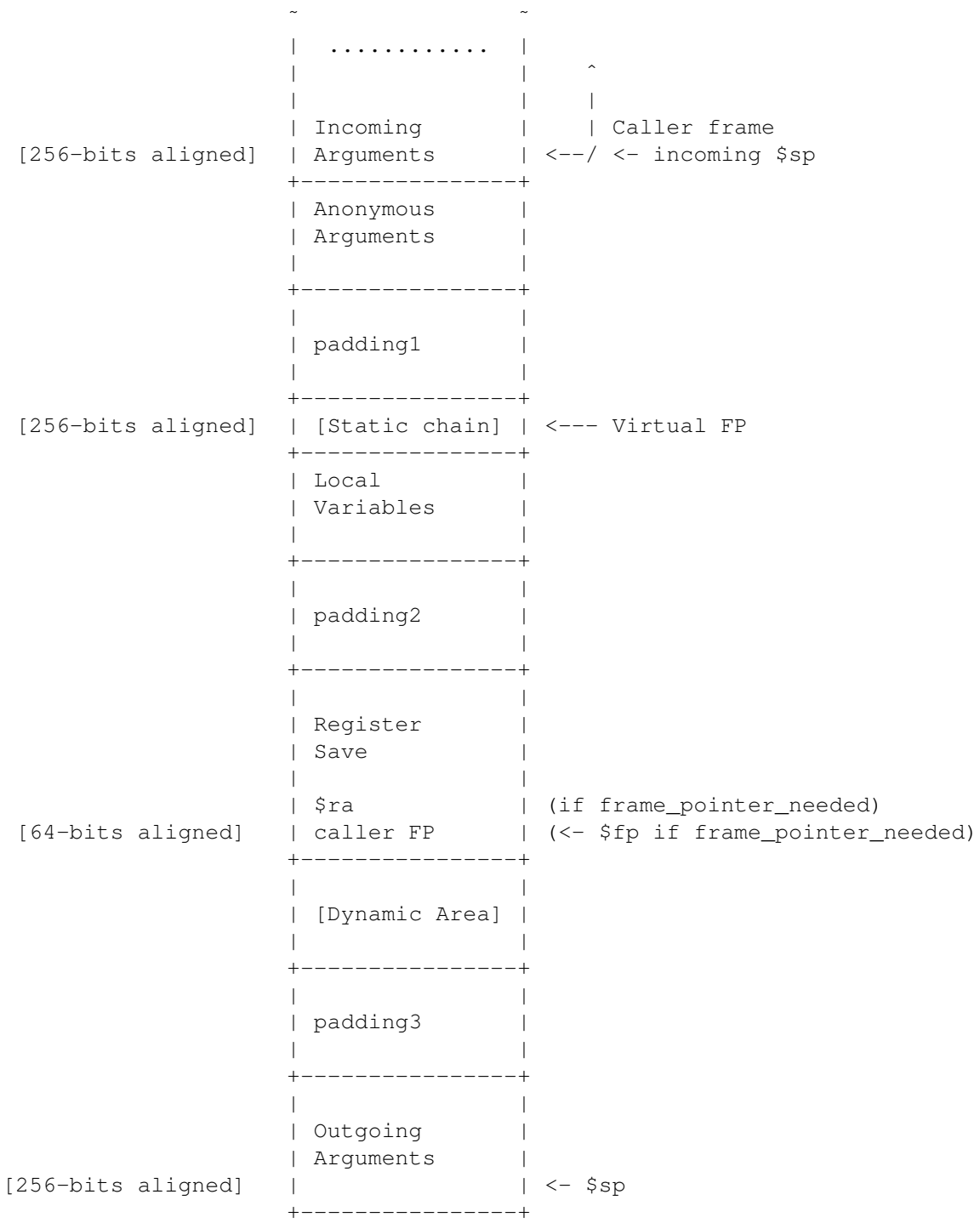


Figure 7: Stack frame regions.

Kalray – all rights reserved

(the caller FP), and the return address of the function. The pointer is NULL if there is no previous frame marker.

Dynamic Area This region is variable-sized and created in case the function calls `alloca()` or declares ISO C99 variable length arrays. When the Dynamic Area exists, a Frame Marker region is required. Otherwise, the Frame Marker region is optional.

Outgoing Arguments Successive arguments are passed in argument slots, with the first twelve ones located in registers R0–R11, and the other on the stack in the Outgoing Arguments region. The Outgoing Argument region on the stack is sized as the maximum required by all functions called from the current one. Stack space is not allocated for the arguments that are passed in registers, whether named or anonymous.

7.5 Nested Functions Implementation

As mentionned in 7.4.2, a stack frame may need a static chain slot which uses the first local variable slot. This slots stores a pointer to the location of the local variables of its caller (whose activation record is stored higher in the stack).

Let's recall that the situation is pretty different for the caller (the function which has a nested function) and the callee (the nested function itself), even though a function can be both at the same time if there is a nested function in a nested function.

The caller has to give away a pointer pointing on its local variables so that the callee can store it in the its static chain slot for later use. Currently this is done by using GPR `$r31` (a callee saved register used as a scratch register). When calling a nested function, the calling function:

1. Store its static chain in `$r31`,

2. Call the nested function

3. The nested function stores the address of the static chain in a special slot in its frame. This is done in the prologue of the function, once `$r31` has been saved it can be used again, saving `$r31` is espacially useful when the depth of the nest of nested function is greater than one (ie, there is at least one nested function in a nested function).

In case there is a need to pass the address of a nested function to another function (whose stack frame will be below the function which performs the call (the general case is not covered)), the calling function sets up a trampoline of size (12 * 2 + 4 = 28) on the stack composed of:

```
 0:  make $r31 = ptr to static chain of caller
 c:  make $r30 = ptr to the nested function
10:  igoto $r30 (Note that icall would destroy $ra.)
```

Once this is done, the calling function gives the address of the trampoline instead of the real address of the nested function. This way, when it is called it first setup the static chain.

Kalray – all rights reserved

7.5.1 Regarding cache alignement and cache invalidation

The macro `TRAMPOLINE_ALIGNEMENT = 32`, should guarantee by construction that the trampoline is correctly aligned on the stack and the three instructions (each in a separate bundle) should not overlap two cache lines such that the a call to `i1invals (static_chain_address)` clears the cache line where those 3 instructions are.

7.6 Function Arguments and Result

Arguments are passed in a combination of general registers and memory, as described below. Arguments are first allocated on a conceptual argument list, that begins at relative address 0, and are allocated towards increasing addresses. Each storage unit in the argument list, called argument slot, has a 8-byte size and alignment. Argument slots are referred to as slot 0, slot 1, slot 2, etc. Arguments are aligned to 8-byte boundary and padded within the argument list according to the following rules:

- Integral scalar arguments smaller than 64 bits are padded on the left to a total width of 64 bits (that is, the value is placed in the least-significant bits of the argument slot), and placed in the next available argument slot.
- Half-precision floating-point arguments are not promoted to single-precision floating-point. They are passed directly with 48-bit of padding.
- Single-precision complex arguments are passed as a single argument. The real part is passed in the 32 least significant bits and the imaginary part in the 32 most significant bits.
- Double-precision complex arguments are passed as a pair of double-precision floating-point scalar arguments. The real part is passed first and the imaginary part second.
- In C++, aggregates that are not trivially copyable classes are always passed by reference. A trivially copyable class has no virtual functions or virtual base classes.
- Aggregates up to 8 bytes in size, when passed by value, are padded to a total width of 64 bits and placed in the next available argument slot. If shorter than 8 bytes, the aggregate begins at the lowest-numbered byte in the storage unit, so the padding is on the highest-numbered bytes.
- Aggregates and data types larger than 8 bytes and up to 32 bytes are passed by value. They are padded to a multiple of 64 bits, aligned to an 8-byte boundary, and placed in as many argument slots as needed. Any padding required is on the highest-numbered bytes of the final argument slot.
- Aggregates and data types of size larger than 32 bytes are passed by reference, with a pointer using a full argument slot in place of the argument itself. It this the caller's responsibility to allocate a memory buffer for a copy of the original argument, copy into it, and pass the pointer to this copy in the corresponding argument slot.

