ACKNOWLEDGEMENT

By utilizing this website and/or documentation, I hereby acknowledge as follows:

Effective October 1, 2012, QUALCOMM Incorporated completed a corporate reorganization in which the assets of certain of its businesses and groups, as well as the stock of certain of its direct and indirect subsidiaries, were contributed to Qualcomm Technologies, Inc. (QTI), a wholly-owned subsidiary of QUALCOMM Incorporated that was created for purposes of the reorganization.

Qualcomm Technology Licensing (QTL), the Company's patent licensing business, continues to be operated by QUALCOMM Incorporated, which continues to own the vast majority of the Company's patent portfolio. Substantially all of the Company's products and services businesses, including QCT, as well as substantially all of the Company's engineering, research and development functions, are now operated by QTI and its direct and indirect subsidiaries[1]. Neither QTI nor any of its subsidiaries has any right, power or authority to grant any licenses or other rights under or to any patents owned by QUALCOMM Incorporated.

No use of this website and/or documentation, including but not limited to the downloading of any software, programs, manuals or other materials of any kind or nature whatsoever, and no purchase or use of any products or services, grants any licenses or other rights, of any kind or nature whatsoever, under or to any patents owned by QUALCOMM Incorporated or any of its subsidiaries. A separate patent license or other similar patent-related agreement from QUALCOMM Incorporated is needed to make, have made, use, sell, import and dispose of any products or services that would infringe any patent owned by QUALCOMM Incorporated in the absence of the grant by QUALCOMM Incorporated of a patent license or other applicable rights under such patent.

Any copyright notice referencing QUALCOMM Incorporated, Qualcomm Incorporated, QUALCOMM Inc., Qualcomm Inc., Qualcomm or similar designation, and which is associated with any of the products or services businesses or the engineering, research or development groups which are now operated by QTI and its direct and indirect subsidiaries, should properly reference, and shall be read to reference, QTI.

[1] The products and services businesses, and the engineering, research and development groups, which are now operated by QTI and its subsidiaries include, but are not limited to, QCT, Qualcomm Mobile & Computing (QMC), Qualcomm Atheros (QCA), Qualcomm Internet Services (QIS), Qualcomm Government Technologies (QGOV), Corporate Research & Development, Qualcomm Corporate Engineering Services (QCES), Office of the Chief Technology Officer (OCTO), Office of the Chief Scientist (OCS), Corporate Technical Advisory Group, Global Market Development (GMD), Global Business Operations (GBO), Qualcomm Ventures, Qualcomm Life (QLife), Quest, Qualcomm Labs (QLabs), Snaptracs/QCS, Firethorn, Qualcomm MEMS Technologies (QMT), Pixtronix, Qualcomm Innovation Center (QuIC), Qualcomm iSkoot, Qualcomm Poole and Xiam.

Hexagon V2 Programmer's Reference

Manual

80-NB419-1 Rev. A

August 4, 2011

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

Contents

Figures

Tables

1 Introduction

1.1 Overview

The Qualcomm Hexagon™ processor is a general-purpose digital signal processor architecture designed for high performance and low power across a wide variety of multimedia and modem applications. V2 is the second version of the Hexagon processor architecture.

This chapter provides an introduction to the following topics:

- Hexagon processor features
- Processor functional units
- Instruction set
- Processor versions
- Software development tools

The chapter concludes with an overview of the manual organization and a description of the terminology and notations used in the manual. It also explains how to contact Qualcomm with any comments or suggestions.

1.2 Features

- **Memory**

  Program code and data memories are stored in a unified 32-bit address space. The load/store architecture features a complete set of addressing modes which efficiently support both compiler code generation and DSP application programming.

- **Registers**

  Thirty two 32-bit general purpose registers can be accessed as single registers or as 64-bit register pairs. The general registers hold all data including scalar, pointer, and packed vector data.

- **Data types**

  Instructions can perform a wide variety of fixed-point operations on scalar or vector data. Scalar instructions operate on 16-, 32-, or 64-bit data. Vector instructions operate on 64-bit vectors of words, halfwords, or bytes.

- **Parallel execution**

  Instructions can be grouped into very long instruction word (VLIW) packets for parallel execution, with each packet containing from one to four instructions. Vector instructions operate on single instruction multiple data (SIMD) vectors.

- **Program flow**

  Nestable zero-overhead hardware loops are supported. Conditional/unconditional jumps and subroutine calls support both PC-relative and register indirect addressing.

- **Instruction pipeline**

  Pipeline hazards are resolved by the hardware: instruction scheduling is not constrained by pipeline restrictions.

- **Cache memory**

  Memory accesses can be cached or uncached. Separate L1 instruction and data caches exist for program code and data. A unified L2 cache can be partly or wholly configured as tightly-coupled memory (TCM).

- **Virtual memory**

  Memory is addressed virtually, with virtual to physical memory mapping handled by the OS. Virtual memory supports the implementation of memory management and memory protection in a hardware-independent manner.

1.3 Functional units

Figure 1-1 shows the major functional units of the Hexagon V2 processor architecture:

- Memory and registers
- Instruction sequencer
- Execution units
- Load and load/store units

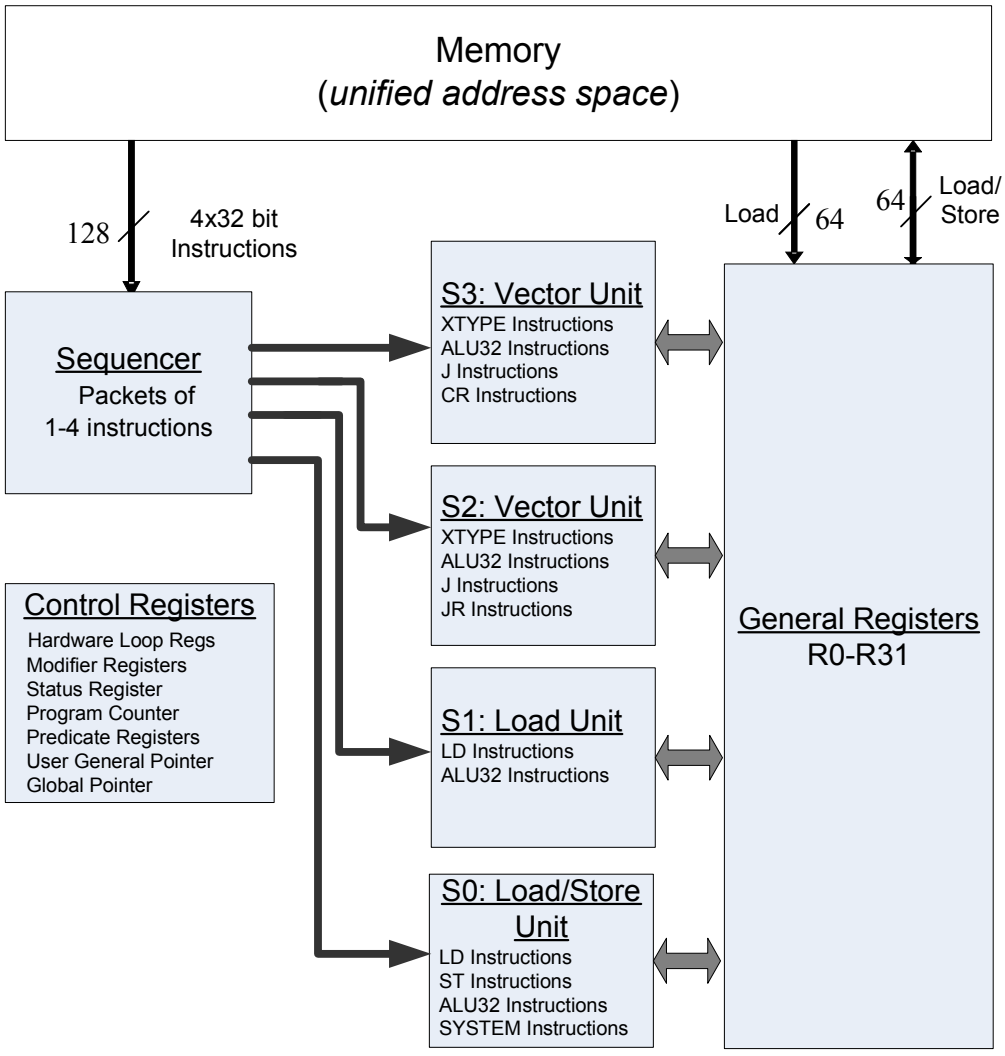


Figure 1-1 Hexagon V2 processor architecture

1.3.1 Memory

The Hexagon processor features a unified byte-addressable memory. This memory has a single 32-bit address space that holds both instructions and data. It operates in little-endian mode.

1.3.2 Registers

The Hexagon processor has two sets of registers: general registers and control registers.

The general registers include thirty-two 32-bit registers which can be accessed either as single registers or as aligned 64-bit register pairs. The general registers are used to contain all pointer, scalar, vector, and accumulator data.

The control registers include special-purpose registers such as program counter, status register, loop registers, etc.

1.3.3 Sequencer

The instruction sequencer processes packets of one to four instructions in each cycle. If a packet contains more than one instruction, the instructions are executed in parallel.

The instruction combinations allowed in a packet are limited to the instruction types that can be executed in parallel in the four execution units (as shown in Figure 1-1).

1.3.4 Vector units

The dual vector execution units are identical: each include a 64-bit shifter and a vector multiply/accumulate unit with four 16x16 multipliers to support scalar and vector instructions.

These units also perform 32- and 64-bit ALU instructions, and jump and loop instructions.

1.3.5 Load units

The Hexagon processor uses a load-store architecture: data must be explicitly loaded from memory to registers before it can operated on.

The two load units – one of which also supports store operations – can operate on signed or unsigned bytes, halfwords (16-bit), words (32-bit), or double words (64-bit).

To increase the number of instruction combinations allowed in packets, the load units also support 32-bit ALU instructions.

1.4 Instruction set

In order for the Hexagon processor to achieve large amounts of work per cycle, the instruction set was designed with the following properties:

- Static grouping (VLIW) architecture
- Extensive compound instructions
- A large set of SIMD and application-specific instructions

To support efficient compilation, the instruction set is designed to be orthogonal with respect to registers, addressing modes, and load/store access size.

1.4.1 Addressing modes

The Hexagon processor supports the following memory addressing modes:

- Global pointer relative
- Indirect
- Indirect with offset
- Indirect with auto-increment (immediate or register)
- Circular with auto-increment (immediate or register)
- Bit-reversed with auto-increment register

For example:

```
R2 = memw(GP+#200)
R2 = memw(R1)
R2 = memw(R3+#100)
R2 = memw(R3++#4)
R2 = memw(R0++M1)
R0 = memw(R2++#8:circ(M0))
R0 = memw(R2++I:circ(M0))
R2 = memw(R0++M1:brev)
```

Auto-increment with register addressing uses one of two dedicated address-modify registers (which are part of the control registers).

NOTE Because circular and bit-reversed addressing do not map directly to any C language construct, they are supported in C through the use of processor-specific intrinsics.

Atomic memory operations (load locked/store conditional) are supported to implement multi-thread synchronization.

1.4.2 Scalar operations

The Hexagon processor includes the following scalar operations on fixed-point data:

- Multiplication of 16-bit, 32-bit, and complex data
- Addition and subtraction of 16-, 32-, and 64-bit data (with and without saturation)
- Logical operations on 32- and 64-bit data (And, Or, XOR, Not)
- Shifts on 32- and 64-bit data (arithmetic and logical)
- Min/max, negation, absolute value, parity, norm, swizzle
- Compares of 8-, 16-, 32-, and 64-bit data
- Sign and zero extension (8- and 16- to 32-bit, 32- to 64-bit)
- Bit manipulation
- Predicate operations

1.4.3 Vector operations

The Hexagon processor includes the following vector operations on fixed-point data:

- Multiplication (halfwords, word by half, vector reduce, dual multiply)
- Addition and subtraction of word and halfword data
- Shifts on word and halfword data (arithmetic and logical)
- Min/max, average, negative average, absolute difference, absolute value
- Compares of word and halfword data
- Reduce, sum of absolute differences on unsigned bytes
- Special-purpose data arrangement (such as pack, splat, shuffle, align, saturate, splice, truncate, complex conjugate, complex rotate, zero extend)

NOTE Certain vector operations support automatic scaling, saturation, and rounding.

For example, the following instruction performs a vector operation:

```
R1:0 += vrmpyh(R3:2,R5:4)
```

It is defined to perform the following operations in one cycle:

```
R1:0 += ( (R2.L * R4.L) +
          (R2.H * R4.H) +
          (R3.L * R5.L) +
          (R3.H * R5.H)
        )
```

Figure 1-2 shows a schematic of this instruction type.

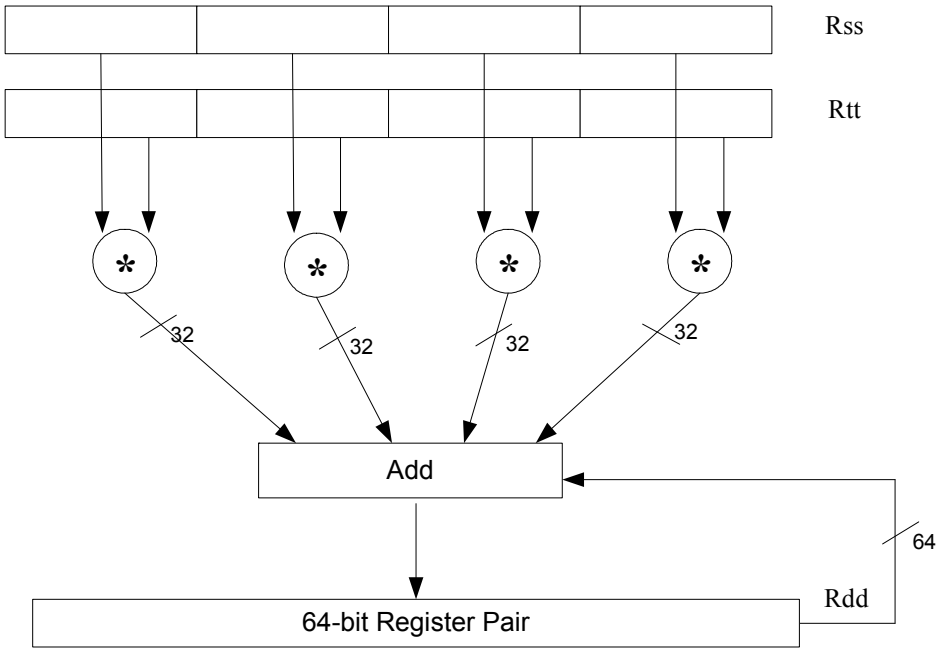


Figure 1-2 Vector instruction example

1.4.4 Program flow

The Hexagon processor supports zero-overhead hardware loops. For example:

```
  loop0(start,#3)       // loop 3 times
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

The loop instructions support nestable loops, with few restrictions on their use.

Software branches use a predicated branch mechanism. Explicit compare instructions generate a predicate bit, which is then tested by conditional branch instructions. For example:

```
P1 = cmp.eq(R2, R3)
if (P1) jump end
```

Jumps and subroutine calls can be conditional or unconditional, and support both PC-relative and register indirect addressing modes. For example:

```
jump end
jumpr R1
call function
callr R2
```

The subroutine call instructions store the return address in register `R31`. Subroutine returns are performed using a jump indirect instruction through this register. For example:

```
jumpr R31     // subroutine return
```

1.4.5 Instruction packets

Sequences of instructions can be explicitly grouped into packets for parallel execution. For example:

```
{
    R8 = memh(R3++#2)
    R12 = memw(R1++#4)
    R15:14 += vdmpy(R11:10,R7:6):<<1:sat
    R7:6 = valignb(R9:8,R7:6,#2)
}
```

Brace characters are used to delimit the start and end of an instruction packet.

Packets can be from one to four instructions long. Packets of varying length can be freely mixed in a program.

Packets have various restrictions on the allowable instruction combinations. The primary restriction is determined by the instruction class of the instructions in a packet.

1.4.6 Instruction classes

The instructions are assigned to specific instruction classes. Classes are important for two reasons:

- Only certain combinations of instructions can be written in parallel (as shown in Figure 1-1), and the allowable combinations are specified by instruction class.
- Instruction classes logically correspond with instruction types, so they serve as mnemonics for looking up specific instructions.

Figure 1-3 presents an overview of the instruction classes and how they can be grouped together.

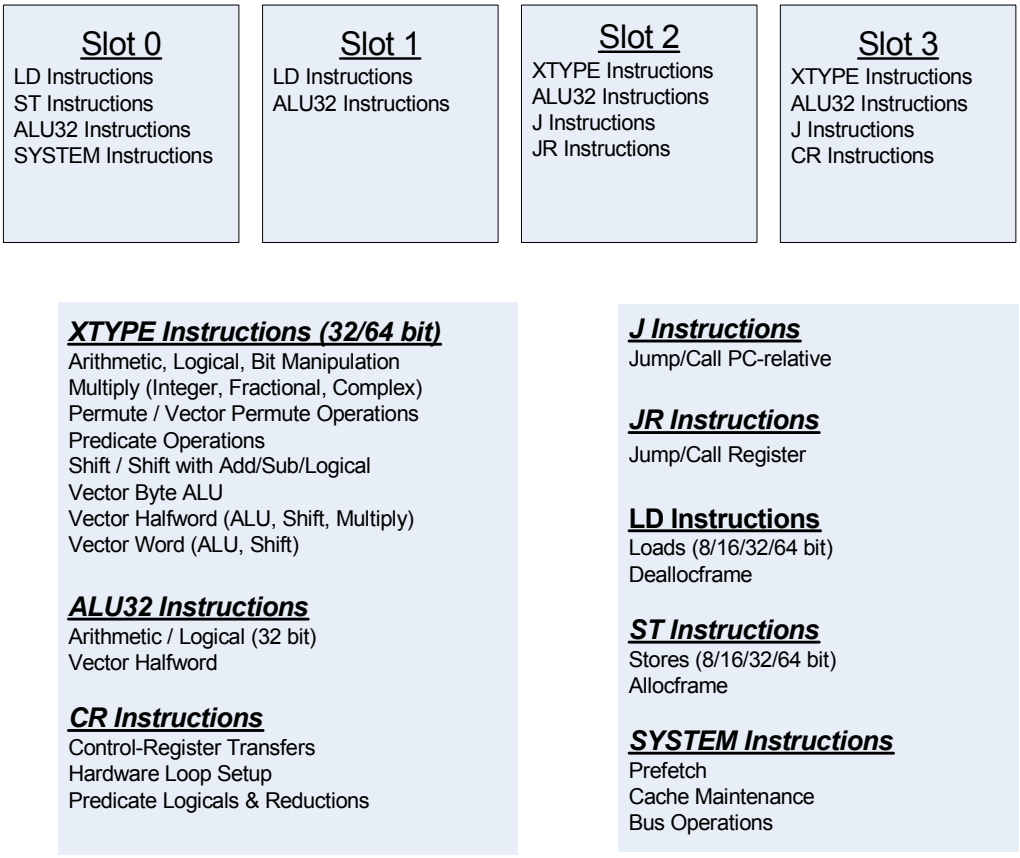


Figure 1-3 Instruction classes and combinations

1.4.7 Instruction intrinsics

To support efficient coding of the time-critical sections of a program (without resorting to assembly language), the C compilers support intrinsics which are used to directly access Hexagon processor instructions from within C code. For example:

```
int main()
{
    long long v1 = 0xFFFF0000FFFF0000;
    long long v2 = 0x0000FFFF0000FFFF;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are defined for most of the Hexagon processor instructions.

1.5 Processor versions

The Hexagon V2 processor is the second version of the Hexagon processor architecture.

The V1 processor was designed as a multi-media applications DSP. In V2 the architecture is extended to achieve the following goals:

- Lower power consumption
- Improved support for modem applications
- Improved support for OS
- Improved efficiency of control code

1.6 Software development tools

The Hexagon processor includes a complete set of tools for developing reliable high-performance software. The tool set includes the following tools:

- ANSI-compliant C and C++ compilers
- Powerful macro assembler
- Run-time debugger
- Instruction set simulator
- Execution profiler
- GNU-based utilities

1.7 Using the manual

This manual describes the Hexagon processor architecture and instruction set.

- Chapter 1, *Introduction*, presents an overview of the Hexagon processor and the manual.
- Chapter 2, *Registers*, describes the processor registers.
- Chapter 3, *Instructions*, describes instructions and instruction packets.
- Chapter 4, *Data Processing*, presents an overview of the data-processing operations.
- Chapter 5, *Memory*, describes the memory address space and addressing modes.
- Chapter 6, *Conditional Execution*, describes the conditional execution model.
- Chapter 7, *Program Flow*, describes instructions and events which affect the program flow, including loops, jumps, and subroutine calls.
- Chapter 8, *Call Stack*, describes the software stack and stack operations.
- Chapter 9, *Implementation-Specific Information*, describes information specific to the current implementation of the Hexagon V2 processor.

- Chapter 10, *Instruction Encoding*, describes the encoding of instruction packets and operands.
- Chapter 11, *Instruction Set*, describes the processor instruction set.

1.8 Notation

This section presents the notational conventions used in this document to describe Hexagon processor instructions:

- Instruction syntax
- Register operands
- Numeric operands

NOTE The notation described here does not appear in actual assembly language instructions. It is used only to specify the instruction syntax and behavior.

1.8.1 Instruction syntax

The following notation is used to describe the syntax of instructions:

- `This font is used for instructions`
- Square brackets enclose optional items (e.g., [`:sat`], means that saturation is optional)
- Braces are used to indicate a choice of items (e.g., {`Rs,#s16`}, means that either Rs or a signed 16-bit immediate can be used)

1.8.2 Register operands

The following notation is used to describe register operands in the syntax and behavior of instructions:

R*ds*[.*elst*]

The *ds* field indicates the register operand type and bit size (as defined in Table 1-1).

Table 1-1 Register symbols

Symbol	Operand Type	Size (in Bits)
d	Destination	32
dd		64
s	First source	32
ss		64
t	Second source	32
tt		64

Table 1-1 Register symbols

Symbol	Operand Type	Size (in Bits)
u	Third source	32
uu		64
x	Source *and* destination	32
xx		64

Examples of *ds* field (describing instruction syntax):

```
Rd = neg(Rs)            // Rd -> 32-bit dest, Rs 32-bit source
Rd = xor(Rs,Rt)         // Rt -> 32-bit second source
Rx = insert(Rs,Rtt)     // Rx -> both source and dest
```

Examples of *ds* field (describing instruction behavior):

```
Rdd = Rss + Rtt         // Rdd, Rss, Rtt -> 64-bit registers
```

The optional *elst* field (short for "element size and type") is used to specify parts of a register when the register is used as a vector. It can specify the following values:

- A signed or unsigned byte, halfword, or word within the register (as defined in Figure 1-4)
- A bit-field within the register (as defined in Table 1-2)

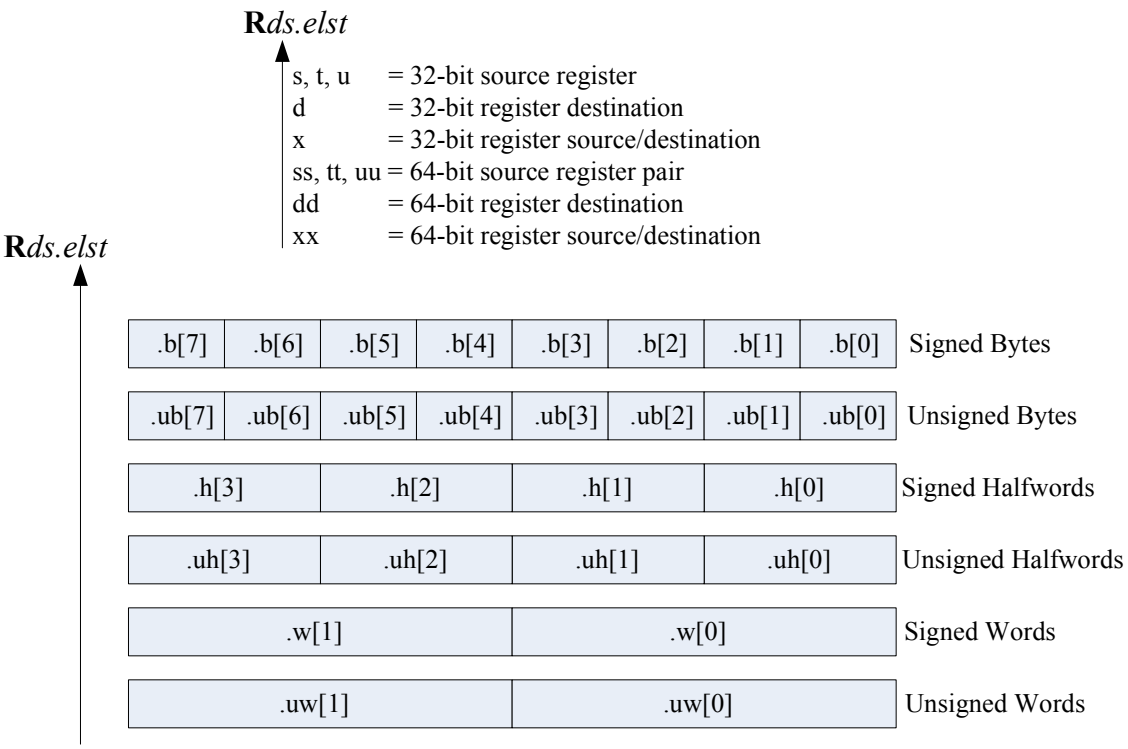


Figure 1-4 Register field symbols

Table 1-2 Register bit field symbols

Symbol	Meaning
.sN	Bits [N-1:0] are treated as a N-bit signed number. For example, R0.s16 means that the least significant 16-bits of R0 are treated as a 16-bit signed number.
.uN	Bits [N-1:0] are treated as a N-bit unsigned number.
.H	The most-significant 16 bits of a 32-bit register.
.L	The least-significant 16 bits of a 32-bit register.