Kalray – all rights reserved

- As an extension, GNU C allows variable-length arrays as members of `struct` or `union` types. Such aggregates have a variable size and are always passed by reference.
- If an argument straddles the boundary between slot 11 and slot 12, the part that lies within the first twelve slots is passed in general registers, and the remainder is passed in the Outgoing Arguments region.

The content of any padding is undefined.

The first 12 arguments slots of the argument list are allocated into the general-purpose registers R0–R11. If more than 12 slots are required by the argument list, these are allocated in the Outgoing Arguments region with increasing addresses, starting by the 8-byte stack slot pointed to by the current SP (Figure 7). So slot 12 is placed at location SP, slot 13 at SP+8, and so on. When allocating arguments to slots, the only alignment enforced is 8 bytes. This is possible because the Kalray kv3 architecture does not require any memory alignment for correct behavior. As a result, some multi-register arguments may end up in mis-aligned register tuples. It is the callee's responsibility to move such arguments into well-aligned register tuples before using them as operands of wide load-store and SIMD instructions. Likewise, it is the caller's responsibility to move such arguments from well-aligned register tuples.

The rules above support variable-argument list functions in both the K&R C and the ISO C99 languages. Since the Kalray kv3 core does not specialize registers, there is no issue whether the presence or absence of a function prototype changes the way arguments are passed. Thus, a variadic function may assume that the anonymous arguments that lie within the first twelve argument slots can all be found in the argument passing general registers, R0–R11. It may then store these registers to memory, providing up to 96 bytes of Anonymous Argument region on top of its own stack frame for R0–R11, as necessary (Figure 7). This arrangement places all the anonymous arguments in one contiguous block of memory.

Return values up to 32 bytes (4×8) bytes are returned directly in registers, according to the same rules as if they were to be passed as the first parameter of an argument list. Return values larger than 32 bytes or of dynamic size are returned in a memory buffer allocated by the caller. In this case, a pointer to the buffer is passed to the called function in R15.

7.7 Function Linkage

7.7.1 Register Usage Conventions

Depending on the code generation model, the Kalray kv3 core registers must obey conventions for proper function linkage. The function linkage conventions specify agreement between the caller and the callee on the following register usage:

reserved All registers that are not allocatable (such as SP) must be reserved.

argument List of function call argument registers.

result List of function call result registers.

struct The result pointer for functions that return a `struct` or `union` type.

Kalray – all rights reserved

Role	Registers
reserved	$r12 $r13
argument	$r0 $r1 $r2 $r3 $r4 $r5 $r6 $r7 $r8 $r9 $r10 $r11
result	$r0 $r1 $r2 $r3
struct	$r15
caller	$r0 $r1 $r2 $r3 $r4 $r5 $r6 $r7 $r8 $r9 $r10 $r11 $r15 $r16 $r17 $r32 $r33 $r34 $r35 $r36 $r37 $r38 $r39 $r40 $r41 $r42 $r43 $r44 $r45 $r46 $r47 $r48 $r49 $r50 $r51 $r52 $r53 $r54 $r55 $r56 $r57 $r58 $r59 $r60 $r61 $r62 $r63
callee	$r14 $r18 $r19 $r20 $r21 $r22 $r23 $r24 $r25 $r26 $r27 $r28 $r29 $r30 $r31
program	$pc
veneer	$r16 $r17
return	$ra
stack	$r12
frame	$r14
local	$r13

Table 15: Register usage conventions.

caller List of caller-saved registers, also known as scratch registers.

callee List of callee-saved registers, also known as preserved registers.

program The program counter register.

return The return address register.

stack The stack pointer (SP) register.

frame The frame pointer (FP) register.

local Local data or Thread Local Storage (TLS) pointer.

Although not listed above, the coprocessor registers are considered caller-saved (scratch). It is recommended to use the coprocessor registers only in leaf functions, as saving and restoring them implies a significant run-time cost.

When compiling, static storage data are accessed by building addresses in code. Register usage conventions for the Kalray kv3 code generation mode are defined in Table 15. When compiling in PIC (Position-Independent Code) mode, static storage data are accessed relative to a Global Pointer (GP) that is set depending on the load module the function belongs to. The Kalray kv3 PIC code generation assumes the callee sets the GP.

7.7.2 Function Call Sequence

Direct and indirect function calls follow the same sequence before the call and after the call. The function calling sequence is as follows:

Kalray – all rights reserved

Call preparation Values in caller-saved (scratch) registers that must be kept live across the call must be saved. They can be saved by either copying them into preserved registers or by saving them into the function stack frame.

Call instruction Use CALL or ICALL for PC-relative direct calls and indirect calls respectively. These instructions write the value of the PC of the next bundle in register RA, then change the PC to the target address of the call.

For direct calls the PC-relative displacement to the target is computed at link time. Compilers may assume that the standard displacement field in the call instruction is sufficiently wide to reach the target of the call. If the displacement is too large, the linker must supply a branch stub at some convenient point in the code; compilers must guarantee the existence of such a point by ensuring that code sections in the relocatable object files are no larger than the maximum reach of the call instruction.

Function entry The prologue code in the target function is responsible for allocating a frame on the memory stack, if necessary. If it is a non-leaf function, it must save the RA register in the stack frame. The prologue must also save any callee-saved (preserved) registers that will be used in this function.

Function exit The epilogue code is responsible for restoring the RA register and any callee-saved (preserved) registers that were saved. If a memory stack frame was allocated, the epilogue code must de-allocate it. Finally, the function exits by branching through the RA register with the RET instruction.

After the call Any caller-saved (scratch) values should be restored.

A compiler can choose to make a tail call (including tail recursion) by simply branching to the function being called. However, this appears to a debugger as if the nested function was called directly from the caller of the original function. If the calling function has a stack frame, it must pop its frame off the stack before making the call. Arguments should be passed in the usual registers. Tail calls can only be made when all parameters can be passed in registers.

7.8 System Call Interface

From the ABI point of view, system calls are seen as regular function calls, so calling conventions are assumed. The system call number is available to the system call handler in the SN field of the PS register.

Kalray – all rights reserved

7.9 Relocation List

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
rel16	R_KVX_16			X						
rel32	R_KVX_32			X						
rel64	R_KVX_64			X						
pcrel_signed16	R_KVX_S16_PCREL	@pcrel		X	X				MAKE	PCREL XSPLATDO
pcrel17	R_KVX_PCREL17			X	X				CB	LOOPDO
pcrel27	R_KVX_PCREL27			X	X				CALL	GOTO
pcrel32	R_KVX_32_PCREL			X	X					
pcrel_signed37	R_KVX_S37_PCREL_UP27 R_KVX_S37_PCREL_LO10	@pcrel		X	X				ABDD	ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
pcrel_signed43	R_KVX_S43_PCREL_EX6 R_KVX_S43_PCREL_UP27 R_KVX_S43_PCREL_LO10	@pcrel		X	X				MAKE	PCREL XSPLATDO
pcrel_signed64	R_KVX_S64_PCREL_EX27 R_KVX_S64_PCREL_UP27 R_KVX_S64_PCREL_LO10	@pcrel		X	X				ABDD	ADDD ANDD ANDND CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD

Kalray – all rights reserved

©2025 Kalray

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
pcrel_signed64										I1INVALS IORD IORND LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAKE MAXD MAXUD MIND MINUD NANDD NEORD NIORD PCREL SB SBFD SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO XSPLATDO
pcrel64	R_KVX_64_PCREL			X	X					
signed16	R_KVX_S16			X					MAKE	 PCREL XSPLATDO
signed32	R_KVX_S32_UP27 R_KVX_S32_LO5	@signed32		X					ABDBO	 ABDD ABDHQ ABDSBO ABDSD ABDSHQ ABDSW ABDSWP ABDUBO ABDUD ABDUHQ ABDUW ABDUWP ABDWP ADDBO ADDCD ADDCD.I ADDD ADDHQ ADDSBO ADDSD ADDSHQ ADDSW ADDSWP ADDUSBO ADDUSD ADDUSHQ ADDUSW ADDUSWP ADDUWD ADDWD ADDWP ADDX16BO ADDX16D ADDX16HQ ADDX16UWD ADDX16W ADDX16WD ADDX16WP ADDX2BO ADDX2D ADDX2HQ ADDX2UWD ADDX2W

Kalray – all rights reserved

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
signed32										ADDX2WD ADDX2WP ADDX32D ADDX32UWD ADDX32W ADDX32WD ADDX4BO ADDX4D ADDX4HQ ADDX4UWD ADDX4W ADDX4WD ADDX4WP ADDX64D ADDX64UWD ADDX64W ADDX64WD ADDX8BO ADDX8D ADDX8HQ ADDX8UWD ADDX8W ADDX8WD ADDX8WP ANDD ANDND AVGBO AVGHQ AVGRBO AVGRHQ AVGRUBO AVGRUHQ AVGRUW AVGRUWP AVGRW AVGRWP AVGUBO AVGUHQ AVGUW AVGUWP AVGW AVGWP CMOVEBO CMOVEHQ CMOVEWP COMPNBO COMPND COMPNHQ COMPNW COMPNWP COMPUWD COMPW COMPWD CRCBELLW CRCBELMW CRCLELLW CRCLELMW EORD FCOMPD FCOMPND FCOMPNHQ FCOMPNW FCOMPNWP FCOMPW IORD IORND LANDW LIORW LNANDW LNIORW MADDD MADDHQ MADDSUWD MADDSUWDP MADDUWD MADDUWDP MADDW MADDWD MADDWDP MADDWP MAXBO

Kalray – all rights reserved

©2025 Kalray

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
signed32										MAXD MAXHQ MAXUBO MAXUD MAXUHQ MAXUWP MAXWP MINBO MIND MINHQ MINUBO MINUD MINUHQ MINUWP MINWP MM212W MMA212W MMS212W MSBFD MSBFHQ MSBFSUWD MSBFSUWDP MSBFUWD MSBFUWDP MSBFW MSBFWD MSBFWDP MSBFWP MULD MULHQ MULSUWD MULSUWDP MULUWD MULUWDP MULW MULWD MULWDP MULWP NANDD NEORD NIORD SBFBO SBFCD SBFCD.I SBFD SBFHQ SBFSBO SBFSD SBFSHQ SBFSW SBFSWP SBFUSBO SBFUSD SBFUSHQ SBFUSW SBFUSWP SBFUWD SBFWD SBFWP SBFX16BO SBFX16D SBFX16HQ SBFX16UWD SBFX16W SBFX16WD SBFX16WP SBFX2BO SBFX2D SBFX2HQ SBFX2UWD SBFX2W SBFX2WD SBFX2WP SBFX32D SBFX32UWD SBFX32W SBFX32WD SBFX4BO SBFX4D SBFX4HQ SBFX4UWD

Kalray – all rights reserved

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
signed32										SBFX4W SBFX4WD SBFX4WP SBFX64D SBFX64UWD SBFX64W SBFX64WD SBFX8BO SBFX8D SBFX8HQ SBFX8UWD SBFX8W SBFX8WD SBFX8WP SBMM8D SBMMT8D STSUD STSUHQ STSUW STSUWP
signed37	R_KVX_S37_UP27 R_KVX_S37_LO10			X					ABDD	ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
gotoff_signed37	R_KVX_S37_GOTOFF_UP27 R_KVX_S37_GOTOFF_LO10	@gotoff		X				X	ABDD	ABDW ADDD ADDW ANDD

Kalray – all rights reserved

©2025 Kalray

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
gotoff_signed37										ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
gotoff_signed43	R_KVX_S43_GOTOFF_EX6 R_KVX_S43_GOTOFF_UP27 R_KVX_S43_GOTOFF_LO10	@gotoff		X				X	MAKE	 PCREL XSPLATDO
gotoff_32	R_KVX_32_GOTOFF	@gotoff		X				X		
gotoff_64	R_KVX_64_GOTOFF	@gotoff		X				X		
got_32	R_KVX_32_GOT	@got		X				X		
got_signed37	R_KVX_S37_GOT_UP27 R_KVX_S37_GOT_LO10	@got		X				X	ABDD	 ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD

Kalray – all rights reserved

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
got_signed37										LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
got_signed43	R_KVX_S43_GOT_EX6 R_KVX_S43_GOT_UP27 R_KVX_S43_GOT_LO10	@got		X				X	MAKE	 PCREL XSPLATDO
got_64	R_KVX_64_GOT	@got		X				X		
glob_dat	R_KVX_GLOB_DAT		X					X		
copy	R_KVX_COPY			X						
jump_slot	R_KVX_JMP_SLOT			X						
relative	R_KVX_RELATIVE			X						
signed43	R_KVX_S43_EX6 R_KVX_S43_UP27 R_KVX_S43_LO10			X					MAKE	 PCREL XSPLATDO
signed64	R_KVX_S64_EX27 R_KVX_S64_UP27 R_KVX_S64_LO10			X					ABDD	 ADDD ANDD ANDND CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD I1INVALS IORD IORND LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAKE MAXD MAXUD MIND MINUD NANDD NEORD NIORD PCREL SB SBFD SBMM8D SBMMT8D SD SH

Kalray – all rights reserved

©2025 Kalray

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
signed64										SO SQ SW XLO XSO XSPLATDO
gotaddr_signed37	R_KVX_S37_GOTADDR_UP27 R_KVX_S37_GOTADDR_LO10	@gotaddr		X	X				ABDD	 ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
gotaddr_signed43	R_KVX_S43_GOTADDR_EX6 R_KVX_S43_GOTADDR_UP27 R_KVX_S43_GOTADDR_LO10	@gotaddr		X	X				MAKE	 PCREL XSPLATDO
gotaddr_signed64	R_KVX_S64_GOTADDR_EX27 R_KVX_S64_GOTADDR_UP27 R_KVX_S64_GOTADDR_LO10	@gotaddr		X	X				ABDD	 ADDD ANDD ANDND CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD I1INVALS IORD IORND LBS LBZ

Kalray – all rights reserved

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
gotaddr_signed64										LD LHS LHZ LO LQ LWS LWZ MAKE MAXD MAXUD MIND MINUD NANDD NEORD NIORD PCREL SB SBFD SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO XSPLATDO
dtpmod64	R_KVX_64_DTPMOD			X						
dtpoff64	R_KVX_64_DTPOFF			X						
dtpoff_signed37	R_KVX_S37_TLS_DTPOFF_UP27 R_KVX_S37_TLS_DTPOFF_LO10	@dtpoff		X					ABDD	ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH

Kalray – all rights reserved

©2025 Kalray

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
dtpoff_signed37										SO SQ SW XLO XSO
dtpoff_signed43	R_KVX_S43_TLS_DTPOFF_EX6 R_KVX_S43_TLS_DTPOFF_UP27 R_KVX_S43_TLS_DTPOFF_LO10	@tlsgd		X					MAKE	 PCREL XSPLATDO
tlsgd_signed37	R_KVX_S37_TLS_GD_UP27 R_KVX_S37_TLS_GD_LO10	@tlsgd		X				X	ABDD	 ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
tlsgd_signed43	R_KVX_S43_TLS_GD_EX6 R_KVX_S43_TLS_GD_UP27 R_KVX_S43_TLS_GD_LO10	@tlsgd		X				X	MAKE	 PCREL XSPLATDO
tlsld_signed37	R_KVX_S37_TLS_LD_UP27 R_KVX_S37_TLS_LD_LO10	@tlsld		X				X	ABDD	 ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL

Kalray – all rights reserved

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
tlsld_signed37										EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
tlsld_signed43	R_KVX_S43_TLS_LD_EX6 R_KVX_S43_TLS_LD_UP27 R_KVX_S43_TLS_LD_LO10	@tlsld		X				X	MAKE	 PCREL XSPLATDO
tpoff64	R_KVX_64_TPOFF			X			X			
tlsie_signed37	R_KVX_S37_TLS_IE_UP27 R_KVX_S37_TLS_IE_LO10	@tlsie		X				X	ABDD	ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND

Kalray – all rights reserved

©2025 Kalray

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
tlsie_signed37										MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
tlsie_signed43	R_KVX_S43_TLS_IE_EX6 R_KVX_S43_TLS_IE_UP27 R_KVX_S43_TLS_IE_LO10	@tlsie		X				X	MAKE	 PCREL XSPLATDO
tlsle_signed37	R_KVX_S37_TLS_LE_UP27 R_KVX_S37_TLS_LE_LO10	@tlsle		X			X		ABDD	 ABDW ADDD ADDW ANDD ANDND ANDNW ANDW CMOVED COMPD DFLUSHL DINVALL DPURGEL DTOUCHL EORD EORW I1INVALS IORD IORND IORNW IORW LBS LBZ LD LHS LHZ LO LQ LWS LWZ MAXD MAXUD MAXUW MAXW MIND MINUD MINUW MINW NANDD NANDW NEORD NEORW NIORD NIORW SB SBFD SBFW SBMM8D SBMMT8D SD SH SO SQ SW XLO XSO
tlsle_signed43	R_KVX_S43_TLS_LE_EX6	@tlsle		X			X		MAKE	

Kalray – all rights reserved

Relocation	Linker relocations	Syntax	Symbol	Symbol + addend	PC	GP	TP	GOT	Instructions	
tlsle_signed43	R_KVX_S43_TLS_LE_UP27 R_KVX_S43_TLS_LE_LO10									PCREL XSPLATDO
rel8	R_KVX_8			X						
pcrel11	R_KVX_PCREL11			X	X					
pcrel38	R_KVX_S38_PCREL_UP27 R_KVX_S38_PCREL_LO11			X	X					

Kalray – all rights reserved

8 Application Optimization Guide

8.1 Uses of the Bit Matrix Multiplies 8×8

8.1.1 Bit Matrix 8×8 Transposition and Multiplication

Bit Matrix Multiplication instructions have initially been introduced in the Cray Y-MP family of supercomputers [4], and the usefulness of such instructions has recently been assessed for RISC ISA design [3]. The Kalray kv3 core includes instructions that perform bit-matrix multiplication and transposition on 64-bit words considered as 8-bit×8-bit matrices. Precisely, let a 64-bit unsigned integer composed of bytes $b_7 \dots b_0$, where b_7 is the most significant byte and b_0 the least significant byte. This integer is interpreted as the following bit matrix:

$$b_7 \dots b_0 \equiv \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} b_0^0 & b_0^1 & b_0^2 & b_0^3 & b_0^4 & b_0^5 & b_0^6 & b_0^7 \\ b_1^0 & b_1^1 & b_1^2 & b_1^3 & b_1^4 & b_1^5 & b_1^6 & b_1^7 \\ b_2^0 & b_2^1 & b_2^2 & b_2^3 & b_2^4 & b_2^5 & b_2^6 & b_2^7 \\ b_3^0 & b_3^1 & b_3^2 & b_3^3 & b_3^4 & b_3^5 & b_3^6 & b_3^7 \\ b_4^0 & b_4^1 & b_4^2 & b_4^3 & b_4^4 & b_4^5 & b_4^6 & b_4^7 \\ b_5^0 & b_5^1 & b_5^2 & b_5^3 & b_5^4 & b_5^5 & b_5^6 & b_5^7 \\ b_6^0 & b_6^1 & b_6^2 & b_6^3 & b_6^4 & b_6^5 & b_6^6 & b_6^7 \\ b_7^0 & b_7^1 & b_7^2 & b_7^3 & b_7^4 & b_7^5 & b_7^6 & b_7^7 \end{pmatrix}$$

In particular, given that the Kalray kv3 core is little-endian (like the Intel ia32) the encoding of the identity 8×8 bit matrix in a 64-bit word is `0x8040201008040201`.

Let `bmt8x8` be the bit matrix transposition on 8-bit×8-bit matrices. The effects of this function, whose C implementation appears in Figure 8, are defined as follows:

$$\texttt{bmt8x8}\left(\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix}\right) = \begin{pmatrix} b_0^0 & b_1^0 & b_2^0 & b_3^0 & b_4^0 & b_5^0 & b_6^0 & b_7^0 \\ b_0^1 & b_1^1 & b_2^1 & b_3^1 & b_4^1 & b_5^1 & b_6^1 & b_7^1 \\ b_0^2 & b_1^2 & b_2^2 & b_3^2 & b_4^2 & b_5^2 & b_6^2 & b_7^2 \\ b_0^3 & b_1^3 & b_2^3 & b_3^3 & b_4^3 & b_5^3 & b_6^3 & b_7^3 \\ b_0^4 & b_1^4 & b_2^4 & b_3^4 & b_4^4 & b_5^4 & b_6^4 & b_7^4 \\ b_0^5 & b_1^5 & b_2^5 & b_3^5 & b_4^5 & b_5^5 & b_6^5 & b_7^5 \\ b_0^6 & b_1^6 & b_2^6 & b_3^6 & b_4^6 & b_5^6 & b_6^6 & b_7^6 \\ b_0^7 & b_1^7 & b_2^7 & b_3^7 & b_4^7 & b_5^7 & b_6^7 & b_7^7 \end{pmatrix}$$

Let `bmm8x8` be the bit matrix multiplication of 8-bit×8-bit matrices. The effects of this function, whose C implementation also appears in Figure 8, are defined as follows:

Kalray – all rights reserved

```
/** Bit matrix transpose 8x8 */
uint64_t
bmt8x8(uint64_t a)
{
  int i, j;
  uint64_t b = 0;
  for (i = 0; i < 8; i++) {
    for (j = 0; j < 8; j++) {
      if (a & 0x1) {
        b ^= (uint64_t)0x1 << (j*8) +
             i;
      }
      a >>= 1;
    }
  }
  return b;
}
```

```
/** Bit matrix multiply 8x8 */
uint64_t
bmm8x8(uint64_t a, uint64_t b)
{
  int i, j;
  uint64_t c = 0;
  for (i = 0; i < 8; i++) {
    for (j = 0; j < 8; j++) {
      if (a & 0x1) {
        c ^= ((b>>(j*8))&0xFF) << i
             *8;
      }
      a >>= 1;
    }
  }
  return c;
}
```

Figure 8: Reference C codes for the `bmt8x8` and `bmm8x8` functions.