Examples of *elst* field:

```
EA = Rt.h[1]                // .h[1] -> bit field 31:16 in Rt
Pd = (Rss.u64 > Rtt.u64)    // .u64  -> unsigned 64-bit value
Rd = mpyu(Rs.L,Rt.H)        // .L/.H -> low/high 16-bit fields
```

NOTE The control and predicate registers use the same notation as the general registers, but are written as `Cx` and `Px` (respectively) instead of `Rx`.

1.8.3 Numeric operands

Table 1-3 describes the notation used to describe numeric operands in the syntax and behavior of instructions:

Table 1-3 Instruction operands

Symbol	Meaning	Min	Max
#uN	Unsigned N-bit immediate value	0	2^N-1
#sN	Signed N-bit immediate value	-2^{N-1}	$2^{N-1}-1$
#mN	Signed N-bit immediate value	$-(2^{N-1}-1)$	$2^{N-1}-1$
#uN:S	Unsigned N-bit immediate value representing integral multiples of 2^S in specified range	0	$(2^N-1) \times 2^S$
#sN:S	Signed N-bit immediate value representing integral multiples of 2^S in specified range	$(-2^{N-1}) \times 2^S$	$(2^{N-1}-1) \times 2^S$
#rN:S	Same as #sN:S, but value is offset from PC of current packet	$(-2^{N-1}) \times 2^S$	$(2^{N-1}-1) \times 2^S$
usat_N	Saturate value to unsigned N-bit number	0	2^N-1
sat_N	Saturate value to signed N-bit number	-2^{N-1}	$2^{N-1}-1$

`#uN`, `#sN`, and `#mN` are used to specify immediate operands in instructions. The `#` symbol appears in the actual instruction to indicate the immediate operand.

`#rN` is used to specify loop and branch destinations in instructions. In this case the `#` symbol does *not* appear in the actual instruction; instead, the entire `#rN` symbol (including its `:S` suffix) is expressed as a loop or branch symbol whose numeric value is determined by the assembler and linker. For example:

```
call my_proc            // instruction example
```

The `:S` suffix indicates that the `S` least-significant bits in a value are implied zero bits and therefore not encoded in the instruction. The implied zero bits are called *scale bits*.

For example, `#s4:2` denotes a signed immediate operand represented by four bits encoded in the instruction, and two scale bits. The possible values for this operand are -32, -28, -24, -20, -16, -12, -8, -4, 0, 4, 8, 12, 16, 20, 24, and 28.

Examples of operand symbols:

```
Rd = add(Rs,#s16)       // #s16   -> signed 16-bit imm value
Rd = memw(Rs++#s4:2)    // #s4:2  -> scaled signed 4-bit imm value
call #r22:2            // #r22:2 -> scaled 22-bit PC-rel addr value
```

1.9 Terminology

Table 1-4 lists the symbols used in Hexagon processor instruction names to specify the supported data types.

Table 1-4 Data symbols

Size	Symbol	Type
8-bit	B	Byte
8-bit	UB	Unsigned Byte
16-bit	H	Half word
16-bit	UH	Unsigned Half word
32-bit	W	Word
32-bit	UW	Unsigned Word
64-bit	D	Double word

2 Registers

2.1 Overview

This chapter describes the Hexagon processor registers:

- General registers
- Control registers

General registers are used for all general-purpose computation including address generation and scalar and vector arithmetic.

Control registers support special-purpose processor features such as hardware loops and predicates.

2.2 General registers

The Hexagon processor has thirty-two 32-bit general-purpose registers (named `R0` through `R31`). These registers are used to store operands in virtually all the instructions:

- Memory addresses for load/store instructions
- Data operands for arithmetic/logic instructions
- Vector operands for vector instructions

For example:

```
R1 = memh(R0)              // Load from R0 into R1
R4 = add(R2,R3)            // Add
R28 = vaddh(R11,R10)       // Vector add halfword
```

Figure 2-1 shows the general registers.

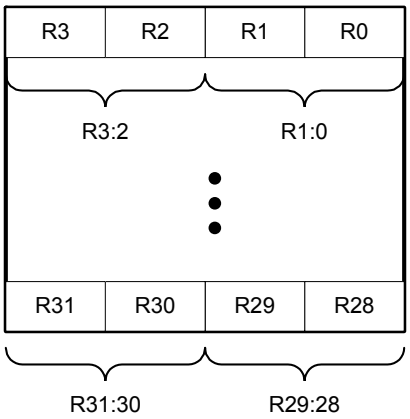


Figure 2-1 General registers

Aliased registers

Three of the general registers – `R29` through `R31` – are used to support subroutines (Section NOTE) and the software stack (Chapter 8). These registers are modified implicitly by the subroutine and stack instructions. They have symbol aliases which are used to indicate when these registers are being accessed as subroutine and stack registers.

For example:

```
SP = add(SP, #-8)    // SP is alias of R29
allocframe           // Modifies SP (R29) and FP (R30)
call init            // Modifies LR (R31)
```

Table 2-1 lists the aliased registers.

Table 2-1 Aliased registers

Register	Alias	Name	Description
R29	SP	Stack pointer	Points to topmost element of stack in memory.
R30	FP	Frame pointer	Points to current procedure frame on stack. Used by external debuggers to examine the stack and determine call sequence, parameters, local variables, etc.
R31	LR	Link register	Stores return address of a subroutine call.

Register pairs

The general registers can be specified as register pairs which represent a single 64-bit register. For example:

```
R1:0 = memd(R3)                    // Load doubleword
R7:6 = valignb(R9:8,R7:6, #2)      // Vector align
```

NOTE The first register in a register pair must always be odd-numbered, and the second must be the next lower register.

Table 2-2 lists the valid register pairs.

Table 2-2 Register pairs

Register	Register Pair
R0	R1:0
R1	
R2	R3:2
R3	
R4	R5:4
R5	
R6	R7:6
R7	
...	
R24	R25:24
R25	
R26	R27:26
R27	
R28	R29:28
R29 (SP)	
R30 (FP)	R31:30 (LR:FP)
R31 (LR)	

2.3 Control registers

The Hexagon processor includes a set of 32-bit control registers which provide access to processor features such as the program counter, hardware loops, and vector predicates.

Unlike general registers, control registers can be used as instruction operands only in the following cases:

- Instructions that require a specific control register as an operand
- Register transfer instructions

For example:

```
R2 = memw(R0++M1)    // Use auto-increment register (M1)
R9 = PC              // Get program counter (PC)
LC1 = R3             // Set hardware loop count (LC1)
```

NOTE When a control register is used in a register transfer, the other operand must be a general register.

Figure 2-2 shows the control registers.

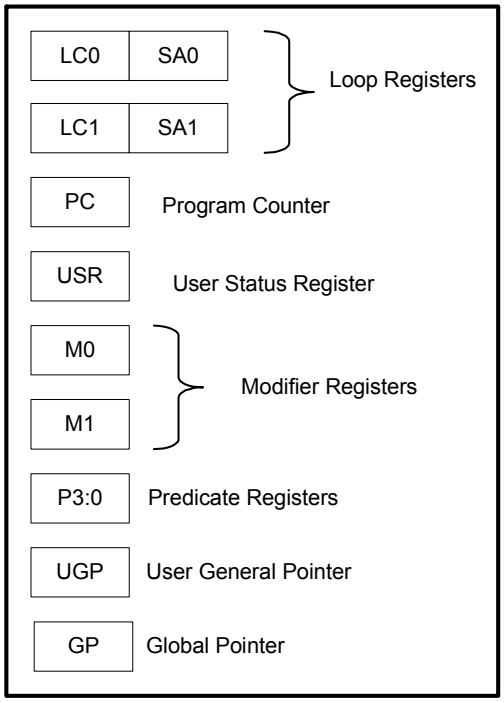


Figure 2-2 Control registers

Aliased registers

The control registers have numeric aliases (C0 through C11).

Table 2-3 lists the aliased control registers.

Table 2-3 Aliased control registers

Register	Alias	Name
SA0	C0	Loop start address register 0
LC0	C1	Loop count register 0
SA1	C2	Loop start address register 1
LC1	C3	Loop count register 1
P3:0	C4	Predicate registers 3:0
reserved	C5	–
M0	C6	Modifier register 0
M1	C7	Modifier register 1
USR	C8	User status register
PC	C9	Program counter
UGP	C10	User general pointer
GP	C11	Global pointer

NOTE The control register numbers (0-11) are used to specify the control registers in instruction encodings (Chapter 10).

2.3.1 Program counter

The Program Counter (PC) register points to the next instruction packet to execute (Section 3.4). It is modified implicitly by instruction execution, but can be read directly. For example:

```
R7 = PC       // Get program counter
```

NOTE The PC register is read-only: writing to it has no effect.

2.3.2 Loop registers

The Hexagon processor includes two sets of loop registers to support nested hardware loops (Section 7.3). Each hardware loop is implemented with a pair of registers containing the loop count and loop start address. The loop registers are modified implicitly by the `loop` instruction, but can also be accessed directly. For example:

```
loop0(start, R4)  // Modifies LC0 and SA0 (LC0=R4, SA0=&start)
LC1 = R22         // Set loop1 count
R9 = SA1          // Get loop1 start address
```

Table 2-4 lists the loop registers.

Table 2-4 Loop registers

Register	Name	Description
LC0, LC1	Loop count	Number of loop iterations to execute.
SA0, SA1	Loop start address	Address of first instruction in loop.

2.3.3 User status register

The user status register (USR) stores the status results that are set by executing certain instructions. For example:

```
R9:8 = vaddw(R9:8, R3:2):sat    // Vector add words
R6 = USR                        // Get saturation status
```

USR stores the following status values:

- Hardware Loop Configuration (Section 7.3)
- Sticky Saturation Overflow (Section 4.3.2)

Table 2-5 defines the user status register.

Table 2-5 User status register

Name	R/W	Bits	Field	Description
USR		32		User Status Register
	R	31:10	reserved	Return 0 if read. Reserved for future expansion. To remain forward-compatible with future processor versions, software should always write this field with the same value read from the field.
	R/W	9:8	LPCFG	Hardware Loop Configuration. Number of loop iterations (0-3) remaining before pipeline predicate should be set.
	R	7:1	reserved	Return 0 if read. Reserved for future expansion. To remain forward-compatible with future processor versions, software should always write this field with the same value read from the field.
	R/W	0	OVF	Sticky Saturation Overflow. 1: Saturation occurred 0: No saturation Set when saturation occurs while executing instruction that specifies optional saturation. Remains set until explicitly cleared by a USR = Rs instruction.

2.3.4 Modifier registers

The modifier registers (M0-M1) are used in the following addressing modes:

- Indirect auto-increment register addressing
- Circular addressing
- Bit-reversed addressing

Indirect auto-increment

In indirect auto-increment register addressing (Section 5.4.4) the modifier registers store a signed 32-bit value which specifies the increment (or decrement) value. For example:

```
M1 = R0                 // Set modifier register
R3 = memw(R2++M1)       // Load word
```

Table 2-6 defines the modifier registers as used in auto-increment register addressing.

Table 2-6 Modifier registers (indirect auto-increment addressing)

Register	Name	Description
M0, M1	Increment	Signed auto-increment value.

Circular

In circular addressing (Section 5.4.6) the modifier registers store the circular buffer length and related "K" and "I" values. For example:

```
M0 = R7                          // Set modifier register
R0 = memb(R2++#8:circ(M0))       // load from circ buffer pointed
                                 //  to by R2 with size/K vals in M0

R0 = memb(R7++I:circ(M1))        // load from circ buffer pointed
                                //  to by R7 with size/K/I vals in M1
```

Table 2-7 defines the modifier registers as used in circular addressing.

Table 2-7 Modifier registers (circular addressing)

Name	R/W	Bits	Field	Description
M0, M1		32		Circular buffer specifier.
	R/W	31:28	I[10:7]	I value (MSB - see Section 5.4.7)
	R/W	27:24	K	K value (Section 5.4.6)
	R/W	23:17	I[6:0]	I value (LSB)
	R/W	16:0	Length	Circular buffer length.

Bit-reversed

In bit-reversed addressing (Section 5.4.8) the modifier registers store a signed 32-bit value which specifies the increment (or decrement) value. For example:

```
M1 = R7                  // Set modifier register
R2 = memub(R0++M1:brev)  // The address is (R0.H | bitrev(R0.L))
                         // The orginal R0 (not reversed) is added
                         // to M1 and written back to R0
```

Table 2-8 defines the modifier registers as used in bit-reversed addressing.

Table 2-8 Modifier registers (bit-reversed addressing)

Register	Name	Description
M0, M1	Increment	Signed auto-increment value.

2.3.5 Predicate registers

The predicate registers (P0-P3) store the status results of the scalar and vector compare instructions (Chapter 6). For example:

```
P1 = cmp.eq(R2, R3)        // Scalar compare
if (P1) jump end           // Jump to address (conditional)
R8 = P1                    // Get compare status (P1 only)
P3:0 = R4                  // Set compare status (P0-P3)
```

The four predicate registers can be specified as a register quadruple (P3:0) which represents a single 32-bit register.

NOTE Unlike the other control registers, the predicate registers are only 8 bits wide because vector compares return a maximum of 8 status results.

Table 2-9 lists the predicate registers.

Table 2-9 Predicate registers

Register	Bits	Description
P0, P1, P2, P3	8	Compare status results.
P3:0	32	Compare status results.
	31:24	P3 register
	23:16	P2 register
	15:8	P1 register
	7:0	P0 register

2.3.6 User general pointer register

The user general pointer (UGP) register is a general-purpose control register. For example:

```
R9 = UGP       // Get UGP
UGP = R3       // Set UGP
```

NOTE UGP is typically used to store the address of thread local storage.

Table 2-10 defines the user general pointer register.

Table 2-10 User general pointer register

Register	Name	Description
UGP	User General Pointer	General-purpose control register.

2.3.7 Global pointer

The Global Pointer (GP) is used in GP-relative addressing. For example:

```
GP = R7                 // Set GP
R2 = memw(GP+#200)      // GP-relative load
```

Table 2-11 defines the global pointer register.

Table 2-11 Global pointer register

Name	R/W	Bits	Field	Description
GP		32		Global Pointer Register
	R/W	31:19	GDP	Global Data Pointer (Section 5.4.1).
	R	18:0	reserved	Return 0 if read. Reserved for future expansion. To remain forward-compatible with future processor versions, software should always write this field with the same value read from the field.

3 Instructions

3.1 Overview

This chapter covers the following topics:

- Instruction syntax
- Instruction classes
- Instruction packets
- Instruction intrinsics

The instruction encoding is described in Chapter 10.

For detailed descriptions of the Hexagon processor instructions see Chapter 11.

3.2 Instruction syntax

Most Hexagon processor instructions have the following syntax:

```
dest = instr_name(source1,source2,...)[:option1][:option2]...
```

The item specified on the left-hand side (LHS) of the equation is assigned the value specified by the right-hand side (RHS). For example:

```
R2 = add(R3,R1)   // Add R3 and R1, assign result to R2
```

Table 3-1 lists symbols commonly used in Hexagon processor instructions.

Table 3-1 Instruction symbols

Symbol	Example	Meaning
=	`R2 = R3`	Assignment of RHS to LHS
;	`R2 = R3; R4 = R5;`	Instruction delimiter, or end of instruction
{ … }	`{R2 = R3; R5 = R6}`	Instruction packet delimiter
(…)	`R2 = memw(R0)`	Memory address delimiter
mem*XX*	`R2 = memub(R3)`	Memory access *XX* specifies access size and type
:endloop*X*	`:endloop0`	Loop end *X* specifies loop instruction (0 or 1)
:t	`if (P0.new) jump:t target`	Direction hint (jump taken)
:nt	`if (!P1.new) jump:nt target`	Direction hint (jump not taken)
:sat	`R2 = add(R1,R2):sat`	Saturate result
:rnd	`R2 = mpy(R1.H,R2.H):rnd`	Round result
:<<16	`R2 = add(R1.L,R2.L):<<16`	Shift result left by halfword
0x	`0xBABE`	Hexadecimal number prefix

3.3 Instruction classes

The Hexagon processor instructions are assigned to specific *instruction classes*. Classes determine what combinations of instructions can be written in parallel (Section 3.4).

Instruction classes logically correspond with instruction types. For instance, the ALU32 class contains ALU instructions which operate on 32-bit operands.

Table 3-2 lists the instruction classes and subclasses.

Table 3-2 Instruction classes

Class	Subclass	Description	Section
XTYPE	–	Various operations	Section 11.10
	ALU	64-bit ALU operations	Section 11.10.1
	Bit	Bit operations	Section 11.10.2
	Complex	Complex math (using real and imaginary numbers)	Section 11.10.3
	Higher-precision Multiply	Double-precision multiply	Section 11.10.4
	Single-precision Multiply	Single-precision multiply	Section 11.10.5
	Permute	Vector permute and format conversion (pack, splat, swizzle)	Section 11.10.6
	Predicate	Predicate operations	Section 11.10.7
	Shift	Shift operations (with optional ALU operations)	Section 11.10.8
	Vector Byte	Vector byte operations	Section 11.10.9
	Vector Halfword	Vector halfword operations	Section 11.10.10
	Vector Word	Vector word operations	Section 11.10.11
ALU32	–	32-bit ALU operations	Section 11.3
	ALU	Arithmetic and logical	Section 11.3.1
	Permute	Permute	Section 11.3.2
	Conditional	Conditional operations	Section 11.3.3
	Vector Halfword	Vector halfword operations	Section 11.3.4
CR	–	Control register access, loops	Section 11.4
JR	–	Jumps (register indirect addressing mode)	Section 11.5
J	–	Jumps (PC-relative addressing mode)	Section 11.6
LD	–	Memory load operations	Section 11.7
ST	–	Memory store operations; alloc stack frame	Section 11.8
SYSTEM	–	Operating system access	Section 11.9
	USER	Application-level access	Section 11.9.1

3.4 Instruction packets

Instructions can be grouped together to form packets of independent instructions which are executed together in parallel. The packets can contain 1, 2, 3, or 4 instructions.

Instruction packets must be explicitly specified in software. They are expressed in assembly language by enclosing groups of instructions in curly braces. For example:

```
{ R0 = R1; R2 = R3 }
```

Various rules and restrictions exist on what types of instructions can be grouped together, and in what order they can appear in the packet. In particular, packet formation is subject to the following constraints:

- *Resource constraints* determine how many instructions of a specific type can appear in a packet. The Hexagon processor has a fixed number of execution units: each instruction is executed on a particular type of unit, and each unit can process at most one instruction at a time. Thus, for example, because the Hexagon processor contains only two load units, an instruction packet with three load instructions is invalid. The resource constraints are described in Section 3.4.3.
- *Grouping constraints* are a small set of rules that apply above and beyond the resource constraints. These rules are described in Section 3.4.4.
- *Dependency constraints* ensure that no write-after-write hazards exist in a packet. These rules are described in Section 3.4.5.
- *Ordering constraints* dictate the ordering of instructions within a packet. These rules are described in Section 3.4.6.
- *Alignment constraints* dictate the placement of packets in memory. These rules are described in Section 3.4.7.
- *Cache constraints* ensure that only one uncached memory access exists in a packet. This rule is described in Section 3.4.8.

NOTE Individual instructions (which are not explicitly grouped in packets) are executed by the Hexagon processor as packets containing a single instruction.

3.4.1 Packet execution semantics

With the exception of dual stores (X), packets are defined to have *parallel execution semantics*. Specifically, the execution behavior of a packet is defined as follows:

- First, all instructions in the packet read their source registers in parallel.
- Next, all instructions in the packet execute.
- Finally, all instructions in the packet write their destination registers in parallel.

For example, consider the following packet:

```
{ R2 = R3; R3 = R2; }
```

In the first phase, registers R3 and R2 are read from the register file. Then, after execution, R2 is written with the old value of R3 and R3 is written with the old value of R2. In effect, the result of this packet is that the values of R2 and R3 are swapped.

3.4.2 Sequencing semantics

Packets can be freely mixed in code. A packet is considered an atomic unit: in essence, a single large "instruction". From the program perspective a packet either executes to completion or not at all; it never executes only partially. For example, if a packet causes a memory exception, the exception point is established before the packet.

A packet containing multiple load/store instructions may require service from the external system. For instance, consider the case of a packet which performs two load operations that both miss in the cache. The packet requires the data to be supplied by the memory system:

- From the memory system perspective the two resulting load requests are processed serially.
- From the program perspective, however, both load operations must complete before the packet can complete.

Thus, the packet is atomic from the program perspective.

Packets have a single PC address which is the address of the start of the packet. Branches cannot be performed into the middle of a packet.

Architecturally, packets execute to completion – including updating all registers and memory – before the next packet begins. As a result, application programs are not exposed to any pipeline artifacts.

3.4.3 Resource constraints

A packet cannot specify more hardware resources than are physically available on the processor. For instance, because the V2 Hexagon processor has only two load units, a packet with three load instructions is invalid. The behavior of such a packet is undefined. The assembler automatically rejects packets that oversubscribe the hardware resources.

The processor supports up to four parallel instructions. The instructions are executed in four parallel pipelines which are referred to as *slots*.The four slots are labeled Slot 0, Slot 1, Slot 2, and Slot 3. (For more information see Section 1.3.)

NOTE `endloopN` instructions (Section 7.3.2) do not use any slots.

Each instruction belongs to a specific *instruction class* (Section 3.3). For example, jumps belong to instruction class J, while loads belong to instruction class LD. An instruction's class determines which slot it can execute in.

Figure 3-1 shows which instruction classes can be assigned to each of the four slots.