$$\texttt{bmm8x8}\left(\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}, \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix}\right) = \begin{pmatrix} a_0^0 b_0 \otimes a_0^1 b_1 \otimes a_0^2 b_2 \otimes a_0^3 b_3 \otimes a_0^4 b_4 \otimes a_0^5 b_5 \otimes a_0^6 b_6 \otimes a_0^7 b_7 \\ a_1^0 b_0 \otimes a_1^1 b_1 \otimes a_1^2 b_2 \otimes a_1^3 b_3 \otimes a_1^4 b_4 \otimes a_1^5 b_5 \otimes a_1^6 b_6 \otimes a_1^7 b_7 \\ a_2^0 b_0 \otimes a_2^1 b_1 \otimes a_2^2 b_2 \otimes a_2^3 b_3 \otimes a_2^4 b_4 \otimes a_2^5 b_5 \otimes a_2^6 b_6 \otimes a_2^7 b_7 \\ a_2^0 b_0 \otimes a_2^1 b_1 \otimes a_2^2 b_2 \otimes a_2^3 b_3 \otimes a_2^4 b_4 \otimes a_2^5 b_5 \otimes a_2^6 b_6 \otimes a_2^7 b_7 \\ a_4^0 b_0 \otimes a_4^1 b_1 \otimes a_4^2 b_2 \otimes a_4^3 b_3 \otimes a_4^4 b_4 \otimes a_4^5 b_5 \otimes a_4^6 b_6 \otimes a_4^7 b_7 \\ a_5^0 b_0 \otimes a_5^1 b_1 \otimes a_5^2 b_2 \otimes a_5^3 b_3 \otimes a_5^4 b_4 \otimes a_5^5 b_5 \otimes a_5^6 b_6 \otimes a_5^7 b_7 \\ a_6^0 b_0 \otimes a_6^1 b_1 \otimes a_6^2 b_2 \otimes a_6^3 b_3 \otimes a_6^4 b_4 \otimes a_6^5 b_5 \otimes a_6^6 b_6 \otimes a_6^7 b_7 \\ a_7^0 b_0 \otimes a_7^1 b_1 \otimes a_7^2 b_2 \otimes a_7^3 b_3 \otimes a_7^4 b_4 \otimes a_7^5 b_5 \otimes a_7^6 b_6 \otimes a_7^7 b_7 \end{pmatrix}$$

Here, $\otimes$ denotes the exclusive or operation applied on 8-bit operands. Each operand takes the form $a_i^j b_j$, which applies the logical and of bit a_i^j to each of the 8 bits of b_j.

8.1.2 Kalray kv3 Bit Matrix 8×8 Instructions

The Kalray kv3 core provides bit matrix multiplication instructions named `sbmm8` for Swapped Bit Matrix Multiply 8×8 and `sbmmt8` for Swapped Bit Matrix Multiply Transposed 8×8. These instructions are defined as follows:

$$\texttt{sbmm8}(a,b) = \texttt{bmm8x8}(b,a)$$
$$\texttt{sbmmt8}(a,b) = \texttt{bmt8x8}(\texttt{bmm8x8}(b,a))$$

The reason for swapping the bit matrix multiplication operands in the Kalray kv3 core instructions is motivated by the fact that immediate value operands are always the rightmost at the assembly language level. However, the majority of cases where an immediate value is useful in bit matrix multiplication is as the first (leftmost) operands.

Kalray – all rights reserved

The Kalray kv3 core support immediate operands up to 64-bit for ALU instructions. This yields two variants of each bit matrix multiply instruction, for instance in case of `sbmm8`:

```
sbmm8 $r0 =$r2, $r3
;;
sbmm8 $r4 =$r5, 0x8040201008040201
;;
```

At the C source level, the bit matrix multiplications and other special instructions of the Kalray kv3 core are available as *intrinsic functions*, whose use can be optimized by the compiler like regular arithmetic instructions. The C prototypes for the bit matrix multiplications builtins are:

```
long long __builtin_kvx_sbmm8d(long long, long long);
long long __builtin_kvx_sbmmt8d(long long, long long);
```

8.1.3 Byte and Half-Word Manipulations

A bit matrix multiplication 8×8 can be interpreted as constructing each byte of the 64-bit result by ANDing each byte of the rightmost operand b with a bit of the leftmost operand a, and XORing the results. Whenever each byte a_i of operand a contain only one bit set, this bit selects the byte of operand b to be written. For instance, to perform a byte swap of a 64-bit word, we use the constant `0x0102040810204080` for operand a:

$$\texttt{0x0102040810204080} \equiv \begin{pmatrix} \texttt{0x80} \\ \texttt{0x40} \\ \texttt{0x20} \\ \texttt{0x10} \\ \texttt{0x08} \\ \texttt{0x04} \\ \texttt{0x02} \\ \texttt{0x01} \end{pmatrix} = \begin{pmatrix} 1 << 7 \\ 1 << 6 \\ 1 << 5 \\ 1 << 4 \\ 1 << 3 \\ 1 << 2 \\ 1 << 1 \\ 1 << 0 \end{pmatrix} \equiv \begin{pmatrix} \text{Select byte 7} \\ \text{Select byte 6} \\ \text{Select byte 5} \\ \text{Select byte 4} \\ \text{Select byte 3} \\ \text{Select byte 2} \\ \text{Select byte 1} \\ \text{Select byte 0} \end{pmatrix}$$

With the `sbmm8` instruction, the first operand of the bit matrix multiplication 8×8 appears on the right hand side, so we can take advantage of the 64-bit immediate constants:

```
sbmm8 $r0 =$r1, 0x0102040810204080  # Byte swap in 64-bit word
;;
```

In Table 16, we collect some useful immediate constants of the `sbmm8` instruction that achieve common byte and half-word manipulations.

8.1.4 Parallel Bit Manipulations

Another interpretation of the bit matrix multiplication 8×8 is that each bit a_i^j of the leftmost operand a is contributed to byte i of the result according to the bit pattern supplied by byte b_j of the rightmost operand b. The same contribution pattern applied in parallel to the eight bytes of operand a. For instance, a parallel bit reverse of each byte of a is achieved when the constant `0x0102040810204080` is supplied as operand b:

Kalray – all rights reserved

Constant	Effects
0x8040201008040201	Identity bit matrix
0x0000000001020408	Byte swap in 32-bit value
0x0000000002010804	Half swap in 32-bit value
0x0102040810204080	Byte swap in 64-bit value
0x0201080420108040	Half swap in 64-bit value
0x0804020180402010	Word swap in 64-bit value
0x8008400420021001	Byte mix in 64-bit value
0x8020080240100401	Reverse byte mix in 64-bit value
0x4020100804020180	Rotate left 8 bits in 64-bit value
0x2010080402018040	Rotate left 16 bits in 64-bit value
0x1008040201804020	Rotate left 24 bits in 64-bit value
0x0101010101010101	Splat byte 0 to 64-bit value
0x0201020102010201	Splat half 0 to 64-bit value
0x0804020108040201	Splat word 0 to 64-bit value

Table 16: Immediate constants for byte and half-word manipulations with the `sbmm8` instruction.

$$\texttt{0x0102040810204080} \equiv \begin{pmatrix} \texttt{0x80} \\ \texttt{0x40} \\ \texttt{0x20} \\ \texttt{0x10} \\ \texttt{0x08} \\ \texttt{0x04} \\ \texttt{0x02} \\ \texttt{0x01} \end{pmatrix} = \begin{pmatrix} 1 << 7 \\ 1 << 6 \\ 1 << 5 \\ 1 << 4 \\ 1 << 3 \\ 1 << 2 \\ 1 << 1 \\ 1 << 0 \end{pmatrix} \equiv \begin{pmatrix} \text{Copy bit 0 to bit 7} \\ \text{Copy bit 1 to bit 6} \\ \text{Copy bit 2 to bit 5} \\ \text{Copy bit 3 to bit 4} \\ \text{Copy bit 4 to bit 3} \\ \text{Copy bit 5 to bit 2} \\ \text{Copy bit 6 to bit 1} \\ \text{Copy bit 7 to bit 0} \end{pmatrix}$$

Because the Kalray kv3 core only allows immediate operands in the leftmost position, the immediate constant must be created first in a register:

```
make $r1 = 0x0102040810204080
;;
sbmm8 $r0 = $r1, $r2
;;
```

Bit reversing is more useful when applied to data formats wider than a byte. In such cases, the parallel bit-reverse in a byte can be combined with a byte reverse in a double word, yielding the following expression for bit-reversing a 64-bit (unsigned) integer:

$$\begin{aligned} \text{bit-reverse}(a) \quad = \quad & \text{bmm8x8}(\quad \texttt{0x0102040810204080}, \\ & \qquad \text{bmm8x8}(a, \texttt{0x0102040810204080})) \end{aligned}$$

Kalray – all rights reserved

Constant	Multiplication matrix
0x00	0x0000000000000000
0x01	0x8040201008040201
0x02	0x4020108884028180
0x03	0xc06030988c068381
0x04	0x201088c44281c040
0x05	0xa050a8d44a85c241
0x07	0xe070b85cce8743c1
0x09	0x90c8e472a9c46221
0x0b	0xd0e8f4fa2dc6e3a1
0x0d	0xb0d86cb6eb45a261
0x0e	0x70b85c2e674321e0

Table 17: Matrices for multiplications by constants less than 16 in AES GF(2^8). Note the matrix for the multiplication by 0 and the identity matrix for the multiplication by 1.

In FFT applications, bit reversing is usually applied to a range of `l2n` bits inside a 32-bit (unsigned) integer, where `l2n` is the base-2 logarithm of the number of points of the FFT. This leads to the following `bitrev` implementation:

```
static inline unsigned
bitrev(unsigned in, unsigned l2n)
{
  unsigned long long cst = 0x0102040810204080ULL;
  unsigned long long r1 = __builtin_kvx_sbmm8d(cst, in);
  unsigned long long r2 = __builtin_kvx_sbmm8d(r1, cst);
  return r2 >> (64 - l2n);
}
```

As a side note, one should be aware that radix-4 of higher radix Cooley-Tukey in-place FFT implementations can be adapted without run-time cost to used bit-reversing instead of digit reversing, thanks to the observation published in 1988 by C. S. Burrus.

8.1.5 Multiplication by Constants in GF(2^8)

In general, multiplication in a GF(2^m) field under a polynomial representation requires that a carry-less multiplication is performed, followed by computing the remainder modulo the irreducible polynomial. The remainder computation amounts to canceling the high-order bits of the carry-less multiplication result by xoring them with the shifted values of the irreducible polynomial until all bits at position greater than or equal to n are zero.

Whenever one of the multiplication operands is constant, it is possible to replace this sequence of steps by a simple $n \times n$ bit matrix multiplication. Precisely, given the irreducible polynomial of a polynomial basis of GF(2^m) and the constant operand, the Mastrovito algorithm [5] computes the bit matrix entries. Details of the matrix generation are discussed for instance by Paar in [6].

Table 17 provides an examples of matrices for multiplication by constants less than 16 over the AES GF(2^8), whose irreducible polynomial is `0x11b`. Those matrices may be directly passed

Kalray – all rights reserved

to the `sbmm8` instruction, but the data requires preparation. Bits from all bytes need to be reorganized, so all bits 0 appear in one byte of the data and so on. This operation is easily performed using matrix transposition. Of course, the transposition needs to be reversed afterward.

On the Kalray kv3 core, this operation may be performed using two instructions (assuming that the input value comes in the `$r0` register, output is available in the same register):

```
sbmmt8 $r1 = $r0, 0x8040201008040201
;;
sbmmt8 $r0 = $r1, 0x4020108884028180
;;
```

The first instruction provides only transposition (the multiplication is performed by the identity matrix). The second instruction multiplies by 2 over AES GF(2^8) and performs the output transposition using one instruction only.

The possible uses of this method include AES (Advanced Encryption Standard) algorithm, which uses multiplications by `0x02` and `0x03` for in MixColumns, and multiplications by `0x9`, `0x0b`, `0x0d`, `0x0e` in InvMixColumns. Another example is the SHA-3 hash candidate function, Groestl, which performs multiplications by `0x02`, `0x03`, `0x04`, `0x05` and `0x07`.

8.1.6 Basis Change in GF(2^8)

Bit matrix multiplication may be also used for basis changes in GF(2^8). Changes of basis may lead to easier computations, for instance of the inverse in GF(2^8) as shown by Canright [2]. On the Kalray kv3 core, we can change the basis of eight elements in GF(2^8) using a single `sbmm8` instruction.

8.1.7 Conversions for Bit Slicing

Bit-slicing is a technique that exploits parallelism available at the bit level when operating words with logical operations. Once a function is decomposed to logic equations, it is possible to compute w instances of it in parallel on a core whose word size is w bits. This techniques was first applied by Biham for the DES [1]. Given the w independent instances of the input data, a bit-sliced implementation requires that all bits from the same positions be packed together. This reordering of bits may be performed using the bit matrix transpose operation.

The Kalray kv3 core, efficient reordering of bits is achieved when eight bytes are processed by a single `sbmmt8` instruction. Then the results of multiple such operations may be combined whenever the w values of interest are multiple of 8. For instance, if we want to prepare 16 bytes to bit-sliced operations (16-bit bit-slicing), we can first perform transposition of the original values and then join the pairs of two 8-bit values into 16-bit ones.

FIXME! This is for k1b. Let us assume that we have the input in registers `$r0, $r1, $r2` and `$r3`. In the first step we transpose them in pairs:

```
sbmmt8 $r2:$r4 = $r2:$r3, 0x10204080:0x01020408
;;
sbmmt8 $r3:$r0 = $r0:$r1, 0x10204080:0x01020408
```

In the second step we join results from the two pairs using the matrix `0x8008400420021001`:

Kalray – all rights reserved

```
    sbmm8 $r3:$r2 = $r2:$r3, 0x20021001:0x80084004
    ;;
    sbmm8 $r1:$r0 = $r4:$r0, 0x20021001:0x80084004
```

Bibliography

[1] Eli Biham. A Fast New DES Implementation in Software. In *FSE '97: Proc. of the 4th International Workshop on Fast Software Encryption*, pages 260–272, 1997.

[2] David Canright. A Very Compact Rijndael Sbox (revised). Technical Report NPS-MA-05-001, Naval Postgraduate School, 2005. Available: http://web.nps.navy.mil/~dcanrig/pub/NPS-MA-05-001.pdf.

[3] Yedidya Hilewitz, Cedric Lauradoux, and Ruby B. Lee. Bit Matrix Multiplication in Commodity Processors. In *Proc. of 19th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 08)*, July 2008 2008.

[4] William Lee, Gary J. Geissler, Steven J. Johnson, and Alan J. Schiffleger. Vector Bit-Matrix Multiply Functional Unit. United States Patent number 5170370, 1992.

[5] Edoardo D. Mastrovito. VLSI Designs for Multiplication Over Finite Fields GF(2m). In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 297–309, 1989.

[6] Christof Paar. A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields. *IEEE Trans. Comput.*, 45(7):856–861, 1996.