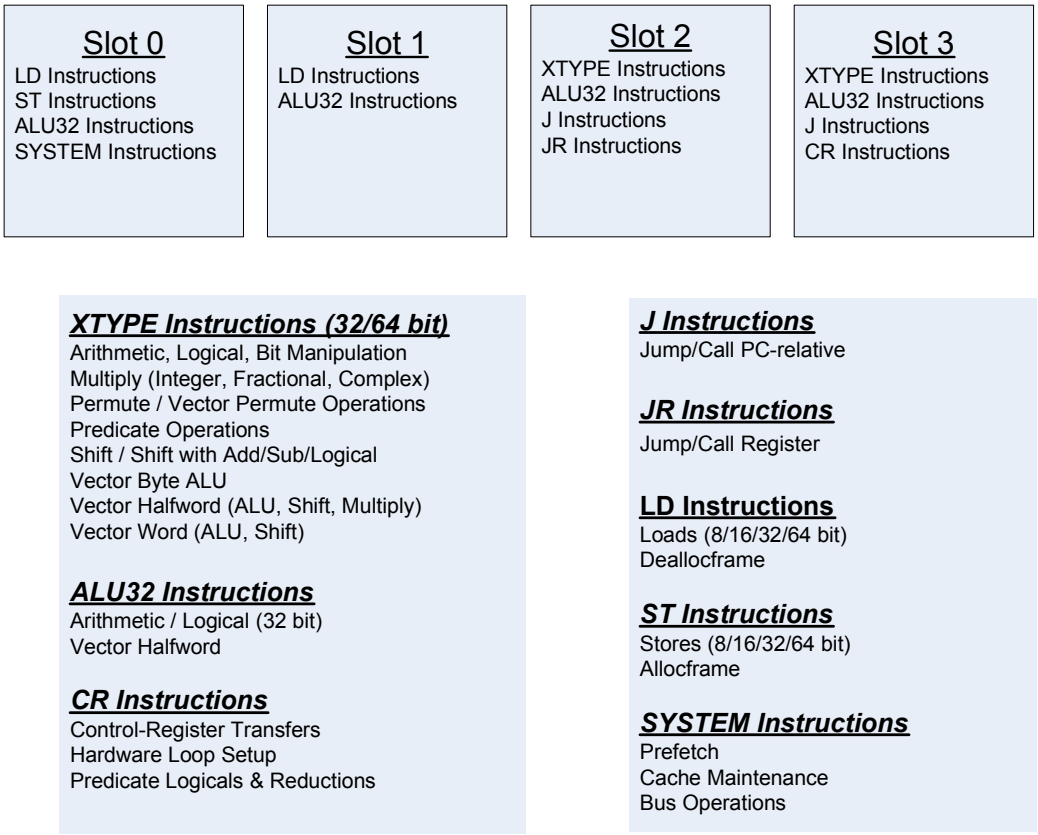


Figure 3-1 Packet grouping combinations

3.4.4 Grouping constraints

A small number of restrictions determines what constitutes a valid packet. The assembler ensures that all packets follow valid grouping rules. If a packet is executed which violates a grouping rule, the behavior is undefined. The following rules must be followed:

- Dot-new conditional instructions (Section 6.2.3) must be grouped in a packet with an instruction that generates the dot-new predicate.
- J-class instructions can be placed in Slots 2 or 3, and JR-class instructions in Slot 2. However, only one program flow instruction (J or JR) is allowed in a packet.
- A `loop` instruction cannot be placed in the same packet as a J-class instruction.
- Restrictions exist which limit the instructions that can appear in a packet at the end of a hardware loop (Section 7.3.4).
- The SYSTEM-class instructions include prefetch (`dcfetch`), cache operations, bus operations, load locked, and store conditional instructions (Section 5.6). These instructions have the following grouping rules:
  - `barrier`, `memw_locked`, `icinva`, `isync`, and `syncht` are *solo instructions*. They must not be grouped with other instructions in a packet.
  - `dccleana`, `dcinva`, `dccleaninva`, and `dczeroa` must execute on Slot 0. Slot 1 must be empty or a NOP.
  - `dcfetch` must execute on Slot 0.

3.4.5 Dependency constraints

Instructions in a packet cannot write to the same destination register. The assembler automatically flags such packets as invalid. If the processor executes a packet with two writes to the same general register, an error exception is raised.

If the processor executes a packet which performs multiple writes to the same predicate or control register, the behavior is undefined. Two special cases exist for this rule:

- Conditional writes are allowed to target the same destination register only if at most one of the writes is actually performed (Section 6.2.4).
- The overflow flag in the user status register has well-defined behavior when multiple instructions write to it (Section 4.3.2). Note that instructions that write to the entire user status register (for example, `USR=R2`) are not allowed to be grouped in a packet with any instruction that writes to a bit in the user status register.

3.4.6 Ordering constraints

In assembly code, instructions can appear in a packet in any order. The assembler automatically encodes instructions in the packet in the proper order.

In the binary encoding of a packet, the instructions must be ordered from Slot 3 down to Slot 0. If the packet contains less than four instructions, any unused slot is skipped – a NOP is unnecessary as the hardware handles the proper spacing of the instructions.

In memory, instructions in a packet must appear in monotonically decreasing slot order. Additionally, if an instruction can go in a higher-numbered slot, and that slot is empty, then it must be moved into the higher-numbered slot.

For example, if a packet contains three instructions and Slot 1 is not used, the instructions should be encoded in the packet as follows:

- Slot 3 instruction at lowest address
- Slot 2 instruction follows Slot 3 instruction
- Slot 0 instructions at the last (highest) address

NOTE The assembler properly orders instructions in a packet.

If a packet contains a single load or store instruction, that instruction must go in Slot 0, which is the highest address. As an example, a packet containing both LD and ALU32 instructions must be ordered so the LD is in Slot 0 and the ALU32 is in Slot 1.

3.4.7 Alignment constraints

Packets have the following constraints on their placement in memory:

- Packets must be word-aligned (32-bit). If the processor executes an improperly aligned packet, it will raise an error exception (Section 7.7).
- Packets should not wrap the 4GB address space. If address wraparound occurs, the processor behavior is undefined.

No other core-based restrictions exist for code placement or alignment.

If the processor branches to a packet which crosses a 16-byte address boundary, the resulting instruction fetch will stall for one processor cycle. Packets that are jump targets or loop body entries can be explicitly aligned to ensure this does not occur (Section 9.5.2).

3.4.8 Cache constraints

Packets cannot contain more than one memory access to uncached memory. If the processor executes a packet containing two load/store instructions and one or more of them accesses uncached memory, it will raise an error exception.

3.5 Instruction intrinsics

To support efficient coding of the time-critical sections of a program (without resorting to assembly language), the C compilers support intrinsics which are used to directly access Hexagon processor instructions from within C code.

The following example shows how an instruction intrinsic is used to directly access the XTYPE instruction "`Rdd = vminh(Rtt,Rss)`":

```
#include <hexagon_protos.h>

int main()
{
    long long v1 = 0xFFFF0000FFFF0000LL;
    long long v2 = 0x0000FFFF0000FFFFLL;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are provided for instructions in the following classes:

- ALU32
- XTYPE
- CR (predicate operations only)
- SYSTEM (`dcfetch` only)

For more information on intrinsics see Chapter 11.

4 Data Processing

4.1 Overview

The Hexagon processor provides a rich set of operations for processing scalar and vector data.

This chapter presents an overview of the operations provided by the following Hexagon processor instruction classes:

- XTYPE – General-purpose data operations
- ALU32 – Arithmetic/logical operations on 32-bit data

For detailed descriptions of these instruction classes see Chapter 11.

4.2 Data types

The Hexagon processor provides operations for processing the following data types:

- Fixed-point data
- Complex data
- Vector data

4.2.1 Fixed-point data

The Hexagon processor provides operations to process 8-, 16-, 32-, or 64-bit fixed-point data. The data can be either integer or fractional, and in signed or unsigned format.

4.2.2 Complex data

The Hexagon processor provides operations to process 32- or 64-bit complex data.

Complex numbers include a signed real portion and a signed imaginary portion. Given two complex numbers (a+bi) and (c+di), the complex multiply operations computes both the real portion (ac-bd) and the imaginary portion (ad+bc) in a single instruction.

Complex numbers can be packed in a general register or register pair. When packed, the imaginary portion occupies the most-significant portion of the register or register pair.

4.2.3 Vector data

The Hexagon processor provides operations to process 64-bit vector data.

Vector data types pack multiple data items – bytes, halfwords, or words – into 64-bit registers. Vector data operations are common in video and image processing.

Eight 8-bit bytes can be packed into a 64-bit register.

Figure 4-1 shows an example of a vector byte operation.

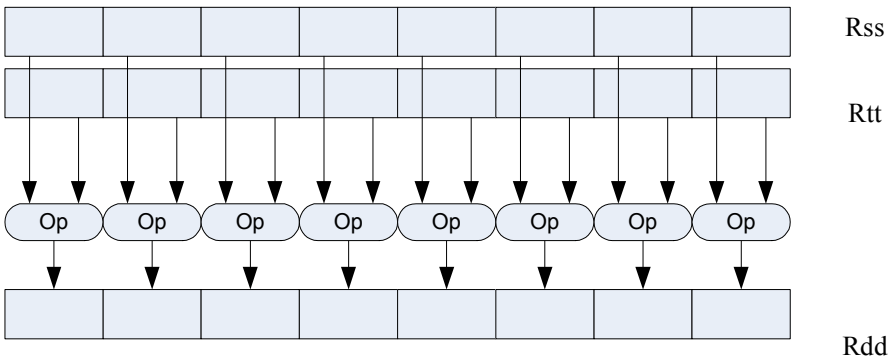


Figure 4-1 Vector byte operation

Four 16-bit halfword values are packed in a single 64-bit register pair.

Figure 4-2 shows an example of a vector halfword operation.

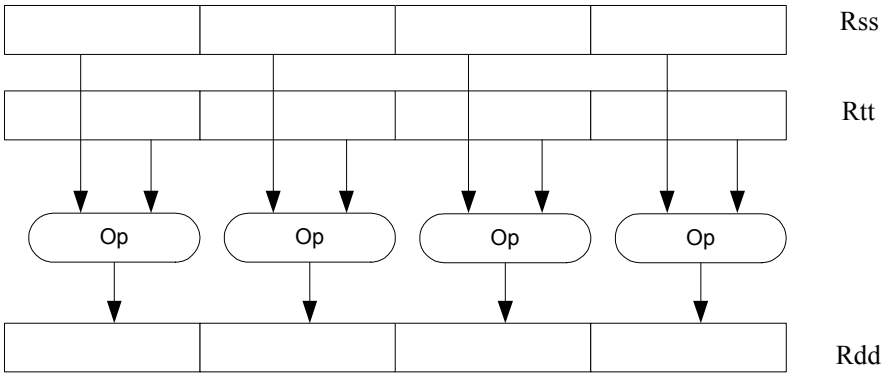


Figure 4-2 Vector halfword operation

Two 32-bit word values are packed in a single 64-bit register pair.

Figure 4-3 shows an example of a vector word operation.

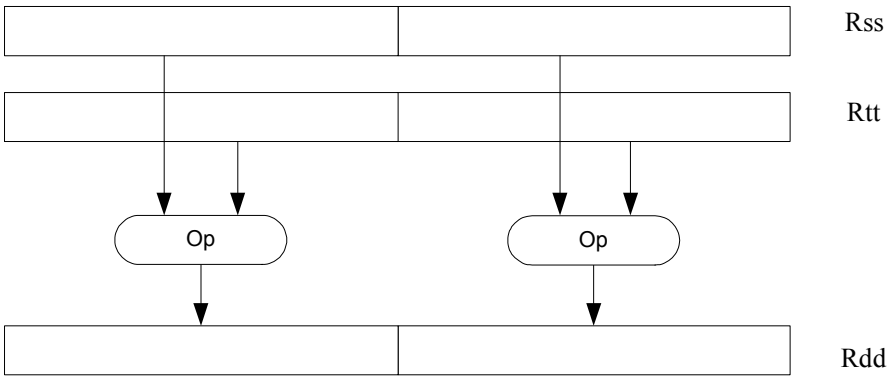


Figure 4-3 Vector word operation

4.3 Instruction options

Some instructions are available with optional scaling, saturation, and rounding. There are no mode bits controlling these options. Instructions with the options are explicitly noted. The options are described in this section.

4.3.1 Fractional scaling

In fractional data format, data is treated as fixed-point fractional values whose range is determined by the word length and radix point position.

Fractional scaling is specified in an instruction by adding the `:<<1` specifier. For example:

```
R3:2 = cmpy(R0,R1):<<1:sat
```

When two fractional numbers are multiplied, the product must be scaled to restore the original fractional data format. The Hexagon processor allows fractional scaling of the product to be specified in the instruction for shifts of 0 and 1. A shift of 1 should be done for Q1.15 numbers, while a shift of 0 should be done for integer multiplication.

4.3.2 Saturation

Certain instructions are available in saturating form. If a saturating arithmetic instruction has a result which is smaller than the minimum value, then the result is set to the minimum value. Similarly, if the operation has a result which is greater than the maximum value, then the result is set to the maximum value.

Saturation is specified in an instruction by adding the `:sat` specifier. For example:

```
R2 = abs(R1):sat
```

The OVF bit in the user status register (Section 2.3.3) is set whenever a saturating operation saturates to the maximum or minimum value. It remains set until explicitly cleared by a control register transfer to SR. For vector-type saturating operations, if any of the individual elements of the vector saturate, then OVF is set.

4.3.3 Arithmetic rounding

Certain signed multiply instructions optionally support arithmetic rounding (also known as biased rounding). The arithmetic rounding operation takes a double precision fractional value and adds 0x8000 to the low 16-bits (least significant 16-bit halfword).

Rounding is specified in an instruction by adding the `:rnd` specifier. For example:

```
R2 = mpy(R1.h,R2.h):rnd
```

NOTE Arithmetic rounding can accumulate numerical errors, especially when the number to be rounded is exactly 0.5. This happens most frequently when dividing by 2 or averaging.

4.3.4 Convergent rounding

To address the problem of error accumulation in arithmetic rounding (Section 4.3.3), the Hexagon processor includes four instructions that support positive and negative averaging with a convergent rounding option.

These instructions work as follows:

1. Compute (A+B) or (A-B) for AVG and NAVG respectively.
2. Based on the two least-significant bits of the result, add a rounding constant as follows:
   - If the two LSBs are 00, add 0
   - If the two LSBs are 01, add 0
   - If the two LSBs are 10, add 0
   - If the two LSBs are 11, add 1
3. Shift the result right by one bit.

4.4 XTYPE operations

The XTYPE instruction class includes most of the data-processing operations performed by the Hexagon processor. These operations are categorized by their operation type:

- ALU
- Bit manipulation
- Complex
- Single precision
- Double precision
- Permute
- Predicate
- Scalar shift
- Vector byte
- Vector halfword
- Vector word

4.4.1 ALU

ALU operations operate on 8-, 16-, 32-, and 64-bit data. The categories of instructions available include:

- Add and subtract with and without saturation
- Add and subtract with accumulate
- Absolute value
- Logical operations
- Min, max, negate instructions
- Register transfers of 64-bit data
- Word to doubleword sign extension
- Comparisons

For more information see Section 11.3.1 and Section 11.10.1.

4.4.2 Bit manipulation

Bit manipulation operations modify bit fields in a register or register pair. These include:

- Bit field insert
- Bit field unsigned extract
- Count leading and trailing bits
- Compare bit masks
- Set / Clear / Toggle bit
- Test bit operation
- Interleave/deinterleave bits
- Bit reverse
- Masked parity and Linear Feedback shift
- Table index formation

For more information see Section 11.10.2.

4.4.3 Complex

Many operations are available to process complex numbers. These include:

- Complex multiply with optional round and pack
- Vector complex multiply
- Vector complex conjugate
- Vector complex rotate
- Vector reduce complex multiply real or imaginary

For more information see Section 11.10.3.

4.4.4 Single precision

In single-precision arithmetic a 16-bit value is multiplied by another 16-bit value. These operands can come from the high portion or low portion of any register. Depending on the instruction, the result of the 16 × 16 operation can optionally be accumulated, saturated, rounded, or shifted left by 0-1 bits.

The instruction set supports operations on signed × signed, and unsigned × unsigned data.

Table 4-1 summarizes the options available for 16 × 16 single precision multiplications. The symbols used in the table are as follows:

- SS – Perform signed × signed multiply
- UU – Perform unsigned × unsigned multiply
- A+ – Result added to accumulator
- A- – Result subtracted from accumulator
- 0 – Result not added to accumulator

Table 4-1 Single-precision multiply options

Multiply	Result	Sign	Accumulate	Sat	Rnd	Scale
16 × 16	32	SS	A+, A-	Yes	No	0-1
16 × 16	32	SS	0	Yes	Yes	0-1
16 × 16	64	SS	A+, A-	No	No	0-1
16 × 16	64	SS	0	No	Yes	0-1
16 × 16	32	UU	A+, A-, 0	No	No	0-1
16 × 16	64	UU	A+, A-, 0	No	No	0-1

For more information see Section 11.10.5.

4.4.5 Double precision

Double precision instructions are available for both 32 × 32 and 32 × 16 multiplication:

- For 32 × 32 multiplication the result can be either 64 or 32 bits. The 32-bit result can be either the high or low portion of the 64-bit product.
- For 32 × 16 multiplication the result is always taken as the upper 32 bits.

The operands can be either signed or unsigned.

Table 4-2 summarizes the options available in double precision multiply.

Table 4-2 Double precision multiply options

Multiply	Result	Sign	Accumulate	Sat	Rnd	Scale
32 × 32	64	SS, UU	A+, A-, 0	No	No	0
32 × 32	32 (upper)	SS, UU	0	No	Yes	0
32 × 32	32 (low)	SS, UU	A+, 0	No	No	0
32 × 16	32 (upper)	SS, UU	A+, 0	Yes	Yes	0-1

For more information see Section 11.10.4.

4.4.6 Permute

Permute operations rearrange or perform format conversion on vector data types. Many types of conversions are supported:

- Swizzle bytes
- Vector shuffle
- Vector align
- Vector saturate and pack
- Vector splat bytes
- Vector splice
- Vector sign extend halfwords
- Vector zero extend bytes
- Vector zero extend halfwords
- Scalar saturate to byte, halfword, word
- Vector pack high and low halfwords
- Vector round and pack
- Vector splat halfwords

For more information see Section 11.3.2 and Section 11.10.6.

4.4.7 Predicate

Predicate operation modify predicate source data. The categories of instructions available include:

- Vector mask generation
- Predicate transfers
- Viterbi packing

For more information see Section 11.3.3 and Section 11.10.7.

4.4.8 Scalar shift

Scalar shift operations perform a variety of 32 and 64-bit shifts followed by an optional add/sub or logical operation. Figure 4-4 shows the general operation.

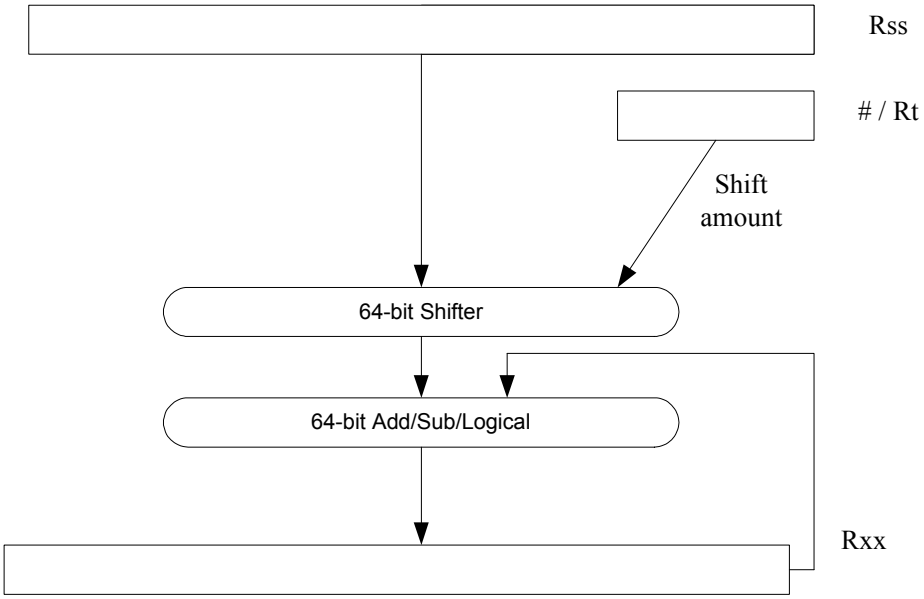


Figure 4-4 64-bit shift and add/sub/logical

Four shift types are supported:

- ASR – Arithmetic shift right
- ASL – Arithmetic shift left
- LSR – Logical shift right
- LSL – Logical shift left

In register-based shifts, the Rt register is a signed two's-complement number. If this value is positive, then the instruction opcode tells the direction of shift (right or left). If this value is negative, then the shift direction indicated by the opcode is reversed.

When arithmetic right shifts are performed, the sign bit is shifted in, whereas logical right shifts shift in zeros. Left shifts always shift in zeros.

Some shifts are available with saturation and rounding options.

For more information see Section 11.10.8.

4.4.9 Vector byte operations

Vector operations are available for packed vectors of unsigned bytes. These instructions include the following types:

- Vector add and subtract unsigned bytes
- Vector average unsigned bytes
- Vector compare unsigned bytes
- Vector min and max unsigned bytes
- Vector reduce add unsigned bytes
- Vector sum of absolute differences unsigned bytes

For more information see Section 11.10.9.

4.4.10 Vector halfword operations

Vector operations are available for processing packed 16-bit halfwords. These instructions include the following types:

- Vector add and subtract halfwords
- Vector average halfwords
- Vector compare halfwords
- Vector min and max halfwords
- Vector shift halfwords
- Vector dual multiply
- Vector dual multiply with round and pack
- Vector multiply even halfwords with optional round and pack
- Vector multiply halfwords
- Vector reduce multiply halfwords

For example, Figure 4-5 shows the operation of the vector arithmetic shift right halfword (VASRH) instruction. In this instruction, each 16-bit half-word is shifted right by the same amount which is specified in a register or with an immediate value. The bits shifted in are either a copy of the sign bit in the case of arithmetic shift or zero in the case of logical shift.

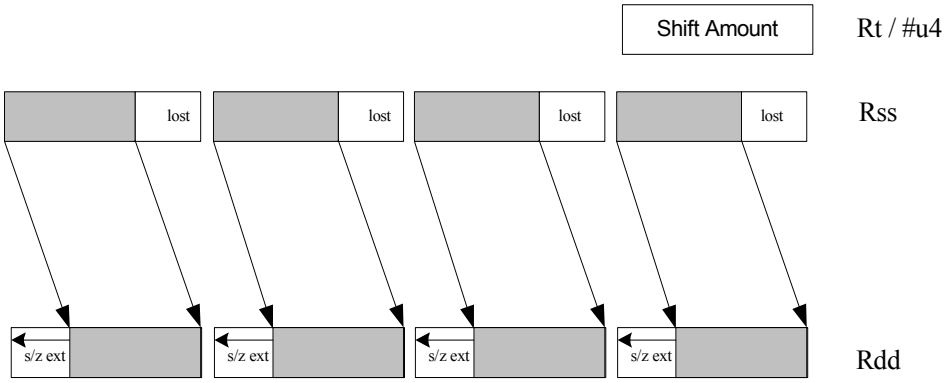


Figure 4-5 Vector halfword shift right

For more information see Section 11.3.4 and Section 11.10.10.

4.4.11 Vector word operations

Vector word operations are available for packed vectors of two words. These instructions include the following types:

- Vector add and subtract words
- Vector average words
- Vector compare words
- Vector min and max words
- Vector shift words with optional truncate and pack

For more information see Section 11.10.11.

4.5 ALU32 operations

The ALU32 instruction class includes general arithmetic/logical operations on 32-bit data:

- Add, subtract, negate without saturation on 32-bit data
- Logical operations such as And, Or, Xor, And with immediate, and Or with immediate
- Scalar 32-bit compares
- Combine halfwords, combine words, combine with immediates, shift halfwords, and Mux
- Conditional add, combine, logical, subtract, and transfer.
- NOP
- Sign and zero-extend bytes and halfwords
- Transfer immediates and registers
- Halfword vector add, subtract, and average

For more information see Section 11.3.

NOTE ALU32 instructions can be executed on any slot (Section 3.4.3).

Chapter 6 describes the conditional execution and compare instructions.

4.6 CR operations

The CR instruction class includes operations that access the control registers (Section 2.3).

Table 4-3 lists the instructions that access the control registers.

Table 4-3 Control register transfer instructions

Syntax	Operation
`Rd = Cs` `Cd = Rs`	Move control register to / from a general register. NOTE - PC is not a valid destination register.

For more information see Section 11.4.

5 Memory

5.1 Overview

The Hexagon processor features a load/store architecture where all numeric instructions operate on registers. Explicit load instructions move operands from memory to registers, while store instructions move operands from registers to memory.

The address space is unified and all accesses target the same linear address space which contains both instructions and data.

5.1 Memory model

This section describes the memory model for the Hexagon processor.

5.1.1 Address space

The Hexagon processor has a 32-bit byte-addressable memory address space. The entire 4G linear address space is addressable by the user application. A virtual-to-physical address translation mechanism is provided.

5.1.2 Byte order

The Hexagon processor is a little-endian machine: the lowest address byte in memory is held in the least significant byte of a register, as shown in Figure 5-1.

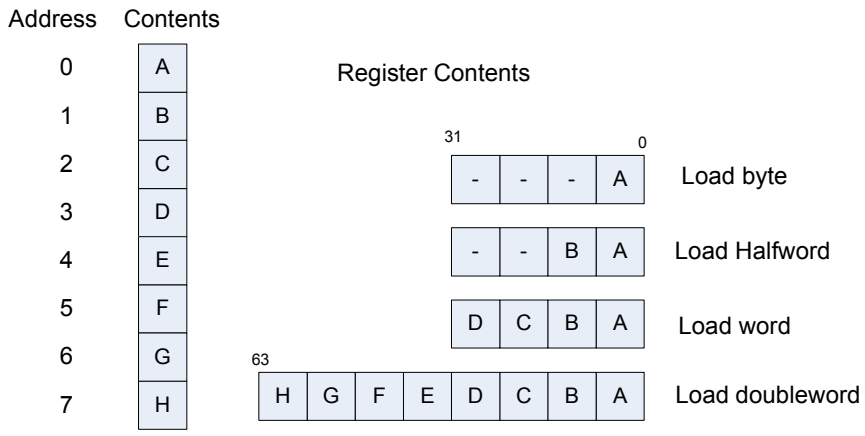


Figure 5-1 Hexagon processor byte order

5.1.3 Alignment

Even though the Hexagon processor memory is byte-addressable, instructions and data must be aligned in memory on specific address boundaries:

- Instructions and instruction packets must be 32-bit aligned
- Data must be aligned to its native access size.

Any unaligned memory access will cause a memory-alignment exception.

The permute instructions (Section 4.4.6) can be used in applications that need to reference unaligned vector data. The loads and stores still must be memory-aligned; however, the permute instructions enable the data to be easily rearranged in registers.

Table 5-1 summarizes the alignment restrictions.

Table 5-1 Memory alignment restrictions

Data Type	Size (bits)	Exception When
Byte Unsigned byte	8	Never
Halfword Unsigned halfword	16	LSB[0] != 0 [a]
Word Unsigned word	32	LSB[1:0] != 00
Doubleword	64	LSB[2:0] != 000
Instruction Instruction packet	32	LSB[1:0] != 00

[a] LSB = Least significant bits of address

5.2 Memory loads

Memory can be loaded in byte, halfword, word, or doubleword sizes. The data types supported are signed or unsigned. The syntax used is `memXX`, where `XX` denotes the data type.

Table 5-2 summarizes the supported load instructions.

Table 5-2 Load instructions

Syntax	Memory Size (bits)	Register Size (bits)	Data Placement	Comment
`Rd = memub(Rs)`	8	32	Low 8 bits	Zero-extend 8 to 32 bits
`Rd = memb(Rs)`	8	32	Low 8 bits	Sign-extend 8 to 32 bits
`Rd = memuh(Rs)`	16	32	Low 16 bits	Zero-extend 16 to 32 bits
`Rd = memh(Rs)`	16	32	Low 16 bits	Sign-extend 16 to 32 bits
`Rd = memubh(Rs)`	16	32	Bytes 0 and 2	Bytes 1 and 3 zeroed [a]
`Rd = memw(Rs)`	32	32	All 32 bits	Load 32-bit register
`Rdd = memubh(Rs)`	32	64	Bytes 0,2,4,6	Bytes 1,3,5,7 zeroed
`Rdd = memd(Rs)`	64	64	All 64 bits	Load 64-bit pair
`deallocframe`	64	64	All 64 bits	See Chapter 8

[a] The `memubh` instructions load contiguous bytes from memory (either 2 or 4 bytes) and unpack these bytes into a vector of halfwords. The instructions are useful when bytes are used as input into halfword vector operations, which is common in video and image processing.

5.3 Memory stores

Memory can be stored in byte, halfword, word, or doubleword sizes. Table 5-3 summarizes the supported store instructions.

Table 5-3 Store instructions

Syntax	Memory Size (bits)	Register Size (bits)	Operation
`memb(Rs) = Rt`	8	32	Store byte (bits 7:0)
`memh(Rs) = Rt`	16	32	Store lower half (bits15:0)
`memh(Rs) = Rt.H`	16	32	Store upper half (bits 31:16)
`memw(Rs) = Rt`	32	32	Store word
`memd(Rs) = Rtt`	64	64	Store doubleword
`allocframe(#u11)`	64	64	See Chapter 8

5.4 Addressing modes

Table 5-4 summarizes the supported addressing modes.

Table 5-4 Addressing modes

Mode	Syntax	Operation [a]
Global-pointer-relative	`memXX(GP+#immediate)` `memXX(#immediate)`	`EA = GP | immediate`
Indirect	`memXX(Rs)`	`EA = Rs`
Indirect with offset	`memXX(Rs+#s11)`	`EA = Rs + imm`
Indirect with auto-increment immediate	`memXX(Rx++#s4)`	`EA = Rx;` `Rx += (imm)`
Indirect with auto-increment register	`memXX(Rx++Mu)`	`EA = Rx;` `Rx += Mu`
Circular with auto-increment immediate	`memXX(Rx++#s4:circ(Mu))`	`EA = Rx;` `Rx=circ_add(Rx,imm,Mu)`
Circular with auto-increment register	`memXX(Rx++I:circ(Mu))`	`EA = Rx;` `Rx = circ_add(Rx,I,Mu)`
Bit-reversed with auto-increment register	`memXX(Rx++Mu:brev)`	`EA = Rx.H+bit_reverse(Rx.L);` `Rx += Mu`

[a] EA (Effective Address) is equivalent to VA (Virtual Address).

5.4.1 Global pointer relative

The global pointer relative addressing mode adds an unsigned offset value to the Hexagon processor global data pointer to create a 32-bit effective memory address. It is used to access global and static data in C.

Global pointer relative addresses can be expressed two ways in assembly language:

- By explicitly adding an unsigned offset value to register GP
- By specifying only the unsigned offset value as the instruction operand

For example:

```
R2 = memh(GP+#100)      // load R2 with signed halfword
                        // from [GP + 100 bytes]

R3 = memh(#200)         // load R3 with signed halfword
                        // from [GP + 200 bytes]
```

The global data pointer is programmed in the GDP field of the global pointer register GP (Section 2.3.7). This register field contains an unsigned 13-bit value which specifies the most significant 13 bits of the 32-bit global data pointer. (The least significant 19 bits of the pointer are defined to always be zero.)

The memory area referenced by the global data pointer is known as the *global data area*. It can be up to 512 KB in length, and – because of the way the global data pointer is defined – must be aligned to a 512 KB boundary in virtual memory (but only to a 4 KB boundary in physical memory).

When expressed in assembly language, the offset values used in global pointer relative addressing always specify byte offsets from the global data pointer. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-5 lists the offset ranges for global pointer relative addressing.

Table 5-5 Offset ranges (Global pointer relative)

Data Type	Offset Range	Offset Must Be Multiple Of
doubleword	0 ... 524280	8
word	0 ... 262140	4
halfword	0 ... 131070	2
byte	0 ... 65535	1

5.4.2 Indirect

The indirect addressing mode uses a 32-bit value stored in a general register to specify the memory address. For example:

```
R2 = memub(R1)   // load R2 with unsigned byte from addr R1
```

5.4.3 Indirect with offset

The indirect with offset addressing mode adds a signed offset value to a general register value to create a 32-bit effective memory address. For example:

```
R2 = memh(R3 + #100)   // load R2 with signed halfword
                       // from [R3 + 100 bytes]
```

When expressed in assembly language, the offset values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-6 lists the offset ranges for indirect with offset addressing.

Table 5-6 Offset ranges (Indirect with offset)

Data Type	Offset Range	Offset Must Be Multiple Of
doubleword	-8192 ... 8184	8
word	-4096 ... 4092	4
halfword	-2048 ... 2046	2
byte	-1024 ... 1023	1

NOTE The offset range is smaller for conditional instructions (Section 5.5).

5.4.4 Indirect with auto-increment immediate

The indirect with auto-increment immediate addressing mode uses a 32-bit value stored in a general register to specify the memory address. However, after the address is accessed, a signed value (known as the *increment*) is added to the register so it specifies a different memory address (which will be accessed in a subsequent instruction). For example:

```
R2 = memw(R3++#4)    // R3 contains the effective address
                     // R3 is then incremented by 4
```

When expressed in assembly language, the increment values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-7 lists the increment ranges for indirect with auto-increment immediate addressing.

Table 5-7 Increment ranges (Indirect with auto-inc immediate)

Data Type	Increment Range	Increment Must Be Multiple Of
doubleword	-64 ... 56	8
word	-32 ... 28	4
halfword	-16 ... 14	2
byte	-8 ... 7	1

5.4.5 Indirect with auto-increment register

The indirect with auto-increment register addressing mode is functionally equivalent to indirect with auto-increment immediate, but uses a modifier register `Mx` (Section 2.3.4) instead of an immediate value to hold the increment. For example:

```
R2 = memw(R0++M1)    // The effective addr is the value of R0.
                     // Next, M1 is added to R0 and the result
                     // is stored in R0.
```

When auto-incrementing with a modifier register, the increment is a signed 32-bit value which is added to the general register. This offers two advantages over auto-increment immediate:

- A larger increment range
- Variable increments (since the modifier register can be programmed at runtime)

When expressed in assembly language, the increment values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

NOTE The signed 32-bit increment range is identical for all instruction data types (doubleword, word, halfword, byte).

5.4.6 Circular with auto-increment immediate

The circular with auto-increment immediate addressing mode is a variant of indirect with auto-increment addressing – it accesses data buffers in a modulo wrap-around fashion. Circular addressing is commonly used in data stream processing.

Circular addressing is expressed in assembly language with the address modifier "`:circ(Mx)`", where `Mx` specifies a modifier register which is programmed to specify the circular buffer length and K value (Section 2.3.4). For example:

```
R0 = memb(R2++#8:circ(M0))    // load from circ buf pointed to
                              // by R2 and length/K value in M0
memw(R2++#4:circ(M1)) = R0    // store to circ buf pointed to
                              // by R2 and length/K value in M1
```

Circular addressing works with buffers of length 3 bytes to (128k-1) bytes.

The K value of a circular buffer specifies a power-of-2 size that is greater than the buffer length. It is based on the following formula:

$$\text{Length} < 2^{(K+2)}$$

Any K value that is larger than the buffer length is valid.

Table 5-8 summarizes the encoded K values.

Table 5-8 Valid circular buffer lengths

Encoded K value	Length must be less than (bytes)
0	4
1	8
2	16
3	32
4	64
5	128
6	256
7	512
8	1K
9	2K
10	4K
11	8K
12	16K
13	32K
14	64K
15	128K

After memory is accessed at the address specified in the general register, the general register is incremented by the immediate increment value and then moduloed by the buffer length to implement wrap-around access of the buffer.

When expressed in assembly language, the increment values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-9 lists the increment ranges for circular with auto-increment immediate addressing.

Table 5-9 Increment ranges (Circular with auto-inc immediate)

Data Type	Increment Range	Increment Must Be Multiple Of
doubleword	-64 ... 56	8
word	-32 ... 28	4
halfword	-16 ... 14	2
byte	-8 ... 7	1

When programming a circular buffer the following rules must be followed:

- The buffer must be aligned to its K value size. For example, a circular buffer of length 200 (K=6) must begin at an address where the least-significant 8 bits are zero.
- ABS(Increment) < Length. The absolute value of the increment must be less than the buffer length.
- Access size < (Length-1). The memory access size (1 for byte, 2 for halfword, 4 for word, 8 for doubleword) must be less than (Length-1).
- Buffers must not wrap around in the 32-bit address space.
- The K value must be larger than the programmed length.

NOTE If any of these rules are not followed the execution result is undefined.

For example, to set up a 150-byte circular buffer the length should be 150. The closest power-of-2 larger than 150 is 256. The K value should be set to 6 as shown in the table above. The data buffer must be aligned on a 256 byte boundary. The modifier register `Mx` can then be programmed and accessed as follows:

```
R4.H = #0x0600
R4.L = #150
M0 = R4
R0 = memb(R2++#8:circ(M0))    // Load byte from circ buf
                              // specified by M0
                              // inc R2 by 8 after load
                              // wrap R2 around if >= 150
```

To set up a 256-byte circular buffer the length should be 256. The closest power-of-2 larger than 256 is 512. K should be set to `#7`, and the length to `#256`.

The following C function precisely describes the behavior of the circular add function:

```
unsigned int
fcircadd(unsigned int pointer, int offset, unsigned int M_reg)
{
    unsigned int K_const, mask, length;
    int new_pointer, start_addr, end_addr;

    K_const = ((M_reg)>>24) & 0xf;      // M[27:24] = K in 2^(K+2)
    length = (M_reg&0x01ffff);          // M[16:0] = buf
    mask = (1<<(K_const+2))-1;          // size mask for low bits
    new_pointer = pointer+offset;
    start_addr = (pointer & (~mask));
    end_addr = (start_addr | length);

    if (new_pointer >= end_addr) {
        new_pointer -= length;
    } else if (new_pointer < start_addr) {
        new_pointer += length;
    }

    return (new_pointer);
}
```

5.4.7 Circular with auto-increment register

The circular with auto-increment register addressing mode is functionally equivalent to circular with auto-increment immediate, but uses a register instead of an immediate value to hold the increment.

Register increments are specified in circular addressing instructions by using the symbol `I` as the increment (instead of an immediate value). For example:

```
R0 = memb(R2++I:circ(M0))     // load from circ buf pointed to by
                              // R2 with size/K/inc values in M0.
                              // Post-increment R2 by I*4
```

When auto-incrementing with a register, the increment is a signed 11-bit value which is added to the general register. This offers two advantages over circular addressing with immediate increments:

- Larger increment ranges
- Variable increments (since the increment register can be programmed at runtime)

The circular register increment value is programmed in the `I` field of the modifier register `Mx` (Section 2.3.4) as part of setting up the circular data access. This register field holds the signed 11-bit increment value.

When expressed in assembly language, the increment values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-10 lists the increment ranges for circular with auto-increment register addressing.

Table 5-10 Increment ranges (Circular with auto-inc register)

Data Type	Increment Range	Increment Must Be Multiple Of
doubleword	-8192 ... 8184	8
word	-4096 ... 4092	4
halfword	-2048 ... 2046	2
byte	-1024 ... 1023	1

When programming a circular buffer (with either a register or immediate increment), all the rules that apply to circular addressing must be followed – for details see Section 5.4.6.

NOTE If any of these rules are not followed the execution result is undefined.

5.4.8 Bit-reversed with auto-increment register

The bit-reversed with auto-increment register addressing mode is a variant of indirect with auto-increment addressing – it accesses data buffers using an address value which is the bit-wise reversal of the value stored in the general register. Bit-reversed addressing is used in fast Fourier transforms (FFT) and Viterbi encoding.

The bit-wise reversal of a 32-bit address value is defined as follows:

- The lower 16 bits are transformed by exchanging bit 0 with bit 15, bit 1 with bit 14, and so on.
- The upper 16 bits remain unchanged.

Bit-reversed addressing is expressed in assembly language with the address modifier "`:brev`". For example:

```
R2 = memub(R0++M1:brev)  // The address is (R0.H | bitrev(R0.L))
                         // The orginal R0 (not reversed) is added
                         // to M1 and written back to R0
```

The maximum length for a bit-reversed buffer is 64K bytes.

To support bit-reversed addressing, buffers must be properly aligned in memory. A bit-reversed buffer is properly aligned when its starting byte address is aligned to a power of 2 greater than or equal to the buffer size (in bytes). For example, the bit-reversed buffer declared above is aligned to 1024 bytes because the buffer size is 1024 bytes (256 integer words × 4 bytes), and 1024 is an integral power of 2.

The buffer location pointer for a bit-reversed buffer must be initialized so the least-significant 16 bits of the address value are bit-reversed.

NOTE To simplify the initialization of the bit-reversed pointer, bit-reversed buffers can be aligned to a 64K byte boundary. This has the advantage of allowing the bit-reversed pointer to be initialized to the base address of the bit-reversed buffer, with no bit-reversing required for the least-significant 16 bits of the pointer value (which are all set to 0 by the 64K alignment).

Since buffers allocated in the stack should have an alignment of 8 bytes or less, in most cases bit-reversed buffers should not be declared on the stack.

After memory is accessed at the bit-reversed value stored in the general register, the general register is incremented by the register increment value. Note that the value in the general register is never affected by the bit-reversal process (which is used only to perform the memory access).

When expressed in assembly language, the increment values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

NOTE The Hexagon processor supports only register increments for bit-reversed addressing – it does not support immediate increments.

5.5 Conditional load/stores

Some load and store instructions can be executed conditionally based on predicate values which were set in a previous instruction. The compiler generates conditional loads and stores to increase instruction-level parallelism.

Conditional loads and stores are expressed in assembly language with the instruction prefix "`if (`*`pred_expr`*`)`", where *`pred_expr`* specifies a predicate register expression (Section 6.2). For example:

```
if (P0) R0 = memw(R2)              // conditional load
if (!P2) memh(R3 + #100) = R1      // conditional store
if (P1.new) R3 = memw(R3++#4)      // conditional load
```

Not all addressing modes are supported in conditional loads and stores. Table 5-11 shows which modes are supported.

Table 5-11 Addressing modes (Conditional load/store)

Addressing Mode	Conditional	Dot-new Conditional
Indirect	Yes	Yes
Indirect with offset	Yes	Yes
Indirect with auto-increment immediate	Yes	No
Indirect with auto-increment register	No	No

Table 5-11 Addressing modes (Conditional load/store)

Addressing Mode	Conditional	Dot-new Conditional
Circular with auto-increment immediate	No	No
Circular with auto-increment register	No	No
Bit-reversed with auto-increment register	No	No
Global pointer relative	No	No

When a conditional load or store instruction uses indirect-with-offset addressing mode, note that the offset range is smaller than the range normally defined for indirect-with-offset addressing (Section 5.4.3).

Table 5-12 lists the conditional and normal offset ranges for indirect-with-offset addressing.

Table 5-12 Conditional offset ranges (Indirect with offset)

Data Type	Offset Range (Conditional)	Offset Range (Normal)	Offset Must Be Multiple Of
doubleword	0 ... 504	-8192 ... 8184	8
word	0 ... 252	-4096 ... 4092	4
halfword	0 ... 126	-2048 ... 2046	2
byte	0 ... 63	-1024 ... 1023	1

NOTE For more information on conditional instructions see Chapter 6.

5.6 Cache memory

The Hexagon processor has a cache-based memory architecture:

- A level 1 *instruction cache* holds recently-fetched instructions.
- A level 1 *data cache* holds recently-accessed data memory[1].

Load/store operations that access memory through the level 1 caches are referred to as *cached accesses.*

Load/stores that bypass the level 1 caches are referred to as *uncached accesses*.

Specific memory areas can be configured so they perform cached or uncached accesses. This configuration is performed by the Hexagon processor's memory management unit (MMU). The operating system is responsible for programming the MMU.

[1] In the Hexagon V2 processor the L1 instruction and data caches are both 32 KB.

Two types of caching are supported (as cache modes):

- W*rite-through caching* keep the cache data consistent with external memory by always writing to the memory any data that is stored in the cache.
- W*rite-back caching* allows data to be stored in the cache without being immediately written to external memory. Cached data that is inconsistent with external memory is referred to as *dirty.*

The Hexagon processor includes dedicated cache maintenance instructions which can be used to push dirty data out to external memory.

5.6.1 Uncached memory

In some cases load/store operations need to bypass the cache memories and be serviced externally (for example, when accessing memory-mapped I/O, registers, and peripheral devices, or other system defined entities). The operating system is responsible for configuring the MMU to generate uncached memory accesses.

If a load or store accesses uncached memory, it must be the only memory access performed in an instruction packet (Section 3.4). If a packet performs multiple memory accesses and one or more of them is to uncached memory, an error exception is raised.

5.6.2 Tightly coupled memory

Tightly-coupled memory at Level 1 (which is defined as memory having similar access properties to the data cache) is not supported in the Hexagon V2 processor.

Tightly-coupled memory at level 2 (which is defined as backing store to the primary caches) is supported.[1]

For more information see Section 9.2.

5.6.3 Cache maintenance operations

The Hexagon processor includes dedicated cache maintenance instructions which can be used to invalidate cache data or push dirty data out to external memory.

The cache maintenance instructions operate on specific memory addresses. If the instruction causes an address error, the processor raises an exception.

NOTE The one exception is `dcfetch`, which never causes an exception.

[1] Implementations are available with 256K and 512K L2 TCM

Whenever maintenance operations are performed on the instruction cache, the `isync` instruction (Section 5.7) must be executed immediately afterwards. This instruction ensures that the maintenance operations will be observed by subsequent instructions.

Table 5-13 lists the cache maintenance instructions.

Table 5-13 Cache instructions (User-level)

Syntax	Permitted In Packet	Operation
`icinva(Rs)`	Solo [a]	Instruction cache invalidate. Look up instruction cache at address Rs. If address is in cache, invalidate it.
`dccleaninva(Rs)`	Slot 1 empty	Data cache clean and invalidate. Look up data cache at address Rs. If address is in cache and has dirty data, flush that data out to memory. The cache line is then invalidated, whether or not dirty data was written.
`dccleana(Rs)`	Slot 1 empty	Data cache clean. Look up data cache at address Rs. If address is in cache and has dirty data, flush that data out to memory.
`dcinva(Rs)`	Slot 1 empty	Data cache invalidate. Look up data cache at address Rs. If address is in cache, invalidate it.
`dcfetch(Rs)`	Normal [b]	Data cache prefetch. Prefetch data at address Rs into data cache. NOTE - This instruction will not cause an exception.

[a] *Solo* means that the instruction must not be grouped with other instructions in a packet.

[b] *Normal* means that the normal instruction-grouping constraints apply.

5.6.4 L2 cache operations

The cache maintenance operations (Section 5.6.3) operate on both the L1 and L2 caches.

The data cache coherency operations (including clean, invalidate, and clean and invalidate) affect both the L1 and L2 caches, and ensure that the memory hierarchy remains coherent.

However, the instruction cache invalidate operation affects only the L1 cache. Therefore, invalidating instructions that may be in the L1 or L2 caches requires a two-step procedure:

1. Use `icinva` to invalidate instructions from the L1 cache.
2. Use `dcinva` separately to invalidate instructions from the L2 cache.

5.6.5 Cache line zero

The Hexagon processor includes the instruction `dczeroa`. This instruction allocates a line in the L1 data cache and clears it (by storing all zeros). The behavior is as follows:

- The Rs register value must be 32-byte aligned. If it is unaligned, the processor will raise an unaligned error exception.
- In the case of a cache hit, the specified cache line is cleared (i.e., written with all zeros) and made dirty.
- In the case of a cache miss, the specified cache line is *not* fetched from external memory. Instead, the line is allocated in the data cache, cleared, and made dirty.

This instruction is useful in optimizing write-only data. It allows for the use of write-back pages – which are the most power and performance efficient – without the need to initially fetch the line to be written. This removes unnecessary read bandwidth and latency.

The `dczeroa` instruction has the same exception behavior as write-back stores.

A packet containing `dczeroa` must have Slot 1 either empty or containing a NOP.

NOTE `dczeroa` works only with write-back cache pages. If it is executed using an uncached or write-through page, the processor will raise an error exception.

5.7 Memory ordering

Some devices may require synchronization of stores and loads when they are accessed. In this case a set of processor instructions enable programmer control of the synchronization and ordering of memory accesses.

Table 5-14 lists the memory-ordering instructions.

Table 5-14 Memory ordering instructions

Syntax	Operation
`isync`	Instruction synchronize. This instruction should be executed after any instruction cache maintenance operation.
`syncht`	Synchronize transactions. Perform "heavyweight" synchronization. Ensure that all previous program transactions (e.g., memw_locked, cached and uncached load/store) have completed before execution resumes past this instruction.
`barrier`	Set memory barrier. Ensure proper ordering between the program accesses performed before the instruction and those performed after the instruction. NOTE - All accesses before the barrier will be globally observable before any access occurring after the barrier can be observed. NOTE - Not all implementations support the use of this instruction. For those that do not, the alternative is to perform a *read-flush* operation: the user reads back the last byte written to ensure it has reached the destination.

Data memory accesses and program memory accesses are treated separately and held in separate caches. Software should ensure coherency between data and program code if necessary.

For example, with generated or self-modified code, the modified code will be placed in the data cache and may be inconsistent with program cache. The software must explicitly force modified data cache lines to memory (either by using a write-through policy, or through explicit cache clean instructions). A `barrier` instruction should then be used to ensure the completion of the stores. Finally, relevant instruction cache contents should be invalidated so the new instructions can be re-fetched.

Here is the recommended code sequence to change and then execute an instruction:

```
memw(R1) = R0  // R1 points to code space, R0 is new instruction
dccleana(R1)   // force data out of data cache
barrier        // ensure data is in memory
icinva(R1)     // clear it from instruction cache
isync          // ensure icinva instr is finished
jumpr R1       // can now execute code at R1
```

NOTE The memory-ordering instructions must not be grouped with other instructions in a packet, otherwise the behavior is undefined.

5.8 Atomic operations

The Hexagon processor includes an LL/SC (Load Locked / Store Conditional) mechanism to provide the atomic read-modify-write operation that is necessary to implement synchronization primitives such as semaphores and mutexes.

These primitives are used to synchronize the execution of different software programs running concurrently on the Hexagon processor. They can also be used to provide atomic memory support between the Hexagon processor and external blocks.

Table 5-15 describes the atomic instructions.

Table 5-15 Atomic instructions

Syntax	Description
`Rd = memw_locked(Rs)`	Load locked. Reserve lock at address Rs.
`memw_locked(Rs,Pd) = Rt`	Store conditional. Perform conditional store to address Rs. If no other atomic operation has been performed at this address (i.e., atomicity is ensured), perform the store to the specified address and return TRUE; otherwise return FALSE. TRUE indicates the LL and SC operations have been performed atomically.

Here is the recommended code sequence to acquire a mutex:

```
// assume mutex address is held in R0
// assume R1,R3,P0,P1 are scratch

lockMutex:
   R3 = #1
lock_test_spin:
   R1 = memw_locked(R0)          // do a normal test
   P1 = cmp.eq(R1,#0)           // to wait for lock to be available
   if (!P1) jump lock_test_spin
   memw_locked(R0,P0) = r3       // do store conditional (SC)
   if (!P0) jump lock_test_spin  // was LL and SC done atomically?
```

Here is the recommended code sequence to release a mutex:

```
// assume mutex address is held in R0
// assume R1 is scratch

R1 = #0
memw(R0) = R1
```

NOTE `memw_locked` cannot be grouped with other instructions in an instruction packet.

Atomic `memw_locked` operations are supported for external accesses that use the AXI bus and support atomic operations. To perform load-locked operations with external memory, the operating system must define the memory page as uncacheable, otherwise the processor behavior is undefined.

If a load locked operation is performed on an address that does not support atomic operations, the behavior is undefined.

For atomic operations on cacheable memory, the page attributes must be set to cacheable and write-back, otherwise the behavior is undefined. Cacheable memory must be used when threads need to synchronize with each other.

NOTE External `memw_locked` operations are not supported on the AHB bus. If they are performed on the AHB bus, the behavior is undefined.

6 Conditional Execution

6.1 Overview

The Hexagon processor uses a conditional execution model based on compare instructions that set predicate bits in one of four 8-bit predicate registers (`P0-P3`). These predicate bits can be used to conditionally execute certain instructions.

Conditional scalar operations examine only the least-significant bit in a predicate register, while conditional vector operations examine multiple bits in the register.

Branch instructions and the `mux` instruction are the main consumers of the predicate registers.

6.2 Scalar predicates

Scalar predicates are 8-bit values which are used in conditional instructions to represent truth values:

- 0xFF represents true
- 0x00 represents false

The Hexagon processor provides the four 8-bit predicate registers P0-P3 to hold scalar predicates (Section 2.3.5). These registers are assigned values by the predicate-generating instructions, and examined by the predicate-consuming instructions.

6.2.1 Generating scalar predicates

:The following instructions generate scalar predicates:

- Compare word, doubleword
- Compare bitmask
- Store conditional

Table 6-1 lists the predicate-generating instructions.

Table 6-1 Scalar predicate-generating instructions

Syntax	Operation
`Pd = cmp.eq(Rs,{Rt,#s8})` `Pd = cmp.eq(Rss,Rtt)`	Equal (signed). Compare register Rs to Rt or a signed immediate for equality. Assign Pd the resulting truth value.
`Pd = cmp.gt(Rs,{Rt,#s8})` `Pd = cmp.gt(Rss,Rtt)`	Greater than (signed). Compare register Rs to Rt or a signed immediate for signed greater than. Assign Pd the resulting truth value.
`Pd = cmp.gtu(Rs,{Rt,#u8})` `Pd = cmp.gtu(Rss,Rtt)`	Greater than (unsigned). Compare register Rs to Rt or an unsigned immediate for unsigned greater than. Assign Pd the resulting truth value.
`Pd = cmp.ge(Rs,#s8)`	Greater than or equal (signed). Compare register Rs to a signed immediate for signed greater than or equal. Assign Pd the resulting truth value.
`Pd = cmp.geu(Rs,#u8)`	Greater than or equal (unsigned). Compare register Rs to an unsigned immediate for unsigned greater than or equal. Assign Pd the resulting truth value.

Table 6-1 Scalar predicate-generating instructions (Continued)

`Pd = tstbit(Rs,{Rt,#u5})`	Test if bit set. Rt or an unsigned immediate specifies a bit position. Test if the bit in Rs that is specified by the bit position is set. Assign Pd the resulting truth value.
`Pd = bitsclr(Rs,{Rt,#u6})`	Test if bits clear. Rt or an unsigned immediate specifies a bitmask. Test if the bits in Rs that are specified by the bitmask are all clear. Assign Pd the resulting truth value.
`Pd = bitsset(Rs,Rt)`	Test if bits set. Rt specifies a bitmask. Test if the bits in Rs that are specified by the bitmask are all set. Assign Pd the resulting truth value.
`memw_locked(Rs,Pd) = Rt`	Store conditional. If no other atomic operation has been performed at the address (i.e., atomicity is ensured), perform the store to the word at address Rs. Assign Pd the resulting truth value.

The compare operations have three basic forms:

- EQ – Compare for equal
- GT – Compare for signed greater than
- GTU – Compare for unsigned greater than

These forms are sufficient to generate all comparisons of signed and unsigned values including EQ, NEQ, LT, LE, GT, and GE.

The truth value returned by each compare operation is a truth value which – by using the logical negation operator `!` – can be tested in either sense (i.e., true or false). For example, consider the scalar case of branch on a not-equal value. The code would appear as:

```
P0 = cmp.eq(R0,R1)          // compare on equal
if (!P0) jump target        // jump if not equal
```

Additionally, note that the register operands can be reversed to produce another comparison. For example, to jump if R0 is less than R1 (R0<R1), use the following:

```
P0 = cmp.gt(R1,R0)         // R1 > R0 equivalent to R0 < R1
if (P0) jump target        // jump if R0 < R1
```

By swapping operands and testing both senses of the result, it is possible to perform the full complement of signed and unsigned comparisons, including EQ, NEQ, LT, LE, GT, and GE.

Table 6-2 summarizes these combinations.

Table 6-2 Compare forms and logic

Instruction	Swap Operands	Output Sense	Operation
EQ	No	True	EQ
EQ	No	False	NEQ
GT	No	True	Signed GT
GT	No	False	Signed LE
GT	Yes	True	Signed LT
GT	Yes	False	Signed GE
GTU	No	True	unsigned GT
GTU	No	False	unsigned LE
GTU	Yes	True	unsigned LT
GTU	Yes	False	unsigned GE

6.2.2 Consuming scalar predicates

Certain instructions can be conditionally executed based on the value of a scalar predicate (or alternatively specify a scalar predicate as an input to their operation).

The conditional instructions that consume scalar predicates examine only the least-significant bit of the predicate value. In the simplest case, this bit value directly determines whether the instruction is executed:

- 1 indicates that the instruction is executed
- 0 indicates that the instruction is not executed

If a conditional instruction includes the operator `!` in its predicate expression, then the logical negation of the bit value determines whether the instruction is executed.

Conditional instructions are expressed in assembly language with the instruction prefix "`if (`*`pred_expr`*`)`", where *`pred_expr`* specifies the predicate expression. For example:

```
if (P0) jump target          // jump if P0 is true
if (!P2) R2 = R5             // assign register if !P2 is true
if (P1) R0 = sub(R2,R3)      // conditionally subtract if P1
if (P2) R0 = memw(R2)        // conditionally load word if P2
```

The following instructions can be used as conditional instructions:

- Jumps and calls (Section 7.4)
- 32-bit register transfer and 64-bit combine word
- Register transfer immediate
- Logical instructions (including AND/OR/XOR)
- 32-bit add/subtract by register or short immediate
- Many load and store instructions (Section 5.5)

When a conditional load or store is executed and the predicate expression is false, the instruction is cancelled (including any exceptions that might occur). For example, if a conditional load uses an address with a memory permission violation, and the predicate expression is false, then the load does not execute and the exception is not raised.

The `mux` instruction accepts a predicate register as one of its basic operands:

```
Rd = mux(Ps,Rs,Rt)
```

`mux` selects either Rs or Rt based on the least significant bit in Ps. If the least-significant bit in Ps is a 1, then Rd is set to Rs, otherwise it is set to Rt.

6.2.3 Dot-new predicates

The Hexagon processor can generate and use a scalar predicate in the same instruction packet (Section 3.4). This feature is expressed in assembly language by appending the suffix "`.new`" to the specified predicate register. For example:

```
if (P0.new) R3 = memw(R4)
```

To see how the dot-new predicate is used, consider the following C statement and the corresponding assembly code that is generated from it by the compiler:

C statement

```
if (R2 == 4)
   R3 = *R4;
else
   R5 = 5;
```

Assembly code

```
{
   P0 = cmp.eq(R2,#4)
   if (P0.new) R3 = memw(R4)
   if (!P0.new) R5 = #5
}
```

In the assembly code a scalar predicate is generated and then consumed twice within the same instruction packet.

The following conditions apply to using the dot-new predicate:

- The predicate must be generated by an instruction in the same packet. The assembler normally enforces this restriction, but if the processor executes a packet that violates this restriction, the execution result is undefined.

- A single packet can contain both the dot-new and normal forms of a predicate. The normal form examines the old value in the predicate register, rather than the newly-generated value. For example:

```
{
  P0 = cmp.eq(R2,#4)
  if (P0.new) R3 = memw(R4)    // use newly-generated P0 value
  if (P0) R5 = #5             // use previous P0 value
}
```

6.2.4 Dependency constraints

Two instructions in an instruction packet should not write to the same destination register (Section 3.4.5). The exception to this rule is if the two instructions are conditional, and only one of them ever has the predicate expression value `true` when the packet is executed.

For example, the following packet is valid as long as `P2` and `P3` never both evaluate to `true` when the packet is executed:

```
{
   if (P2) R3 = #4       // P2, P3, or both must be false
   if (P3) R3 = #7
}
```

Because predicate values change at runtime, the programmer is responsible for ensuring that such packets are always valid during program execution. If they are invalid, the processor takes the following actions:

- When writing to general registers, an error exception is raised.
- When writing to predicate or control registers, the result is undefined.

6.3 Vector predicates

The predicate registers are also used for conditional vector operations. Unlike scalar predicates, vector predicates contain multiple truth values which are generated by vector predicate-generating operations.

For example, a vector compare instruction compares each element of a vector and assigns the compare results to a predicate register. Each bit in the predicate vector contains a truth value indicating the outcome of a separate compare performed by the vector instruction.

The vector `mux` instruction uses a vector predicate to selectively merge elements from two separate vectors into a single destination vector. This operation is useful for enabling the vectorization of loops with control flow (i.e., branches).

The vector instructions that use predicates are described in the following sections.

6.3.1 Vector compare

A vector compare instruction inputs two 64-bit vectors, performs separate compares for each pair of vector elements, and generates a predicate value which contains a bit vector of truth values.

Figure 6-1 shows an example of a vector byte compare.

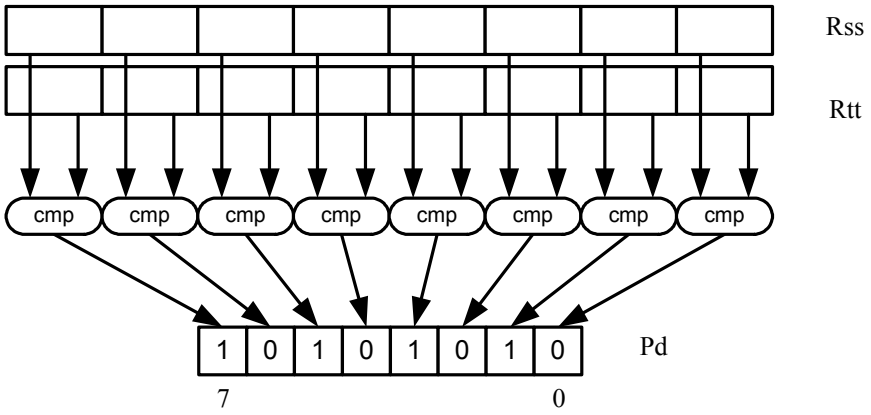


Figure 6-1 Vector byte compare

In Figure 6-1 two 64-bit vectors of bytes (contained in Rss and Rtt) are being compared. The result is assigned as a vector predicate to the destination register Pd.

In the example vector predicate shown in Figure 6-1, note that every other compare result in the predicate is true (i.e., 1).

Figure 6-2 shows how a vector halfword compare generates a vector predicate.

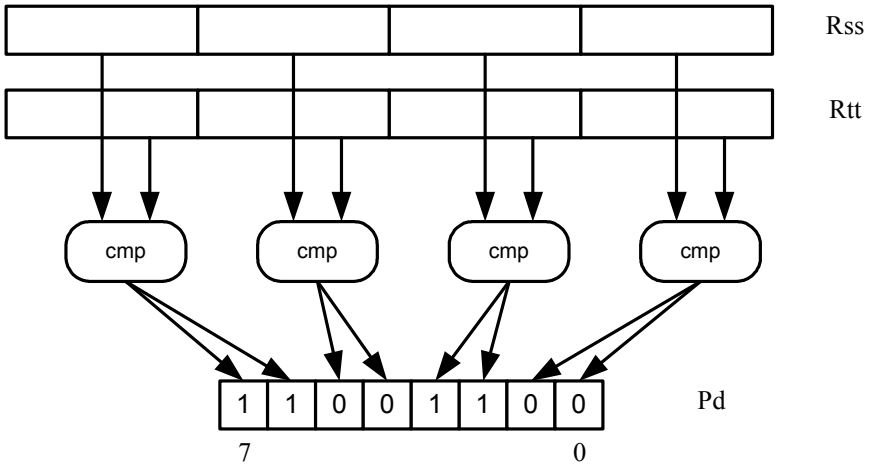


Figure 6-2 Vector halfword compare

In Figure 6-2 two 64-bit vectors of halfwords are being compared. The result is assigned as a vector predicate to the destination register Pd.

Because a vector halfword compare yields only four truth values, each truth value is encoded as two bits in the generated vector predicate.

6.3.2 Vector mux instruction

A vector mux instruction is used to conditionally select the elements from two vectors. The instruction takes as input two source vectors and a predicate register. For each byte in the vector, the corresponding bit in the predicate register is used to choose from one of the two input vectors. The combined result is written to the destination register.

Figure 6-3 shows the operation of the vector mux instruction.

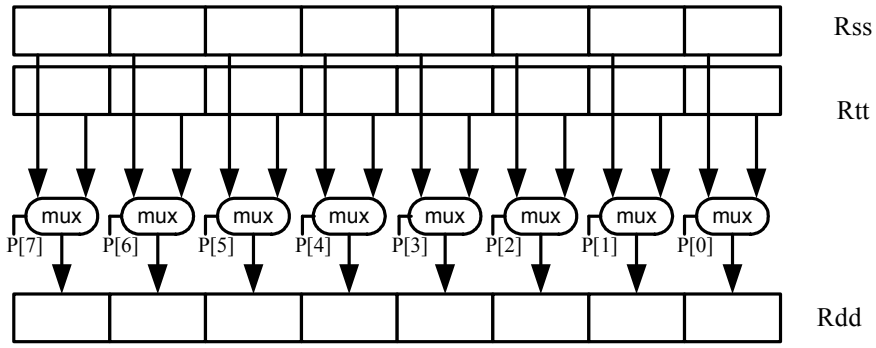


Figure 6-3 Vector mux instruction

Table 6-3 defines the vector mux instruction.

Table 6-3 Vector mux instruction

Syntax	Operation
`Rdd = vmux(Ps,Rss,Rtt)`	Select bytes from Rss and Rtt

Note that by changing the order of the source operands in a mux instruction, both senses of the result can be formed. For example:

```
R1:0 = vmux(P0,R3:2,R5:4)     // choose bytes from R3:2 if true
R1:0 = vmux(P0,R5:4,R3:2)     // choose bytes from R3:2 if false
```

NOTE By replicating the predicate bits generated by word or halfword compares, the vector mux instruction can be used to select words or halfwords.

6.3.3 Using vector conditionals

Vector conditional support is used to vectorize loops with conditional statements.

Consider the following C statement:

```
for (i=0; i<8; i++) {
   if (A[i]) {
      B[i] = C[i];
   }
}
```

Assuming arrays of bytes, this code can be vectorized as follows:

```
R1:0 = memd(R_A)              // R1:0 holds A[7]-A[0]
R3 = #0                      // clear R3:2
R2 = #0
P0 = vcmpb.eq(R1:0,R3:2)     // compare bytes in A to zero
R5:4 = memd(R_B)             // R5:4 holds B[7]-B[0]
R7:6 = memd(R_C)             // R7:6 holds C[7]-C[0]
R3:2 = vmux(P0,R7:6,R5:4)    // if (A[i]) B[i]=C[i]
memd(R_B) = R3:2             // store B[7]-B[0]
```

6.4 Predicate operations

The Hexagon processor provides a set of operations for manipulating and moving predicate registers.

Table 6-4 lists the predicate register instructions.

Table 6-4 Predicate register instructions

Syntax	Operation
`Pd = Ps`	Transfer predicate Ps to Pd
`Pd = Rs`	Transfer register Rs to predicate Pd
`Rd = Ps`	Transfer predicate Ps to register Rd
`Pd = and(Ps,[!]Pt)`	Set Pd to bitwise AND of Ps and [NOT] Pt
`Pd = or(Ps,[!]Pt)`	Set Pd to bitwise OR of Ps and [NOT] Pt
`Pd = not(Ps)`	Set Pd to bitwise inversion of Ps
`Pd = xor(Ps,Pt)`	Set Pd to bitwise exclusive OR of Ps and Pt
`Pd = any8(Ps)`	Set Pd to 0xFF if any bit in Ps is 1, 0x00 otherwise
`Pd = all8(Ps)`	Set Pd to 0x00 if any bit in Ps is 0, 0xFF otherwise

NOTE These instructions belong to instruction class CR.

Predicate registers can be transferred to and from the general registers either individually or as register quadruples (Section 2.3.5).

7 Program Flow

7.1 Overview

The Hexagon processor supports the following program flow facilities:

- Conditional instructions
- Hardware loops
- Software branches
- Speculative jumps
- Exceptions

7.2 Conditional instructions

Many Hexagon processor instructions can be conditionally executed. For example:

```
if (P0) R0 = memw(R2)     // conditionally load word if P0
if (!P1) jump label       // conditionally jump if not P1
```

The following instructions can be specified as conditional:

- Jumps and calls
- 32-bit register transfer and 64-bit combine word
- Register transfer immediate
- Logical instructions (including AND/OR/XOR)
- 32-bit add/subtract by register or short immediate
- Many load and store instructions

For more information see Section 5.5 and Chapter 6.

7.3 Hardware loops

The Hexagon processor includes *hardware loop* instructions which can perform loop branches with a zero-cycle overhead. For example:

```
  loop0(start,#3)           // loop 3 times
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

Two hardware loop instructions are provided – `loop0` and `loop1` – to enable hardware loops to be nested one level deep. For example:

```
// Sum the rows of a 100x200 matrix.

  loop1(outer_start,#100)
outer_start:
    R0 = #0
    loop0(inner_start,#200)
inner_start:
      R3 = memw(R1++#4)
      { R0 = add(R0,R3) }:endloop0
    { memw(R2++#4) = R0 }:endloop1
```

The hardware loop instructions are used as follows:

- For non-nested loops, `loop0` is used.
- For nested loops, `loop0` is used for the inner loop, and `loop1` for the outer loop.

NOTE If a program needs to create loops nested more than one level deep, the two innermost loops can be implemented as hardware loops, with the remaining outer loops implemented as software branches.

Each hardware loop is associated with a pair of dedicated loop registers:

- The *loop start address* register `SAn` is set to the address of the first instruction in the loop (which is typically expressed in assembly language as a label).
- The *loop count* register `LCn` is set to a 32-bit unsigned value which specifies the number of loop iterations to perform. When the PC reaches the end of the loop, `LCn` is examined to determine whether the loop should repeat or exit.

To set up a hardware loop these registers do not need to be explicitly programmed – the hardware loop instruction does this automatically. However, because the loop registers completely specify the hardware loop state, they can be saved and restored (either automatically by a processor interrupt or manually by the programmer), enabling a suspended hardware loop to be resumed normally once its loop registers are reloaded with their original values.

The Hexagon processor provides two sets of loop registers for the two hardware loops:

- `SA0` and `LC0` are used by `loop0`
- `SA1` and `LC1` are used by `loop1`

Table 7-1 lists the hardware loop instructions.

Table 7-1 Loop instructions

Syntax	Description
loopN(start, Rs)	Hardware loop with register loop count. Set registers SAn and LCn for hardware loop N: ■ SAn is assigned the specified start address of the loop. ■ LCn is assigned the value of general register Rs. NOTE - The loop start operand is represented by a PC-relative immediate value.
loopN(start, #count)	Hardware loop with immediate loop count. Set registers SAn and LCn for hardware loop N: ■ SAn is assigned the specified start address of the loop. ■ LCn is assigned the specified immediate value (0-1023). NOTE - The loop start operand is represented by a PC-relative immediate value.
:endloopN	Hardware loop end instruction. Performs the following operation: `if (LCn > 1) {PC = SAn; LCn = LCn-1}` NOTE - This instruction appears in assembly as a suffix appended to the last packet in the loop. It is encoded in the last packet.
SAn = Rs	Set loop start address to general register Rs
LCn = Rs	Set loop count to general register Rs

NOTE The loop instructions are assigned to instruction class CR.