Kalray – all rights reserved

Instruction index

abdbo, 463
abdd, 465
abdhq, 470
abdsbo, 472
abdsd, 474
abdshq, 476
abdsw, 478
abdswp, 480
abdubo, 482
abdud, 484
abduhq, 486
abduw, 488
abduwp, 490
abdw, 492
abdwp, 495
absbo, 1461
absd, 1462
abshq, 1463
abssbo, 1464
abssd, 1465
absshq, 1466
abssw, 1467
absswp, 1468
absw, 1469
abswp, 1470
acswapd, 160
acswapq, 163
acswapw, 166
addbo, 497
addcd, 499
addcd.i, 501
addd, 503
addhq, 508
addrbod, 510
addrhqd, 511
addrwpd, 512
addsbo, 513
addsd, 515
addshq, 517
addsw, 519
addswp, 521
addurbod, 523
addurhqd, 524
addurwpd, 525
addusbo, 526
addusd, 528
addushq, 530
addusw, 532
adduswp, 534
adduwd, 536
addw, 538
addwd, 541
addwp, 543
addx16bo, 545
addx16d, 547
addx16hq, 549
addx16uwd, 551
addx16w, 553
addx16wd, 555
addx16wp, 557
addx2bo, 559
addx2d, 561
addx2hq, 563
addx2uwd, 565
addx2w, 567
addx2wd, 569
addx2wp, 571
addx32d, 573
addx32uwd, 575
addx32w, 577
addx32wd, 579
addx4bo, 581
addx4d, 583
addx4hq, 585
addx4uwd, 587
addx4w, 589
addx4wd, 591
addx4wp, 593
addx64d, 595
addx64uwd, 597
addx64w, 599
addx64wd, 601
addx8bo, 603
addx8d, 605
addx8hq, 607
addx8uwd, 609
addx8w, 611
addx8wd, 613
addx8wp, 615
aladdd, 169
aladdw, 172
alclrd, 175
alclrw, 178
ald, 181
alw, 184
andd, 617
andnd, 622
andnw, 627
andrbod, 630
andrhqd, 631
andrwpd, 632
andw, 633
asd, 187
asw, 190
avgbo, 636
avghq, 638
avgrbo, 640
avgrhq, 642
avgrubo, 644
avgruhq, 646
avgruw, 648
avgruwp, 650
avgrw, 652
avgrwp, 654
avgubo, 656
avguhq, 658
avguw, 660
avguwp, 662
avgw, 664
avgwp, 666
await, 114

barrier, 115
break, 116

call, 117

Kalray – all rights reserved

©2025 Kalray

cb, 118
cbsd, 668
cbsw, 669
cbswp, 670
clrf, 671
clsd, 672
clsw, 673
clswp, 674
clzd, 675
clzw, 676
clzwp, 677
cmovebo, 678
cmoved, 681
cmovehq, 685
cmovewp, 687
cmuldt, 1076
cmulghxdt, 1077
cmulglxdt, 1078
cmulgmxdt, 1079
cmulxdt, 1080
compd, 689
compnbo, 693
compnd, 695
compnhq, 697
compnw, 699
compnwp, 701
compuwd, 703
compw, 705
compwd, 707
copyd, 1471
copyo, 193
copyq, 1081
copyw, 1472
crcbellw, 1082
crcbelmw, 1084
crclellw, 1086
crclelmw, 1088
ctzd, 709
ctzw, 710
ctzwp, 711

d1inval, 194
dflushl, 195
dflushsw, 199
dinvall, 200
dinvalsw, 204
dot2suwd, 1090
dot2suwdp, 1091
dot2uwd, 1092
dot2uwdp, 1093
dot2w, 1094
dot2wd, 1095
dot2wdp, 1096
dot2wzp, 1097
dpurgel, 205
dpurgesw, 209
dtouchl, 210

eord, 712
eorrbod, 717
eorrhqd, 718
eorrwpd, 719
eorw, 720
errop, 119
extfs, 723
extfz, 724

fabsd, 1196
fabshq, 1197
fabsw, 1198
fabswp, 1199
faddd, 1200
fadddc, 1526
fadddc.c, 1201
fadddp, 1202
faddho, 1203
faddhq, 1204
faddw, 1205
faddwc, 1527
faddwc.c, 1206
faddwcp, 1528
faddwcp.c, 1207
faddwp, 1208
faddwq, 1209
fcdivd, 1210
fcdivw, 1211
fcdivwp, 1212
fcompd, 1213
fcompnd, 1215
fcompnhq, 1217
fcompnw, 1219
fcompnwp, 1221
fcompw, 1223
fdot2w, 1225
fdot2wd, 1226
fdot2wdp, 1227
fdot2wzp, 1228
fence, 214
ffdmasw, 1229
ffdmaswp, 1230
ffdmaswq, 1231
ffdmaw, 1232
ffdmawp, 1233
ffdmawq, 1234
ffdmdaw, 1235
ffdmdawp, 1236
ffdmdawq, 1237
ffdmdsw, 1238
ffdmdswp, 1239
ffdmdswq, 1240
ffdmsaw, 1241
ffdmsawp, 1242
ffdmsawq, 1243
ffdmsw, 1244
ffdmswp, 1245
ffdmswq, 1246
ffmad, 1247
ffmaho, 1248
ffmahq, 1249
ffmahw, 1250
ffmahwq, 1251
ffmaw, 1252
ffmawc, 1253
ffmawcp, 1255
ffmawd, 1257
ffmawdp, 1258
ffmawp, 1259
ffmawq, 1260
ffmsd, 1261
ffmsho, 1262
ffmshq, 1263
ffmshw, 1264

Kalray – all rights reserved

ffmshwq, 1265
ffmsw, 1266
ffmswc, 1267
ffmswcp, 1269
ffmswd, 1271
ffmswdp, 1272
ffmswp, 1273
ffmswq, 1274
fixedd, 1275
fixedud, 1276
fixeduw, 1277
fixeduwp, 1278
fixedw, 1279
fixedwp, 1280
floatd, 1281
floatud, 1282
floatuw, 1283
floatuwp, 1284
floatw, 1285
floatwp, 1286
fmaxd, 1287
fmaxhq, 1288
fmaxw, 1289
fmaxwp, 1290
fmind, 1291
fminhq, 1292
fminw, 1293
fminwp, 1294
fmm212w, 1295
fmm222w, 1296
fmma212w, 1297
fmma222w, 1298
fmms212w, 1300
fmms222w, 1301
fmuld, 1303
fmulho, 1304
fmulhq, 1305
fmulhw, 1306
fmulhwq, 1307
fmulw, 1308
fmulwc, 1309
fmulwcp, 1310
fmulwd, 1312
fmulwdp, 1313
fmulwp, 1314
fmulwq, 1315
fnarrowdw, 1316
fnarrowdwp, 1317
fnarrowwh, 1318
fnarrowwhq, 1319
fnegd, 1320
fneghq, 1321
fnegw, 1322
fnegwp, 1323
frecw, 1324
frsrw, 1325
fsbfd, 1326
fsbfdc, 1529
fsbfdc.c, 1327
fsbfdp, 1328
fsbfho, 1329
fsbfhq, 1330
fsbfw, 1331
fsbfwc, 1530
fsbfwc.c, 1332
fsbfwcp, 1531
fsbfwcp.c, 1333
fsbfwp, 1334
fsbfwq, 1335
fsdivd, 1336
fsdivw, 1337
fsdivwp, 1338
fsrecd, 1339
fsrecw, 1340
fsrecwp, 1341
fsrsrd, 1342
fsrsrw, 1343
fsrsrwp, 1344
fwidenlhw, 1345
fwidenlhwp, 1346
fwidenlwd, 1347
fwidenmhw, 1348
fwidenmhwp, 1349
fwidenmwd, 1350

get, 120
goto, 121
i1inval, 215
i1invals, 216
icall, 122
iget, 123
igoto, 124
insf, 725
iord, 726
iornd, 731
iornw, 736
iorrbod, 739
iorrhqd, 740
iorrwpd, 741
iorw, 742

landd, 745
landw, 746
lbs, 220
lbz, 227
ld, 234
lhs, 241
lhz, 248
liord, 748
liorw, 749
lnandd, 751
lnandw, 752
lniord, 754
lniorw, 755
lnord, 1473
lnorw, 1474
lo, 255
loopdo, 125
lord, 1475
lorw, 1476
lq, 265
lws, 272
lwz, 279

maddd, 1098
madddt, 1100
maddhq, 1101
maddhwq, 1103
maddmwq, 1104
maddsudt, 1105
maddsuhwq, 1106

Kalray – all rights reserved

©2025 Kalray

maddsumwq, 1107
maddsuwd, 1108
maddsuwdp, 1110
maddudt, 1112
madduhwq, 1113
maddumwq, 1114
madduwd, 1115
madduwdp, 1117
madduzdt, 1119
maddw, 1120
maddwd, 1122
maddwdp, 1124
maddwp, 1126
maddwq, 1128
make, 757
maxbo, 760
maxd, 762
maxhq, 767
maxrbod, 769
maxrhqd, 770
maxrwpd, 771
maxubo, 772
maxud, 774
maxuhq, 779
maxurbod, 781
maxurhqd, 782
maxurwpd, 783
maxuw, 784
maxuwp, 787
maxw, 789
maxwp, 792
minbo, 794
mind, 796
minhq, 801
minrbod, 803
minrhqd, 804
minrwpd, 805
minubo, 806
minud, 808
minuhq, 813
minurbod, 815
minurhqd, 816
minurwpd, 817
minuw, 818
minuwp, 821
minw, 823
minwp, 826
mm212w, 1129
mma212w, 1131
mms212w, 1133
msbfd, 1135
msbfdt, 1137
msbfhq, 1138
msbfhwq, 1140
msbfmwq, 1141
msbfsudt, 1142
msbfsuhwq, 1143
msbfsumwq, 1144
msbfsuwd, 1145
msbfsuwdp, 1147
msbfudt, 1149
msbfuhwq, 1150
msbfumwq, 1151
msbfuwd, 1152
msbfuwdp, 1154
msbfuzdt, 1156
msbfw, 1157
msbfwd, 1159
msbfwdp, 1161
msbfwp, 1163
msbfwq, 1165
muld, 1166
muldt, 1168
mulhq, 1169
mulhwq, 1171
mulmwq, 1172
mulsudt, 1173
mulsuhwq, 1174
mulsumwq, 1175
mulsuwd, 1176
mulsuwdp, 1178
muludt, 1180
muluhwq, 1181
mulumwq, 1182
muluwd, 1183
muluwdp, 1185
mulw, 1187
mulwd, 1189
mulwdp, 1191
mulwp, 1193
mulwq, 1195

nandd, 828
nandw, 833
negbo, 1477
negd, 1478
neghq, 1479
negsbo, 1480
negsd, 1481
negshq, 1482
negsw, 1483
negswp, 1484
negw, 1485
negwp, 1486
neord, 836
neorw, 841
niord, 844
niorw, 849
nop, 852
nord, 1487
norw, 1489
notd, 1490
notw, 1491
nxord, 1492
nxorw, 1494

ord, 1495
ornd, 1497
ornw, 1499
orrbod, 1500
orrhqd, 1501
orrwpd, 1502
orw, 1503

pcrel, 853

ret, 126
rfe, 127
rolw, 856
rolwps, 858
rorw, 860
rorwps, 862
rswap, 128

Kalray – all rights reserved

sb, 286
sbfbo, 864
sbfcd, 866
sbfcd.i, 868
sbfd, 870
sbfhq, 875
sbfsbo, 877
sbfsd, 879
sbfshq, 881
sbfsw, 883
sbfswp, 885
sbfusbo, 887
sbfusd, 889
sbfushq, 891
sbfusw, 893
sbfuswp, 895
sbfuwd, 897
sbfw, 899
sbfwd, 902
sbfwp, 904
sbfx16bo, 906
sbfx16d, 908
sbfx16hq, 910
sbfx16uwd, 912
sbfx16w, 914
sbfx16wd, 916
sbfx16wp, 918
sbfx2bo, 920
sbfx2d, 922
sbfx2hq, 924
sbfx2uwd, 926
sbfx2w, 928
sbfx2wd, 930
sbfx2wp, 932
sbfx32d, 934
sbfx32uwd, 936
sbfx32w, 938
sbfx32wd, 940
sbfx4bo, 942
sbfx4d, 944
sbfx4hq, 946
sbfx4uwd, 948
sbfx4w, 950
sbfx4wd, 952
sbfx4wp, 954
sbfx64d, 956
sbfx64uwd, 958
sbfx64w, 960
sbfx64wd, 962
sbfx8bo, 964
sbfx8d, 966
sbfx8hq, 968
sbfx8uwd, 970
sbfx8w, 972
sbfx8wd, 974
sbfx8wp, 976
sbmm8, 1504
sbmm8d, 978
sbmmt8, 1506
sbmmt8d, 983
scall, 130
sd, 293
set, 132
sh, 300
sleep, 135
sllbos, 988
slld, 990
sllhqs, 992
sllw, 994
sllwps, 996
slsbos, 998
slsd, 1000
slshqs, 1002
slsw, 1004
slswps, 1006
slusbos, 1008
slusd, 1010
slushqs, 1012
slusw, 1014
sluswps, 1016
so, 307
sq, 317
srabos, 1018
srad, 1020
srahqs, 1022
sraw, 1024
srawps, 1026
srlbos, 1028
srld, 1030
srlhqs, 1032
srlw, 1034
srlwps, 1036
srsbos, 1038
srsd, 1040
srshqs, 1042
srsw, 1044
srswps, 1046
stop, 136
stsud, 1048
stsuhq, 1050
stsuw, 1052
stsuwp, 1054
sw, 324
sxbd, 1508
sxhd, 1509
sxlbhq, 1056
sxlhwp, 1057
sxmbhq, 1058
sxmhwp, 1059
sxwd, 1510
syncgroup, 137

tlbdinval, 138
tlbiinval, 139
tlbprobe, 140
tlbread, 141
tlbwrite, 142

waitit, 143
wfxl, 144
wfxm, 146

xaccesso, 148
xaligno, 154
xandno, 1351
xando, 1352
xclampwo, 1353
xcopyo, 1354
xcopyv, 1355
xcopyx, 1356
xeoro, 1358
xffma44hw, 1359
xfmaxhx, 1361

Kalray – all rights reserved

©2025 Kalray

xfminhx, 1362
xfmma484hw, 1363
xfnarrow44wh, 1366
xfscalewo, 1368
xiorno, 1369
xioro, 1370
xlo, 331
xmadd44bw0, 1371
xmadd44bw1, 1372
xmaddifwo, 1373
xmaddsu44bw0, 1374
xmaddsu44bw1, 1375
xmaddu44bw0, 1376
xmaddu44bw1, 1377
xmma4164bw, 1378
xmma484bw, 1380
xmmasu4164bw, 1382
xmmasu484bw, 1384
xmmau4164bw, 1386
xmmau484bw, 1388
xmmaus4164bw, 1390
xmmaus484bw, 1392
xmovefd, 1394
xmovefo, 1395
xmovefq, 1396
xmovetd, 1060
xmovetq, 1064
xmsbfifwo, 1397
xmt44d, 1520
xnando, 1398
xneoro, 1399
xnioro, 1400
xnoro, 1521
xnxoro, 1522
xord, 1511
xorno, 1523
xoro, 1524
xorrbod, 1513
xorrhqd, 1514
xorrwpd, 1515
xorw, 1516
xplb, 348
xpld, 366
xplh, 384
xplo, 402
xplq, 420
xplw, 438
xrecvo, 1066
xsbmm8dq, 1401
xsbmmt8dq, 1402
xsendo, 1067
xsendrecvo, 1068
xso, 456
xsplatdo, 1069
xsplatov, 1403
xsplatox, 1404
xsx48bw, 1406
xtrunc48wb, 1407
xxoro, 1525
xzx48bw, 1408

zxbd, 1517
zxhd, 1518
zxlbhq, 1072
zxlhwp, 1073
zxmbhq, 1074
zxmhwp, 1075
zxwd, 1519

Kalray – all rights reserved