7.3.1 Loop setup

To set up a hardware loop, the loop registers SAn and LCn must be set to the proper values. This can be done in two ways:

- A loopN instruction
- Register transfers to SAn and LCn

The loopN instruction performs all the work of setting SAn and LCn. For example:

```
  loop0(start,#3)           // SA0=&start, LC0=3
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

In this example the hardware loop (consisting of a single multiply instruction) is executed three times. The loop0 instruction sets register SA0 to the address value of label start, and LC0 to 3.

Loop counts are limited to the range 0-1023 when they are expressed as immediate values in loopN. If the desired loop count exceeds this range, it must be specified as a register value. For example:

Using loopN:

```
  R0 = #20000;
  loop0(start,R0)          // LC0=20000, SA0=&start
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

Using register transfers:

```
  R0 = #20000
  LC0 = R0                 // LC0=20000
  R0 = #start
  SA0 = R0                 // SA0=&start
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

If a loopN instruction is located too far from its loop start address, the PC-relative offset value that is used to specify the start address may exceed the maximum range of the instruction's start-address operand. If this occurs, either move the loopN instruction closer to the loop start, or use SAn and LCn to set up the loop.

7.3.2 Loop end

The loop end instruction indicates the last packet in a hardware loop. It is expressed in assembly language by appending the packet with the symbol "`:endloopN`", where `N` specifies the hardware loop (0 or 1). For example:

```
  loop0(start,#3)
start:
    { R0 = mpyi(R0,R0) } :endloop0  // last packet in loop
```

The loop end instruction must always be expressed in assembly language as a packet (using curly braces), even if it is the only instruction in the packet.

Nested hardware loops can specify the same instruction as the end of both the inner and outer loops. For example:

```
// Sum the rows of a 100x200 matrix.
// Software pipeline the outer loop.

     p0 = cmp.gt(R0,R0)              // p0 = false
     loop1(outer_start,#100)
outer_start:
     { if (p0) memw(R2++#4) = R0
       p0 = cmp.eq(R0,R0)            // p0 = true
       R0 = #0
       loop0(inner_start,#200) }
inner_start:
     R3 = memw(R1++#4)
     { R0 = add(R0,R3) }:endloop0:endloop1
     memw(R2++#4) = R0
```

Though `endloopN` behaves like a regular instruction (by implementing the loop test and branch), note that it does not execute in any instruction slot, and does not count as an instruction in the packet. Therefore a single instruction packet which is marked as a loop end can perform up to six operations:

- Four regular instructions (the normal limit for a packet)
- The `endloop0` test and branch
- The `endloop1` test and branch

NOTE The `endloopN` modifier is encoded in the instruction packet (Section 10.5).

7.3.3 Loop execution

After a hardware loop is set up, the loop body always executes at least once regardless of the specified loop count (because the loop count is not examined until the last instruction in the loop). Therefore, if a loop needs to be optionally executed zero times, it must be preceded with an explicit conditional branch. For example:

```
    loop0(start,R1)
    P0 = cmp.eq(R1,#0)
    if (P0) jump skip
start:
    { R1 = add(R1,#1) } :endloop0
skip:
```

In this example a hardware loop is set up with the loop count in R1, but if the value in R1 is zero a software branch skips over the loop body.

After the loop end instruction of a hardware loop is executed, the Hexagon processor examines the value in the corresponding loop count register:

- If the value is greater than 1, the processor decrements the loop count register and performs a zero-cycle branch to the loop start address.
- If the value is less than or equal to 1, the processor resumes program execution at the instruction immediately following the loop end instruction.

NOTE Because nested hardware loops can share the same loop end instruction, the processor may examine both loop count registers in a single operation.

7.3.4 Loop restrictions

Hardware loops have the following restrictions:

- The last instruction packet in a loop cannot contain any program flow instructions (including jumps or calls).
- A `loopN` instruction cannot appear in the same packet as a jump or call.
- The loop end packet in `loop0` cannot contain any instruction that changes `SA1` or `LC1`. Similarly, the loop end packet in `loop1` cannot contain any instruction that changes `SA0` or `LC0`.
- The loop end packet in `sNloop` (Section 7.3.5) cannot contain any instruction that changes `P3`.

NOTE `SA1` and `LC1` can be changed at the end of `loop0`, while `SA0` and `LC0` can be changed at the end of `loop1`.

7.3.5 Pipelined loops

Software pipelined loops are common for VLIW architectures such as the Hexagon processor. They offer increased code performance in loops by overlapping multiple loop iterations.

A software pipeline has three sections:

- A *prologue* in which the loop is primed
- A *kernel* (or steady-state) portion
- An *epilogue* which drains the pipeline

This is best illustrated with a simple example, as shown in Table 7-2.

Table 7-2 Software pipelined loop

```
int foo(int *A, int *result)
{
    int i;
    for (i=0;i<100;i++) {
        result[i]= A[i]*A[i];
    }
}
```

```
foo:
{       R3 = R1
        loop0(.kernel,#98)          // Decrease loop count by 2
}
        R1 = memw(R0++#4)           // 1st prologue stage
{       R1 = memw(R0++#4)           // 2nd prologue stage
        R2 = mpyi(R1,R1)
}
        .falign
.kernel:
{       R1 = memw(R0++#4)           // kernel
        R2 = mpyi(R1,R1)
        memw(R3++#4) = R2
}:endloop0
{       R2 = mpyi(R1,R1)            // 1st epilogue stage
        memw(R3++#4) = R2
}
        memw(R3++#4) = R2           // 2nd epilogue stage
        jumpr lr
```

In Table 7-2 the kernel section of the pipelined loop performs three iterations of the loop in parallel:

- The load for iteration N+2
- The multiply for iteration N+1
- The store for iteration N

One drawback to software pipelining is the extra code necessary for the prologue and epilogue sections of a pipelined loop.

To address this issue the Hexagon processor provides the `sNloop0` instruction, where the "`N`" in the instruction name indicates a digit in the range 1-3. For example:

```
P3 = s2loop0(start,#10)      // Set up pipelined loop
```

`sNloop` is a variant of the `loop0` instruction: it sets up a normal hardware loop using `SA0` and `LC0`, but also performs the following additional operations:

- When the `sNloop` instruction is executed, it assigns the truth value `false` to the predicate register `P3`.
- After the associated loop has executed N times, `P3` is automatically set to true.

This feature (which is known as *automatic predicate control*) enables the store instructions in the kernel section of a pipelined loop to be conditionally executed by `P3` and thus – because of the way `sNloop` controls `P3` – not be executed during the pipeline warm-up. This can reduce the code size of many software pipelined loops by eliminating the need for prologue code.

`sNloop` cannot be used to eliminate the epilogue code from a pipelined loop; however, in some cases it is possible to do this through the use of programming techniques.

Typically, the issue affecting the removal of epilogue code is *load safety*. If the kernel section of a pipelined loop can safely access past the end of its arrays – either because it is known to be safe, or because the arrays have been padded at the end – then epilogue code is unnecessary. However, if load safety cannot be ensured, then explicit epilogue code is required to drain the software pipeline.

Table 7-3 shows how `sNloop` and load safety simplify the code shown in Table 7-2.

Table 7-3 Software pipelined loop (using sNloop)

```
int foo(int *A, int *result)
{
    int i;
    for (i=0;i<100;i++) {
        result[i]= A[i]*A[i];
    }
}
```

```
foo:
{ // load safety assumed
        P3 = s2loop0(.kernel,#102)      // set up pipelined loop
        R3 = R1
}
.falign
.kernel:
{       R1 = memw(R0++#4)               // kernel
        R2 = mpyi(R1,R1)
        if (P3) memw(R3++#4) = R2
}:endloop0

        jumpr lr
```

NOTE The count value that `sNloop` uses to control the `P3` setting is stored in the user status register `USR.LPCFG`.

7.4 Software branches

Unlike hardware loops, *software branches* require one processor cycle to perform a branch operation. Software branches include the following instructions:

- Jumps
- Calls
- Returns

The target address for branch instructions can be specified as register indirect or PC-relative. Branch instructions can be unconditional or conditional, with the execution of conditional instructions controlled by a predicate expression.

Table 7-4 summarizes the software branch instructions.

Table 7-4 Software branch instructions

Syntax	Operation
[if (pred_expr)] jump label [if (pred_expr)] jumpr Rs	Branch to address specified by register Rs or PC-relative offset. Can be conditionally executed.
[if (pred_expr)] call label [if (pred_expr)] callr Rs	Branch to address specified by register Rs or PC-relative offset. Store subroutine return address in link register LR. Can be conditionally executed.
[if (pred_expr)] jumpr LR	Branch to subroutine return address contained in link register LR. Can be conditionally executed.

7.4.1 Jumps

Jump instructions change the program flow to a target address which can be specified by either a register or a PC-relative immediate value. Jump instructions can be conditional based on the value of a predicate expression.

Table 7-5 lists the jump instructions.

Table 7-5 Jump instructions

Syntax	Operation
jump label	Direct jump. Branch to address specified by label. Label is encoded as PC-relative signed immediate value.
jumpr Rs	Indirect jump. Branch to address contained in general register Rs.
if ([!]Ps) jump label if ([!]Ps) jumpr Rs	Conditional jump. Perform jump if predicate expression evaluates to true.

NOTE Conditional direct jumps can be specified as speculative (Section 7.5).

7.4.2 Calls

Call instructions are used to jump to subroutines. The instruction performs a jump to the target address and also stores the return address in the link register `LR`.

The forms of call are functionally similar to jump instructions and include both PC-relative and register indirect in both unconditional and conditional forms.

Table 7-6 lists the call instructions.

Table 7-6 Call instructions

Syntax	Operation
call label	Direct subroutine call. Branch to address specified by label, and store return address in register LR. Label is encoded as PC-relative signed immediate value.
callr Rs	Indirect subroutine call. Branch to address contained in general register Rs, and store return address in register LR.
if ([!]Ps) call label if ([!]Ps) callr Rs	Conditional call. If predicate expression evaluates to true, perform subroutine call to specified target address.

7.4.3 Returns

Return instructions are used to return from a subroutine. The instruction performs an indirect jump to the subroutine return address stored in link register `LR`.

Returns are implemented as jump register indirect instructions, and support both unconditional and conditional forms.

Table 7-7 lists the return instructions.

Table 7-7 Return instructions

Syntax	Operation
jumpr LR	Subroutine return. Branch to subroutine return address contained in link register LR.
if (Ps) jumpr LR	Conditional return. If predicate Ps is true, perform subroutine return to specified target address.
if (!Ps) jumpr LR	Conditional return. If predicate Ps is false, perform subroutine return to specified target address.

NOTE The link register `LR` is aliased to general register `R31`. Therefore subroutine returns can be performed with the instruction `jumpr R31`.

7.5 Speculative jumps

Conditional instructions normally depend on a predicate which is generated in a previous instruction packet. However, the dot-new predicate (Section 6.2.3) enables conditional instructions to both generate and use a predicate in the same packet.

When the dot-new predicate is used with a conditional jump, the resulting instruction is called a *speculative jump*. For example:

```
{
   P0 = cmp.eq(R2,#16)          // single-packet compare-and-jump
   IF (P0.new) jump:t target    // ... enabled by use of P0.new
}
```

Speculative jumps require the programmer to specify a *direction hint* in the jump instruction, which predicts whether the conditional jump will be executed or skipped. If the prediction is wrong, the speculative jump instruction takes two cycles to execute instead of one (due to a pipeline stall).

Hints can improve program performance by successfully predicting how speculative jumps will execute over the course of a program: the more often the specified hint matches how the instruction actually executes, the better the overall program performance.

NOTE Hints are specified statically, while instruction execution is dynamic. The programmer must understand a program's dynamic behavior in order to specify the optimal hint for a given speculative jump instruction.

Hints are expressed in assembly language by appending the suffix "`:t`" or "`:nt`" to the `jump` instruction symbol:

- `jump:t` – The jump instruction will most often be executed ("taken")
- `jump:nt` – The jump instruction will most often be skipped ("not taken")

Table 7-8 lists the speculative jump instructions.

Table 7-8 Speculative jump instructions

Syntax	Operation
if ([!]Ps.new) jump:t label if ([!]Ps.new) jump:nt label	Speculative direct jump. If predicate expression evaluates to true, jump to address specified by label. Label is encoded as PC-relative signed immediate value. Hints :t and :nt interact with Ps.new to determine cycle count.

NOTE Speculative indirect jumps are not supported in V2.

7.6 Branches to and from packets

Instruction packets are atomic: even if they contain multiple instructions, they can be referenced only by the address of the first instruction in the packet. Therefore, branches to a packet can target only the packet's first instruction.

Packets can contain a single branch instruction which branches out of the packet.

Packets can also contain a branch instruction which branches to the first instruction in the packet. The result is effectively a single-packet software loop.

NOTE If a packet is at the end of a hardware loop, it cannot contain a branch instruction.

7.7 Exceptions

Exceptions are internally-generated disruptions to the program flow.

The Hexagon processor OS handles fatal exceptions by terminating the execution of the application system. The user is responsible for fixing the problem and recompiling their applications.

The error messages generated by exceptions include the following information to assist in locating the problem:

- Cause code – Hex value indicating the type of exception that occurred
- User IP – PC value indicating the instruction executed when exception occurred
- Bad VA – Virtual address indicating the data accessed when exception occurred

NOTE The cause code, user IP, and Bad VA values are stored in the Hexagon processor system control registers `SSR[7:0]`, `ELR`, and `BADVA` respectively.

Table 7-9 lists the cause code and Bad VA value for each exception type.

Table 7-9 Processor exceptions

Cause Code	Bad VA	Description
0x01	–	Precise BIU error (bus error, timeout, L2 parity error, etc.)
0x03	–	Double Exception (exception occurs while exception state already set)
0x11	–	Privilege violation: User mode execute to page with no execute permissions.
0x15	–	Invalid Packet. ■ Packet of more than one instruction in uncacheable memory ■ Invalid opcode or reserved instruction ■ Bad packet parse bits (packet too long, or using reserved pattern)
0x1B	–	Privilege violation: Supervisor instruction executed in user mode.
0x1C	–	Program Counter values not properly aligned
0x20	Set	Load to misaligned address
0x21	Set	Store to misaligned address
0x22	Set	Privilege violation: User-mode read to page with no read permission
0x23	Set	Privilege violation: User-mode write to page with no write permissions
0x28	–	Bad cacheability type. ■ Packet accessing multiple memory locations, with one or more of them to uncached memory. ■ `dczeroa` to page that is not write-back ■ LL/SC to write-through page
0x29	–	Packet with multiple writes to same destination register. This exception is subject to the following rules: ■ It is non-recoverable. Supervisor register ELR points to the packet that contained multiple writes; however, the registers and memory written by the offending packet are undefined. ■ It applies to the general-purpose registers (Rs), but not to the predicate or control registers. If a packet performs multiple writes to a predicate or control register, the behavior is undefined; the assembler rejects these packets. ■ It applies to two different instructions in a packet, but not to a single instruction that writes to the same destination twice, such as "R6=memw(R6++#4)". The assembler rejects this case. If the processor executes such an instruction, the behavior is undefined.

If multiple exceptions occur simultaneously, the exception with the lowest error code value has the highest exception priority.

If a packet contains multiple loads, or a load and a store, and both operations have an exception of any type, then all Slot 1 exceptions are processed before any Slot 0 exception is processed.

8 Call Stack

8.1 Overview

The Hexagon processor includes dedicated registers and instructions to support a *call stack* for subroutine execution.

The stack structure follows standard C conventions.

8.2 Stack structure

Figure 8-1 shows the stack structure.

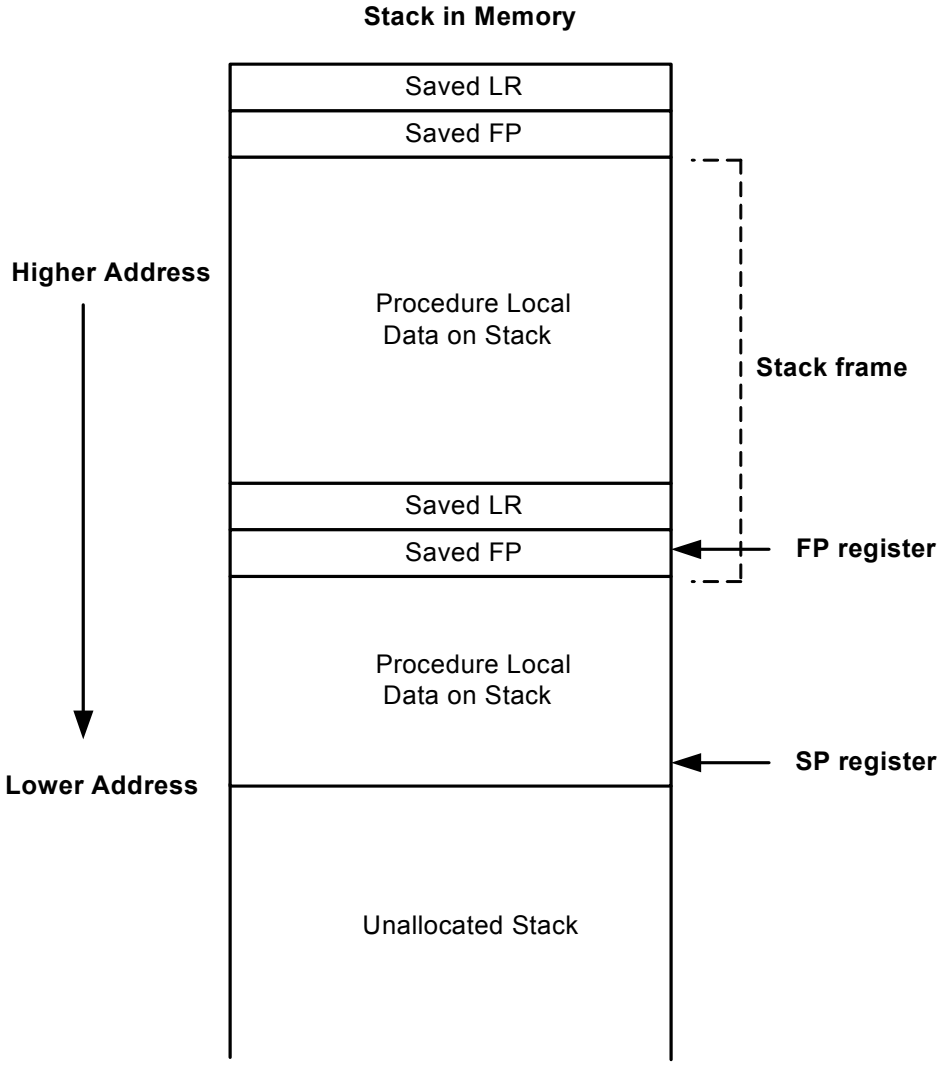


Figure 8-1 Stack structure

The stack is defined to grow from high addresses to low addresses. The stack pointer register `SP` points to the data element that is currently on the top of the stack.

NOTE The Hexagon processor supports two dedicated stack instructions: `allocframe` and `deallocframe` (Section 8.5).

The `SP` address must always remain 8-byte aligned for the stack instructions to work properly.

8.3 Stack frames

The stack is used to store *stack frames*, which are data structures that store state information on the active subroutines in a program (i.e., those that were called but have not yet returned). Each stack frame corresponds to an active subroutine in the program.

A stack frame contains the following elements:

- The local variables and data used by the subroutine
- The return address for the subroutine call (pushed from the link register `LR`)
- The address of the previous stack frame allocated on the stack (pushed from the frame pointer register `FP`)

The frame pointer register `FP` always contains the address of the previous stack frame on the stack. It facilitates debugging by enabling a debugger to examine the stack in memory and easily determine the call sequence, function parameters, etc.

8.4 Stack registers

Table 8-1 lists the stack registers.

Table 8-1 Stack registers

Register	Name	Description	Alias
SP	Stack pointer	Points to topmost stack element in memory	R29
FP	Frame pointer	Points to previous stack frame on stack	R30
LR	Link register	Contains return address of subroutine call	R31

NOTE The stack registers are aliases of three general registers (Section 2.2). These general registers are conventionally dedicated for use as stack registers.

8.5 Stack instructions

The Hexagon processor includes the instructions `allocframe` and `deallocframe` to efficiently allocate and deallocate stack frames on the call stack.

Table 8-2 describes these instructions.

Table 8-2 Stack instructions

Syntax	Operation
`allocframe(#u14)`	Allocate stack frame. This instruction is used after a call. It first pushes LR and FP to the top of stack. It then subtracts an unsigned immediate from SP in order to allocate room for local variables. FP is set to the address of the old frame pointer on the stack. The immediate value as expressed in assembly syntax specifies the byte offset. This value must be 8-byte aligned. The valid range is from 0 to 16 KB. (e.g., 11 bits are encoded, and then shifted left by 3 bits to form the constant)
`deallocframe`	Deallocate stack frame. This instruction is used just before a return in order to free a stack frame. It first loads the saved FP and saved LR values from the address at FP. It then points SP back to the previous frame.

NOTE `allocframe` and `deallocframe` load and store the `LR` and `FP` registers on the stack as a single aligned 64-bit register pair (i.e., `LR:FP`).

9 Implementation-Specific Information

9.1 Overview

This chapter describes features and programming guidelines specific to the current implementation of the V2 Hexagon processor.

It covers the following topics:

- L2 cache / TCM
- Instruction latencies
- Data access considerations
- Instruction access considerations
- Code optimization for low power

9.2 L2 cache / TCM

The V2 Hexagon processor contains an internal L2 memory which can be used as L2 cache or tightly-coupled memory (TCM). The L2 memory size is 256 KB.

The L2 memory can be partitioned between L2 cache and TCM.

From the software perspective TCM is simply a piece of physical memory. In can be used just like any other memory, but with the important property of having low-latency access from the core.

Section 9.2.2 describes the supported methods for accessing TCM.

Figure 9-1 shows a block diagram of the L2 cache for the V2 Hexagon processor.

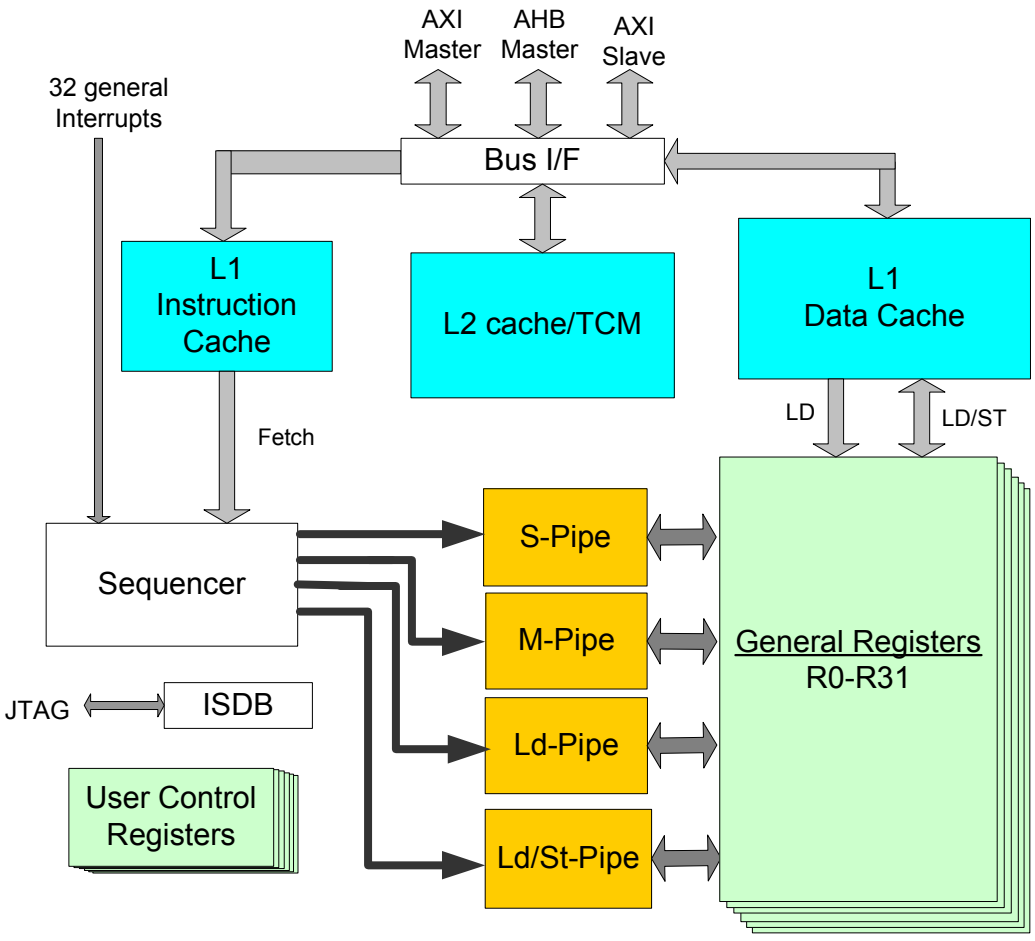


Figure 9-1 L2 memory cache

9.2.1 Using L2 memory as cache

The L2 cache has the following properties:

- 4 way set associative
- Random replacement
- 32-byte lines
- Supports Write-Though (WT) pages

When the L2 memory is configured as cache, the lines configured as L2 cacheable are copied into L2 as well as L1 on a fill. It is important to correctly mark memory as L2 cacheable/non-cacheable.

Data that is used and then unlikely to be used again should be marked non L2-cacheable. This transient data will increase the pollution in L2 and reduce its effectiveness.

Similarly, memory that is not performance- and power-critical should be excluded from L2. The L2 cache should be reserved for memory which will take the most benefit from it.

9.2.2 Accessing TCM data

The Hexagon processor L2 TCM is located at a programmable address range in the physical memory map. The default behavior is that whenever any physical address in the system is in the range 0xD8000_0000 to 0xDFFF_FFFF, that access is directed to L2 TCM.

Revision 1.0 supports a 256 KB memory, so the valid address range is from 0xD800_0000 to 0xD803_FFFF. If the address targets Hexagon processor TCM but falls outside the supported range (0xD804_0000 to 0xDFFF_FFFF by default), then a bus error condition is reported.

TCM memory can be accessed from either the Hexagon processor core or from the external slave port. Read/write accesses from the slave port originate from external hardware blocks, such as the DMA. This is described in Section 9.2.2.3.

NOTE Hexagon processor memory accesses are categorized as either uncached (Section 9.2.2.1) or cached (Section 9.2.2.2).

9.2.2.1 Uncached accesses to TCM

The Hexagon processor has a full-featured MMU that translates virtual addresses to physical addresses. Each page contains a set of cache attribute bits. To access TCM using the uncached method, the following must be true:

- The virtual page must be set to uncached type. The operating system is responsible for configuring the MMU.
- The physical page falls within the programmed range. The default is 0xD800_0000 to 0xD803_FFFF.

If a load instruction meets these conditions, then the data is brought from TCM directly into the target general purpose register. This access completely bypasses the L1 memory system. The program that initiated the uncached load to TCM stalls for at least 4 cycles. During that time, the program does not issue any other instructions.

If a store instruction meets these conditions, then the store data is sent to TCM. The bus unit contains buffers to post the store data. Assuming a buffer is available, the program that issued the uncached store does not stall; it continues on to the next packet. If a buffer is not available, the program stalls until resources become available.

The Hexagon processor supports only one uncached memory operation per packet. If the uncached memory operation is in a packet with another memory accesses, then an error exception is raised.

9.2.2.2 Cached accesses to TCM

For performance reasons, it is often desirable to cache TCM data in the level 1 data or instruction caches. This allows for dual-access zero-overhead load and store operations. To access TCM using the cached method, the following conditions should be true:

- The virtual page should be set to cacheable write-through or write-back. The Operating System is responsible for configuring the MMU cacheability types.
- The physical page falls within the programmed range. The default range is 0xD800_0000 to 0xD803_FFFF.

If a load instruction meets these conditions, the load instruction first looks for the data in the L1 data cache. If there is a cache hit, the load instruction proceeds without a stall. If a cache miss occurs, the missing line is brought from TCM into the L1 data cache. The load instruction stalls for at least 4 cycles for the line refill. The replacement is done using the normal data cache replacement policy[1], meaning that the TCM data can go anywhere in the L1 data cache.

If a store instruction meets these conditions, then the store instruction first looks for the data in the L1 data cache. If there is a cache hit, the data is written to the cache line. If the page is set to write-through, the data is also sent to TCM.

When TCM is accessed using the cached method, the hardware does not guarantee coherency between the L1 data cache and TCM. For example, if TCM data is present in the L1 cache and an external DMA changes the contents of L2 TCM, then the copy of the data in L1 is stale. Software must perform cache maintenance operations to ensure proper coherency. If the page attribute is set to write-through, then store data produced by the core is coherent with TCM. If the page is set to write-back, then stores keep dirty data in the L1 cache and do not write back to L2 TCM. For write-back, the software should use cache clean operations to push the dirty data back to L2 TCM before it can be used by external hardware.

[1] FIFO replacement is used in Rev1.0

Before external hardware writes data into TCM, the core should execute data cache invalidate operations to clear any stale data from the cache. This ensures that the new data is properly reloaded into the L1 cache.

For data that is resident in TCM and not accessed via the slave port – for example, a static data structure – a write-back operation may be preferred. In this case, there are no coherency issues, and the write-back mode can reduce traffic from L1 into TCM.

L2 TCM can be used to store code. For example, it may be desirable to keep time critical code such as interrupt handlers in TCM. If an instruction cache miss occurs and the needed instructions are in TCM (physical address in range and page marked either cacheable write-back or cacheable write-through), then the instruction cache line is refilled from TCM. The normal instruction cache replacement policy (FIFO) is used to chose a line for replacement. This means that TCM code can appear anywhere in the L1 instruction cache.

9.2.2.3 Slave TCM access

External hardware can read/write Hexagon processor TCM via the AXI slave port. Permissions to use this port are control by a Memory Protection Unit (MPU), which is described in another document. This port can be used by any hardware entity with access to the AXI bus: for example, Scorpion, DM2, the video hardware, etc.[1]

L2 TCM is fully shared between the Hexagon processor and any external hardware with access to AXI and permissions to use the slave port. Because L2 TCM is shared, software is responsible for ensuring coherent operation. For example, if the Hexagon processor and a slave port request simultaneously write to the same address in TCM, the hardware will not guarantee that the writes are performed in any specific order.

9.3 Instruction latencies

The Hexagon processor does not have any programmer-visible instruction latencies. The following list summarizes the latencies by instruction type:

- ALU32, CR, XTYPE, program flow: zero delay.
- Speculative jump with incorrect hint: 1-cycle penalty.
- Load/store: Zero delay in the common case of a cached access with a cache hit. Certain cases, such as a cache miss or bank conflict, will stall the packet (Section 9.4).
- Data prefetch: zero delay.

[1] One exception is the Hexagon processor itself, which cannot access its own TCM via the slave port. It accesses the TCM locally without incurring AXI bus transactions.

9.4 Data access considerations

This section describes implementation-specific considerations for memory access (load/store) instructions.

9.4.1 Data cache

The Hexagon processor contains a 32 KB level 1 data cache, with the following properties:

- 32 KB
- 16-way set associative
- 32-byte cache lines
- FIFO replacement policy
- Hardware next-line prefetching

The pipeline is designed so that a program that incurs a miss will stall until the miss is resolved.

To minimize the performance penalty due to data cache misses, use the following techniques:

- Applications should strive to minimize the size of key data structures. This reduces the cache working set size.
- Heavily-accessed data items should be placed next to each other in memory. This allows a single cache line miss (32 bytes) to bring in more useful data. Similarly, data structures should be organized linearly if possible.
- The `dcfetch` instruction can be used to pre-fetch data before use. This instruction does not stall the pipeline on a cache miss. Rather, the program continues running while the cache line miss is processed. In certain cases, such as if an exception is detected, the `dcfetch` instruction is ignored by the hardware. The `dcfetch` instruction should be used carefully. Pre-fetching data that is not needed unnecessarily consumes bus bandwidth.
- When needed, latency-sensitive data can be placed in L2 TCM memory (Section 9.2) to minimize performance degradation on a cache miss.

9.4.2 Bank conflicts

The data cache is pseudo-dual ported. In most cases the cache can support dual access with no penalty. Occasionally, a bank conflict stall is incurred when the cache cannot support dual access. The cases in which bank conflict occur are:

- Load/Store + Load: Conflict if address bits [31:7] do not match, but [6:3] match.
- Load + Load: No conflict if address bits [31:3] match (same line dual-access).
- Store + Load: Conflict if address bits [31:3] match and a store-load size overlap exists.

9.4.3 Page crossover

The following addressing modes compute their effective address by adding an offset value to a base address:

- Global pointer relative
- Indirect with offset
- Indirect with register offset

If the sum of the base address and offset resides on a different memory page than the base address, a single-cycle stall occurs. Consider the following example:

```
R0 = #0x3fff
R2 = memb(R0+#1)
```

In this example, the base address in R0 may reside in a different 4KB page than the sum (R0+#1). In this case a single-cycle page crossover stall occurs. Note that this stall occurs only with the base-plus-offset addressing modes – all other addressing modes do not cause page crossover stalls.

NOTE The page size is programmed in the MMU, and the stall occurs only if the sum crosses the MMU-defined page size.

9.5 Instruction access considerations

This section describes implementation-specific considerations for program fetch.

9.5.1 Instruction cache

The Hexagon processor contains a level 1 instruction cache data cache, with the following properties:

- 32 KB
- 8-way set associative
- Segmented line with two 32-byte segments
- 32-byte cache lines
- FIFO replacement policy
- Hardware next-line prefetching

A program that incurs an instruction cache miss stalls until the miss is resolved.

To minimize the performance penalty due to instruction cache misses, use the following techniques:

- Frequently accessed code should be grouped together. The profile feedback feature of the compiler can be used to sort functions in memory by how frequently they are accessed.
- Code which needs to be accessed quickly, for example interrupt handlers, can be placed in L2 TCM memory (Section 9.2) to minimize cache miss latency.

9.5.2 Fetch alignment considerations

The pipeline stalls for one cycle whenever a branch occurs to an instruction packet that crosses an aligned 16-byte boundary. If such a packet executes as the start of an inner loop, the cumulative performance penalty can become significant.

The best solution to this problem is to align such packets to 16-byte boundaries by inserting NOP instructions into the empty slots of the preceding packets. However, this can be difficult to perform manually because it requires keeping track of the alignment of every instruction.

To simplify this task, the Hexagon assembler provides the `.falign` directive. When writing assembly language code, simply insert this directive before any labels that will be jumped to frequently. The assembler will then do its best to prevent this kind of stall.

For the same reason, assembly language functions that are called frequently should be 16-byte aligned using the assembler directive "`.p2align 4`".

Figure 9-2 shows an example of how to use the `.falign` and `.p2align` directives to prevent fetch alignment stalls.

```
//
// align function and loop on 16-byte boundaries
//

   .text
   .p2align 4
   .globl freq_func
   .type   freq_func, @function
   {
      insn
      insn
      loop0(.L5,#128)
   }
   .falign
.L5:
   {
      insn
      insn
   }
   {
      insn
      insn
   }:endloop0
```

Figure 9-2 Instruction fetch alignment

9.6 Programming for low power

9.6.1 Small loops

Loops that fit within a cache line will run at lower power due to a reduction in tag lookups. Important loops should be aligned accordingly.

9.6.2 Use of vector instructions

Less power is needed to execute a vector instruction as compared to two scalar instructions that achieve the same result. For example, consider the following serial code:

```
R5 = add(R1,R3)
R4 = add(R0,R2)
```

A lower power alternative is:

```
R5:4 = vaddw(R1:0,R3:2)
```

9.6.3 Slot 3 forwarding

Slot 3 has special hardware to forward operands from one packet to the next and reduce the required register file power. Consider the following packet:

```
{
R7:6 = valignb(R9:8,R7:6,#2)
R15:14 += vdmpy(R13:12,R7:6):<<1:sat
R1 = R21
R3 = add(R0, #8)
}
{
R15:14 = vasrw(R15:14,#3)              // Scaling output
R17:16 += vdmpy(R13:12,R7:6):<<1:sat
R10 = memw(R1++#4)                     // load coeficients
}
```

In this example the register pair R7:6 that is produced by the `valignb` instruction in the first packet is used by the `vdmpy` instruction in the second packet. The assembler should try to arrange the instructions so that both the `valignb` and subsequent `vdmpy` execute in Slot 3. If this occurs, power usage is reduced.

10 Instruction Encoding

10.1 Overview

This chapter describes the binary encoding of Hexagon processor instructions and instruction packets. It covers the following topics:

- Instructions
- Instruction classes
- Instruction packets
- Loop packets
- Immediate operands
- Scaled immediate operands
- Instruction mapping

10.2 Instructions

All Hexagon processor instructions are encoded in a 32-bit instruction word. The instruction word format varies according to the instruction type.

The instruction words contain two types of bit fields:

- *Common fields* appear in every processor instruction, and are defined the same in all instructions.
- *Instruction-specific fields* appear only in some instructions, or vary in definition across the instruction set.

Table 10-1 lists the instruction bit fields.

Table 10-1 Instruction fields

Name	Description	Type
ICLASS	Instruction class	Common
Parse	Packet / loop bits	
MajOp Maj	Major opcode	Instruction-specific
MinOp Min	Minor opcode	
RegType	Register type (32-bit, 64-bit)	
Type	Operand type (byte, halfword, etc.)	
Amode	Addressing mode	
d*n*	Destination register operand	
s*n*	Source register operand	
t*n*	Source register operand #2	
x*n*	Index register operand (circ, brev, auto-inc)	
u*n*	Predicate or modifier register operand	
sH	Source register bit field (Rs.H or Rs.L)	
tH	Source register #2 bit field (Rt.H or Rt.L)	
UN	Unsigned operand	
Rs	No source register read	
P	Predicate expression	
PS	Predicate sense (Pu or !Pu)	
DN	Dot-new predicate	
PT	Predict taken	
sm	Supervisor mode only	

NOTE In some cases instruction-specific fields are used to encode instruction attributes other than the ones described for the fields in Table 10-1.

10.3 Instruction classes

The instruction class (Section 3.3) is encoded in the four most-significant bits of the instruction word (31:28). These bits are referred to as the instruction's ICLASS field.

Table 10-2 lists the encoding values for the instruction classes. The Slots column indicates which slots can receive the instruction class.

Table 10-2 Instruction class encoding

Encoding	Instruction Class	Slots
0000 0001 0010 0011	Reserved	–
0100	LD ST (conditional or GP-relative)	0,1
0101	J	2,3
0110	CR	3
0111	A32	0,1,2,3
1000	XTYPE S	2,3
1001	LD	0,1
1010	ST	0
1011	ALU32	0,1,2,3
1100	XTYPE S	2,3
1101	XTYPE ALU64	2,3
1110	XTYPE M	2,3
1111	A32	0,1,2,3

For details on encoding the individual class types see Chapter 11.

10.4 Instruction packets

Instruction packets are encoded using two bits of the instruction word (15:14), which are referred to as the instruction word's Parse field. The field values have the following definitions:

- '11' indicates that an instruction is the last instruction in a packet (i.e., the instruction word at the highest address).
- '01' or '10' indicate that an instruction is not the last instruction in a packet.
- '00' is reserved for future use.

If any instruction contains the Parse field value '00', or any sequence of four consecutive instructions occurs without one of them containing '11', the processor will raise an error exception (illegal opcode).

Figure 10-1 shows the location of the Parse field in an instruction word.

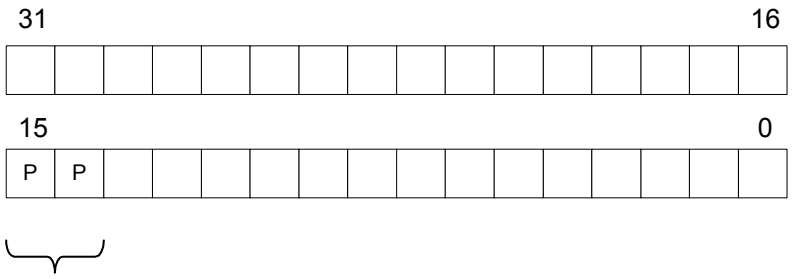


Figure 10-1 Instruction packet encoding

The following examples show how the Parse field is used to encode instruction packets:

```
{ A ; B}
  01  11              // Parse fields of instrs A,B

{ A ; B ; C}
  01  01  11          // Parse fields of instrs A,B,C

{ A ; B ; C ; D}
  01  01  01  11      // Parse fields of instrs A,B,C,D
```

10.5 Loop packets

In addition to encoding the last instruction in a packet, the instruction word's Parse field (Section 10.4) is used to encode the last packet in a hardware loop.

The Hexagon processor supports two hardware loops, labelled 0 and 1 (Section 7.3). The last packet in these loops is subject to the following restrictions:

- The last packet in a hardware loop 0 must contain two or more instructions.
- The last packet in a hardware loop 1 must contain three or more instructions.

If the last packet in a loop is expressed in assembly language with fewer than the required number of instructions, the assembler automatically adds one or two NOP instructions to the encoded packet so it contains the minimum required number of instructions.

The Parse fields in a packet's first and second instructions (i.e., the instruction words at the lowest addresses) encode whether or not the packet is the last packet in a hardware loop.

Table 10-3 shows how the Parse fields are used to encode loop packets.

Table 10-3 Loop packet encoding

Packet	Parse Field in First Instruction	Parse Field in Second Instruction
Not last in loop	01 or 11	01 or 11 [a]
Last in loop 0	10	01 or 11
Last in loop 1	01	10
Last in loops 0 & 1	10	10

[a] Not applicable for single-instruction packets.

The following examples show how the Parse field is used to encode loop packets:

```
{ A   B}:endloop0
  10  11                          // Parse fields of instrs A,B

{ A   B   C}:endloop0
  10  01  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop0
  10  01  01  11                  // Parse fields of instrs A,B,C,D

{ A   B   C}:endloop1
  01  10  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop1
  01  10  01  11                  // Parse fields of instrs A,B,C,D

{ A   B   C}:endloop0:endloop1
  10  10  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop0:endloop1
  10  10  01  11                  // Parse fields of instrs A,B,C,D
```

10.6 Immediate values

To conserve encoding space, the Hexagon processor stores immediate values in instruction fields of limited size. When an instruction operates on an immediate operand, the processor automatically extends the immediate value to the bit size required by the operation:

- Signed immediate values are *sign-extended*
- Unsigned immediate values are *zero-extended*

10.7 Scaled immediates

To minimize the number of bits used in instruction words to store certain immediate values, the Hexagon processor stores the values as *scaled immediates*. Scaled immediate values are used when an immediate value needs to represent integral multiples of 2 in a specific range.

For example, consider an instruction operand whose possible values are the following:

-32, -28, -24, -20, -16, -12, -8, -4, 0, 4, 8, 12, 16, 20, 24, 28

Normally this operand requires six bits to store (since the range -32..28 is bit-equivalent to the unsigned range 0..63, or 2^6). However, if the operand is stored as a scaled immediate, it can first be shifted right by two bits, with only the four remaining bits being stored in the instruction word. When the operand is fetched from the instruction word, the processor automatically shifts the value left by two bits to recreate the original operand value.

NOTE The scaled immediate value in the example above is represented notationally as `#s4:2`. For more information see Section 1.8.

Scaled immediate values are commonly used to encode address offsets which apply to data types of varying size. For example, Table 10-4 shows how the byte offsets used in immediate-with-offset addressing mode are stored as 11-bit scaled immediate values. This enables the offsets to span the same range of data elements regardless of the data type.

Table 10-4 Scaled immediate encoding (indirect offsets)

Data Type	Offset Size (Stored)	Scale Bits	Offset Size (Effective)	Offset Range (Bytes)	Offset Range (Elements)
byte	11	0	11	-1024 ... 1023	-1024 ... 1023
halfword	11	1	12	-2048 ... 2046	-1024 ... 1023
word	11	2	13	-4096 ... 4092	-1024 ... 1023
doubleword	11	3	14	-8192 ... 8184	-1024 ... 1023

10.8 Instruction mapping

Some Hexagon processor instructions are encoded by the assembler as variants of other instructions. This is done for operations that are functionally equivalent to other instructions, but are still defined as separate instructions because of their programming utility as common operations.

Table 10-5 lists some of the instructions that are mapped to other instructions.

Table 10-5 Instruction mapping

Instruction	Mapping
`Rd = not(Rs)`	`Rd = sub(#-1,Rs)`
`Rd = neg(Rs)`	`Rd = sub(#0,Rs)`
`Rdd = Rss`	`Rdd = combine(Rss.H32, Rss.L32)`

11 Instruction Set

11.1 Overview

This chapter describes the instruction set for version 2 of the Hexagon processor.

The instructions are listed alphabetically within instruction categories. The following information is provided for each instruction:

- Instruction name
- A brief description of the instruction
- A high-level functional description (syntax and behavior) with all possible operand types
- Instruction class and slot information for grouping instructions in packets
- Notes on miscellaneous issues
- Any C intrinsic functions that provide access to the instruction
- Instruction encoding

11.2 Instruction categories

- ALU32 — 32-bit ALU operations
  - Arithmetic and logical
  - Register transfer
  - Permute
  - Predicate
  - Vector halfword
- CR — Control registers, looping
- J — Jump
- JR — Jump from register
- LD — Load operations
- ST — Store operations
- System
- XTYPE — 32-bit and 64-bit operations
  - Arithmetic, logical, bit manipulation
  - Bit
  - Complex (multiply, round and pack, conjugate, rotate)
  - Higher precision multiply
  - Single precision multiply
  - Permute / vector permute
  - Predicate operations
  - Shift / shift with add/subtract/logical
  - Vector byte ALU
  - Vector halfword (ALU, shift, multiply)
  - Vector word (ALU, shift)

Table 11-1 summarizes symbols that are in instruction syntax.

Table 11-1 Instruction syntax symbols

Symbol	Example	Meaning
=	`R2 = R3;`	Assignment of RHS to LHS
;	`R2 = R3;`	Marks the end of an instruction or group of instructions
{ … }	`{R2 = R3; R5 = R6;}`	Indicates a group of parallel instructions.
#	`#100`	An immediate constant value
0x	`R2 = #0x1fe;`	Indicates Hexadecimal number
MEMxx	`R2 = MEMxx(R3);`	Access memory. xx specifies the size and type of access.
:sat	`R2 = add(r1,r2):sat`	Perform optional saturation
:rnd	`R2 = mpy(r1.h,r2.h):rnd`	Perform optional rounding
:<<16	`R2 = add(r1.l,r2.l):<<16`	Shift left by a halfword

Table 11-2 summarizes symbols that are used in the description of instruction operation.

Table 11-2 Instruction operations symbols

Symbol	Example	Meaning
#uN	`R2 = #u16`	Unsigned N-bit immediate value
#sN	`R2 = add(R3,#s16)`	Signed N-bit immediate value
#mN	`Rd = mpyi(Rs,#m9)`	Signed N-bit immediate value
#uN:S	`R2 = memh(#u16:1)`	Unsigned N-bit immediate value representing integral multiples of 2^S in specified range
#sN:S	`Rd = memw(Rs++#s4:2)`	Signed N-bit immediate value representing integral multiples of 2^S in specified range
#rN:S	`call #r22:2`	Same as #sN:S, but value is offset from PC of current packet

NOTE When an instruction contains more than one immediate operand, the operand symbols are specified in upper and lower case (e.g., #uN and #UN) to indicate where they appear in the instruction encodings.

The instruction behavior is specified using a superset of the C language. Table 11-3 lists symbols not defined in C which are used to specify the instruction behavior.

Table 11-3 Instruction behavior symbols

Symbol	Example	Meaning
usat_N	`usat_16(Rs)`	Saturate a value to an unsigned N-bit number
sat_N	`sat_16(Rs)`	Saturate a value to a signed N-bit number
sxt $_{x->y}$	`sxt`$_{32->64}$`(Rs)`	Sign-extend value from x to y bits
zxt $_{x->y}$	`zxt`$_{32->64}$`(Rs)`	Zero-extend value from x to y bits
>>>	`Rss >>> offset`	Logical right shift

11.3 ALU32

The ALU32 instruction class includes instructions that perform arithmetic and logical operations on 32-bit data, and register transfer operations.

11.3.1 ALU32 / Arithmetic & Logical

The ALU32 arithmetic and logical instructions operate on 32-bit data.

Add

Add a source register either to another source register or to a signed 16-bit immediate value. Store result in destination register. Source and destination registers are 32 bits. If the result overflows 32 bits, it wraps around.

For 64-bit and saturating versions of this operation, see the XTYPE add instructions.

Syntax	Behavior
`Rd=add(Rs,#s16)`	`Rd=Rs+#s;`
`Rd=add(Rs,Rt)`	`Rd=Rs+Rt;`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=add(Rs,#s16)`	`Word32 Q6_R_add_RI(Word32 Rs, Word32 Is)`
`Rd=add(Rs,Rt)`	`Word32 Q6_R_add_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS											s5					Parse											d5					
1	0	1	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,#s16)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,Rt)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Logical operations

Perform bitwise logical operations (AND, OR, XOR, NOT) either on two source registers or on a source register and a signed 10-bit immediate value. Store result in destination register. Source and destination registers are 32 bits.

For 64-bit versions of these operations, see the XTYPE logical instructions.

Syntax	Behavior
`Rd=and(Rs,#s10)`	`Rd=Rs&#s;`
`Rd=and(Rs,Rt)`	`Rd=Rs&Rt;`
`Rd=not(Rs)`	`Assembler maps to: "Rd=sub(#-1,Rs)"`
`Rd=or(Rs,#s10)`	`Rd=Rs|#s;`
`Rd=or(Rs,Rt)`	`Rd=Rs|Rt;`
`Rd=xor(Rs,Rt)`	`Rd=Rs^Rt;`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=and(Rs,#s10)`	`Word32 Q6_R_and_RI(Word32 Rs, Word32 Is)`
`Rd=and(Rs,Rt)`	`Word32 Q6_R_and_RR(Word32 Rs, Word32 Rt)`
`Rd=not(Rs)`	`Word32 Q6_R_not_R(Word32 Rs)`
`Rd=or(Rs,#s10)`	`Word32 Q6_R_or_RI(Word32 Rs, Word32 Is)`
`Rd=or(Rs,Rt)`	`Word32 Q6_R_or_RR(Word32 Rs, Word32 Rt)`
`Rd=xor(Rs,Rt)`	`Word32 Q6_R_xor_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				Rs	MajOp			MinOp			s5					Parse												d5					
0	1	1	1	0	1	1	0	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=and(Rs,#s10)
0	1	1	1	0	1	1	0	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=or(Rs,#s10)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=and(Rs,Rt)
1	1	1	1	0	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=or(Rs,Rt)
1	1	1	1	0	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=xor(Rs,Rt)

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Negate

Perform arithmetic negation on a source register. Store result in destination register. Source and destination registers are 32 bits.

For 64-bit and saturating versions of this instruction, see the XTYPE negate instructions.

Syntax	Behavior
`Rd=neg(Rs)`	`Assembler maps to: "Rd=sub(#0,Rs)"`

Type: Mapping (slots 0,1,2,3)

Intrinsics

`Rd=neg(Rs)`	`Word32 Q6_R_neg_R(Word32 Rs)`

Nop

Perform no operation. This instruction is used for padding and alignment.

Within a packet it can be positioned in any slot 0-3.

Syntax	Behavior
`nop`	

Type: ALU32 (slots 0,1,2,3)

Encoding

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	
ICLASS				Rs	MajOp											Parse																
0	1	1	1	1	1	1	1	-	-	-	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	nop

Field name	Description
`MajOp`	Major Opcode
`Rs`	No Rs read
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Subtract

Subtract a source register from either another source register or from a signed 10-bit immediate value. Store result in destination register. Source and destination registers are 32 bits. If the result underflows 32 bits, it wraps around.

For 64-bit and saturating versions of this operation, see the XTYPE subtract instructions.

Syntax	Behavior
`Rd=sub(#s10,Rs)`	`Rd=#s-Rs;`
`Rd=sub(Rt,Rs)`	`Rd=Rt-Rs;`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=sub(#s10,Rs)`	`Word32 Q6_R_sub_IR(Word32 Is, Word32 Rs)`
`Rd=sub(Rt,Rs)`	`Word32 Q6_R_sub_RR(Word32 Rt, Word32 Rs)`

Encoding

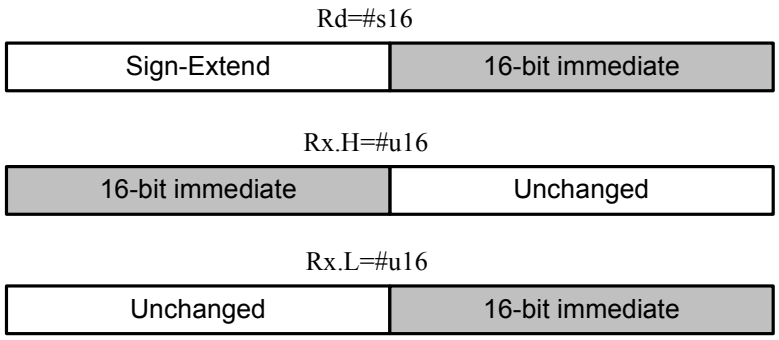
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	1	1	0	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=sub(#s10,Rs)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt,Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`p`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Transfer immediate

Assign an immediate value to a 32-bit destination register.

Two types of assignment are supported. The first sign-extends a 16-bit signed immediate value to 32 bits. The second assigns a 16-bit unsigned immediate value to either the upper or lower 16 bits of the destination register, leaving the other 16 bits unchanged.



Syntax	Behavior
`Rd=#s16`	`Rd=#s;`
`Rdd=#s8`	`if ("#s8<0") { Assembler maps to: "Rdd=combine(#-1,#s8)"; } else { Assembler maps to: "Rdd=combine(#0,#s8)"; };`
`Rx.[HL]=#u16`	`Rx.h[01]=#u;`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=#s16`	`Word32 Q6_R_equals_I(Word32 Is)`
`Rdd=#s8`	`Word64 Q6_P_equals_I(Word32 Is)`
`Rx.H=#u16`	`Word32 Q6_Rh_equals_I(Word32 Rx, Word32 Iu)`
`Rx.L=#u16`	`Word32 Q6_Rl_equals_I(Word32 Rx, Word32 Iu)`

Encoding

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	
ICLASS				Rs	MajOp			MinOp			x5					Parse																
0	1	1	1	0	0	0	1	**i**	**i**	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	Rx.L=#u16
0	1	1	1	0	0	1	0	**i**	**i**	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	Rx.H=#u16
ICLASS				Rs	MajOp			MinOp								Parse											d5					
0	1	1	1	1	0	0	0	**i**	**i**	-	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=#s16

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`x5`	Field to encode register x

Transfer register

Transfer a source register to a destination register. Source and destination registers are either 32 bits or 64 bits.

Syntax	Behavior
`Rd=Rs`	`Rd=Rs;`
`Rdd=Rss`	`Assembler maps to: "Rdd=combine(Rss.H32,Rss.L32)"`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=Rs`	`Word32 Q6_R_equals_R(Word32 Rs)`
`Rdd=Rss`	`Word64 Q6_P_equals_P(Word64 Rss)`

Encoding

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	
ICLASS				Rs	MajOp				MinOp				s5				Parse										d5					
0	1	1	1	0	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Rs

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

11.3.2 ALU32 / Permute

The ALU32 permute instructions rearrange or perform format conversion on vector data types.

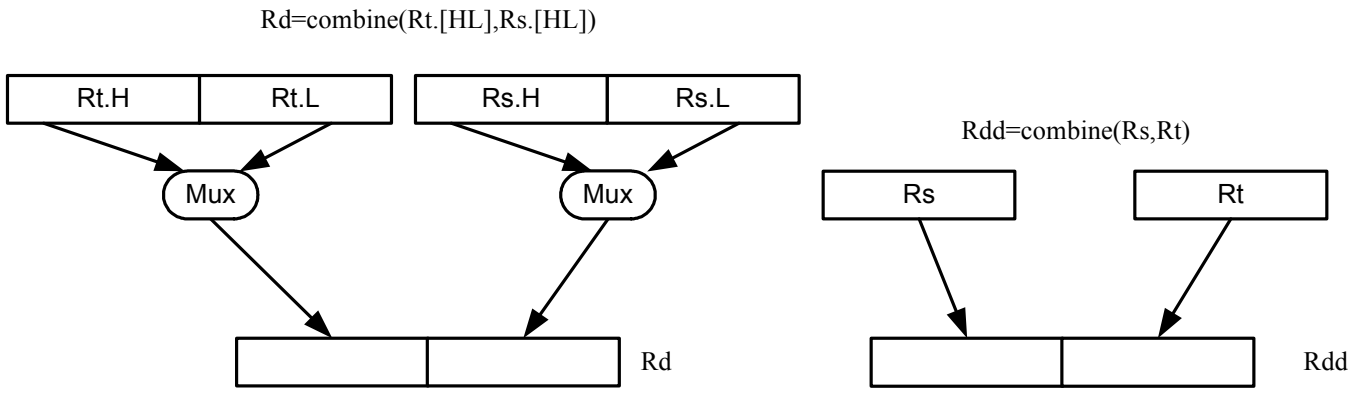
Combine

Combine halfwords or words into larger values.

In a halfword combine, either the high or low halfword of the first source register is transferred to the most-significant halfword of the destination register, while either the high or low halfword of the second source register is transferred to the least-significant halfword of the destination register. Source and destination registers are 32 bits.

In a word combine, the first source register is transferred to the most-significant word of the destination register, while the second source register is transferred to the least-significant word of the destination register. Source registers are 32 bits and the destination register is 64 bits.

In a variant of word combine, signed 8-bit immediate values (instead of registers) are transferred to the most- and least-significant words of the 64-bit destination register.



Syntax	Behavior
`Rd=combine(Rt.[HL],Rs.[HL])`	`Rd = (Rt.uh[01]<<16) | Rs.uh[01];`
`Rdd=combine(#s8,#S8)`	`Rdd.w[0]=#S;` `Rdd.w[1]=#s;`
`Rdd=combine(Rs,Rt)`	`Rdd.w[0]=Rt;` `Rdd.w[1]=Rs;`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=combine(Rt.H,Rs.H)`	`Word32 Q6_R_combine_RhRh(Word32 Rt, Word32 Rs)`
`Rd=combine(Rt.H,Rs.L)`	`Word32 Q6_R_combine_RhRl(Word32 Rt, Word32 Rs)`
`Rd=combine(Rt.L,Rs.H)`	`Word32 Q6_R_combine_RlRh(Word32 Rt, Word32 Rs)`
`Rd=combine(Rt.L,Rs.L)`	`Word32 Q6_R_combine_RlRl(Word32 Rt, Word32 Rs)`
`Rdd=combine(#s8,#S8)`	`Word64 Q6_P_combine_II(Word32 Is, Word32 IS)`
`Rdd=combine(Rs,Rt)`	`Word64 Q6_P_combine_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				Rs	MajOp			MinOp								Parse											d5					
0	1	1	1	1	1	0	0	-	l	l	l	l	l	l	l	P	P	l	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=combine(#s8,#S8)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.H,Rs.H)
1	1	1	1	0	0	1	1	1	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.H,Rs.L)
1	1	1	1	0	0	1	1	1	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.L,Rs.H)
1	1	1	1	0	0	1	1	1	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=combine(Rt.L,Rs.L)
ICLASS				P	MajOp						s5					Parse			t5								d5					
1	1	1	1	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rdd=combine(Rs,Rt)

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Mux

Select between two source registers based on the least-significant bit of a predicate register. If the bit is 1, transfer the first source register to the destination register; otherwise, transfer the second source register. Source and destination registers are 32 bits.

In a variant of mux, signed 8-bit immediate values can be used instead of registers for either or both source operands.

For 64-bit versions of this instruction, see the XTYPE vmux instruction.

Syntax	Behavior
`Rd=mux(Pu,#s8,#S8)`	`(Pu[0]) ? (Rd=#s):(Rd=#S);`
`Rd=mux(Pu,#s8,Rs)`	`(Pu[0]) ? (Rd=#s):(Rd=Rs);`
`Rd=mux(Pu,Rs,#s8)`	`(Pu[0]) ? (Rd=Rs):(Rd=#s);`
`Rd=mux(Pu,Rs,Rt)`	`(Pu[0]) ? (Rd=Rs):(Rd=Rt);`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=mux(Pu,#s8,#S8)`	`Word32 Q6_R_mux_pII(Byte Pu, Word32 Is, Word32 IS)`
`Rd=mux(Pu,#s8,Rs)`	`Word32 Q6_R_mux_pIR(Byte Pu, Word32 Is, Word32 Rs)`
`Rd=mux(Pu,Rs,#s8)`	`Word32 Q6_R_mux_pRI(Byte Pu, Word32 Rs, Word32 Is)`
`Rd=mux(Pu,Rs,Rt)`	`Word32 Q6_R_mux_pRR(Byte Pu, Word32 Rs, Word32 Rt)`

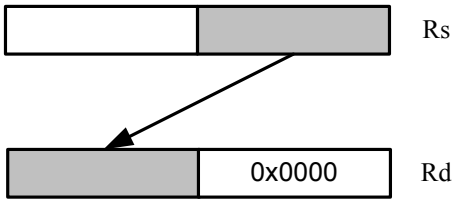
Encoding

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	
ICLASS				Rs	MajOp				u2		s5					Parse		d5														
0	1	1	1	0	0	1	1	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,Rs,#s8)
0	1	1	1	0	0	1	1	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,#s8,Rs)
ICLASS				Rs			u1									Parse		d5														
0	1	1	1	1	0	1	**u**	**u**	**l**	**l**	**l**	**l**	**l**	**l**	**l**	**P**	**P**	**l**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,#s8,#S8)
ICLASS				P	MajOp						s5					Parse			t5						u2		d5					
1	1	1	1	0	1	0	0	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	**u**	**u**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,Rs,Rt)

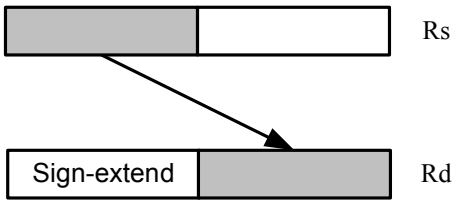
Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
MajOp	Major Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u2	Field to encode register u

Shift halfword

The ASLH instruction performs an arithmetic left shift of the 32-bit source register by 16 bits (one halfword). The lower 16-bits of the destination are zero-filled.



The ASRH instruction performs an arithmetic right shift of the 32-bit source register by 16-bits (one halfword). The upper 16-bits of the destination are sign-extended.



Syntax	Behavior
`Rd=aslh(Rs)`	`Rd=Rs<<16;`
`Rd=asrh(Rs)`	`Rd=Rs>>16;`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=aslh(Rs)`	`Word32 Q6_R_aslh_R(Word32 Rs)`
`Rd=asrh(Rs)`	`Word32 Q6_R_asrh_R(Word32 Rs)`

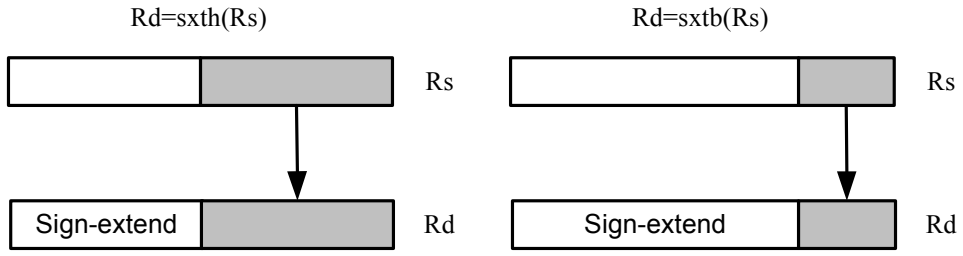
Encoding

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	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	0	0	0	0	0	s	s	s	s	s	P	P	-	-	-	-	-	-	-	-	-	d	d	d	d	d	Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	1	s	s	s	s	s	P	P	-	-	-	-	-	-	-	-	-	d	d	d	d	d	Rd=asrh(Rs)

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Sign extend

Sign-extend the least significant byte or halfword from Rs, and put the 32-bit result in Rd.



Syntax	Behavior
`Rd=sxtb(Rs)`	`Rd = sxt8->32(Rs);`
`Rd=sxth(Rs)`	`Rd = sxt16->32(Rs);`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=sxtb(Rs)`	`Word32 Q6_R_sxtb_R(Word32 Rs)`
`Rd=sxth(Rs)`	`Word32 Q6_R_sxth_R(Word32 Rs)`

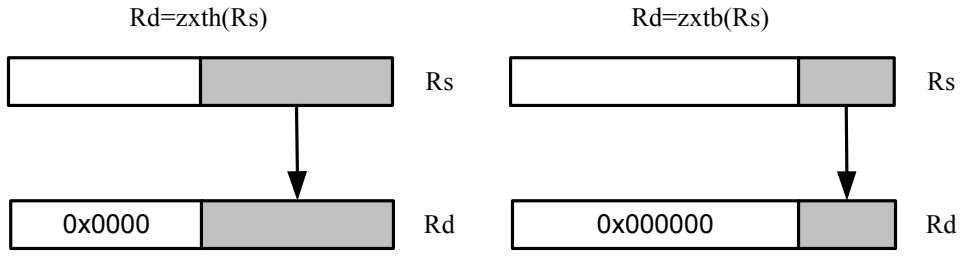
Encoding

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	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sxth(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

Zero extend

Zero-extend the least significant byte or halfword from Rs, and put the 32-bit result in Rd.



Syntax	Behavior
`Rd=zxtb(Rs)`	`Assembler maps to: "Rd=and(Rs,#255)"`
`Rd=zxth(Rs)`	`Rd = zxt16->32(Rs);`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=zxtb(Rs)`	`Word32 Q6_R_zxtb_R(Word32 Rs)`
`Rd=zxth(Rs)`	`Word32 Q6_R_zxth_R(Word32 Rs)`

Encoding

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	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=zxth(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

11.3.3 ALU32 / Predicate

The ALU32 predicate instructions perform conditional arithmetic and logical operations based on the values stored in a predicate register.

Conditional add

If the least-significant bit of predicate Pu is set, then add a 32-bit source register to either another register or an immediate value. The result is put in a 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rd=add(Rs,#s8)`	`if([!]Pu[.new][0]){ Rd=Rs+#s; } else { NOP; };`
`if ([!]Pu[.new]) Rd=add(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs+Rt; } else { NOP; };`

Type: ALU32 (slots 0,1,2,3)

Encoding

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	
ICLASS				Rs	MajOp			PS	u2		s5					Parse		DN									d5					
0	1	1	1	0	1	0	0	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=add(Rs,#s8)
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5					PS	u2		d5					
1	1	1	1	1	0	1	1	0	**-**	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	**-**	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	**-**	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	**-**	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=add(Rs,Rt)

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
DN	Dot-new
PS	Predicate sense
DN	Dot-new
MajOp	Major Opcode
MinOp	Minor Opcode

Field name	Description
P	Predicated
PS	Predicate sense
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional combine

If the least-significant bit of predicate Pu is set, then the most-significant word of destination Rdd is taken from the first source register Rs, while the least-significant word is taken from the second source register Rt. If the predicate is false, this instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rdd=combine(Rs,Rt)`	`if ([!]Pu[.new][0]) { Rdd.w[0]=Rt; Rdd.w[1]=Rs; } else { NOP; };`

Type: ALU32 (slots 0,1,2,3)

Encoding

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	
ICLASS				P	MajOp						s5					Parse		DN	t5						PS	u2		d5					
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rdd=combine(Rs,Rt)

Field name	Description
DN	Dot-new
MajOp	Major Opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional logical operations

If the least-significant bit of predicate Pu is set, then perform a logical operation on the source values. The result is put in a 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rd=and(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs&Rt; } else { NOP; };`
`if ([!]Pu[.new]) Rd=or(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs|Rt; } else { NOP; };`
`if ([!]Pu[.new]) Rd=xor(Rs,Rt)`	`if([!]Pu[.new][0]){ Rd=Rs^Rt; } else { NOP; };`

Type: ALU32 (slots 0,1,2,3)

Encoding

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	
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5					PS	u2		d5					
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=xor(Rs,Rt)

Field name	Description
DN	Dot-new
MajOp	Major Opcode
MinOp	Minor Opcode

Field name	Description
P	Predicated
PS	Predicate sense
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional subtract

If the least-significant bit of predicate Pu is set, then subtract a 32-bit source register Rt from register Rs. The result is put in a 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rd=sub(Rt,Rs)`	`if([!]Pu[.new][0]){ Rd=Rt-Rs; } else { NOP; };`

Type: ALU32 (slots 0,1,2,3)

Encoding

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	
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5						PS	u2		d5					
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=sub(Rt,Rs)

Field name	Description
DN	Dot-new
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional transfer

If the LSB of predicate Pu is set, then transfer register Rs or a signed immediate value into destination Rd. If the predicate is false, this instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rd=#s12`	`if ([!]Pu[.new][0]) Rd=#s; else NOP;`
`if ([!]Pu[.new]) Rd=Rs`	`Assembler maps to: "if ([!]Pu[.new]) Rd=add(Rs,#0)"`
`if ([!]Pu[.new]) Rdd=Rss`	`Assembler maps to: "if ([!]Pu[.new]) Rdd=combine(Rss.H32,Rss.L32)"`

Type: ALU32 (slots 0,1,2,3)

Encoding

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	
ICLASS				Rs	MajOp			PS	u2							Parse		DN									d5					
0	1	1	1	1	1	1	0	0	**u**	**u**	-	**i**	**i**	**i**	**i**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=#s12
0	1	1	1	1	1	1	0	0	**u**	**u**	-	**i**	**i**	**i**	**i**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=#s12
0	1	1	1	1	1	1	0	1	**u**	**u**	-	**i**	**i**	**i**	**i**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=#s12
0	1	1	1	1	1	1	0	1	**u**	**u**	-	**i**	**i**	**i**	**i**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=#s12

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
DN	Dot-new
PS	Predicate sense
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
u2	Field to encode register u

Compare

The register form compares two 32-bit registers for unsigned greater than, greater than, or equal.

The immediate form compares a register against a signed or unsigned immediate value. The 8-bit predicate register Pd is set to all 1's or all 0's depending on the result. For 64-bit versions of this instruction, see the XTYPE compare instructions.

Syntax	Behavior
`Pd=cmp.eq(Rs,#s10)`	`Pd=(Rs==#s) ? 0xff : 0x00;`
`Pd=cmp.eq(Rs,Rt)`	`Pd=(Rs==Rt) ? 0xff : 0x00;`
`Pd=cmp.ge(Rs,#s8)`	`Assembler maps to: "Pd=cmp.gt(Rs,#s8-1)"`
`Pd=cmp.geu(Rs,#u8)`	`if ("#u8==0") { Assembler maps to: "Pd=cmp.eq(Rs,Rs)"; } else { Assembler maps to: "Pd=cmp.gtu(Rs,#u8-1)"; };`
`Pd=cmp.gt(Rs,#s10)`	`Pd=(Rs>#s) ? 0xff : 0x00;`
`Pd=cmp.gt(Rs,Rt)`	`Pd=(Rs>Rt) ? 0xff : 0x00;`
`Pd=cmp.gtu(Rs,#u9)`	`Pd=(Rs.uw[0]>#u) ? 0xff : 0x00;`
`Pd=cmp.gtu(Rs,Rt)`	`Pd=(Rs.uw[0]>Rt.uw[0]) ? 0xff : 0x00;`
`Pd=cmp.lt(Rs,Rt)`	`Assembler maps to: "Pd=cmp.gt(Rt,Rs)"`
`Pd=cmp.ltu(Rs,Rt)`	`Assembler maps to: "Pd=cmp.gtu(Rt,Rs)"`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Pd=cmp.eq(Rs,#s10)`	`Byte Q6_p_cmp_eq_RI(Word32 Rs, Word32 Is)`
`Pd=cmp.eq(Rs,Rt)`	`Byte Q6_p_cmp_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.ge(Rs,#s8)`	`Byte Q6_p_cmp_ge_RI(Word32 Rs, Word32 Is)`
`Pd=cmp.geu(Rs,#u8)`	`Byte Q6_p_cmp_geu_RI(Word32 Rs, Word32 Iu)`
`Pd=cmp.gt(Rs,#s10)`	`Byte Q6_p_cmp_gt_RI(Word32 Rs, Word32 Is)`

`Pd=cmp.gt(Rs,Rt)`	`Byte Q6_p_cmp_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.gtu(Rs,#u9)`	`Byte Q6_p_cmp_gtu_RI(Word32 Rs, Word32 Iu)`
`Pd=cmp.gtu(Rs,Rt)`	`Byte Q6_p_cmp_gtu_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.lt(Rs,Rt)`	`Byte Q6_p_cmp_lt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.ltu(Rs,Rt)`	`Byte Q6_p_cmp_ltu_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				Rs	MajOp			MinOp			s5					Parse		d2															
0	1	1	1	0	1	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	-	-	**d**	**d**	Pd=cmp.eq(Rs,#s10)
0	1	1	1	0	1	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	-	-	**d**	**d**	Pd=cmp.gt(Rs,#s10)
0	1	1	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	-	-	**d**	**d**	Pd=cmp.gtu(Rs,#u9)
ICLASS				P	MajOp			MinOp			s5					Parse			t5											d2		
1	1	1	1	0	0	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=cmp.gt(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=cmp.gtu(Rs,Rt)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

11.3.4 ALU32 / Vector Halfword

Two 16-bit half-word values can be packed in a single 32-bit register.

The ALU32 vector halfword instructions perform arithmetic and logical operations in parallel on 32-bit half-word vectors.

Vector add halfwords

Add the two 16-bit halfwords of Rs to the two 16-bit halfwords of Rt. Optionally saturate the results to signed or unsigned 16-bit values.

Syntax	Behavior

`Rd=vaddh(Rs,Rt)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=[sat_16](Rs.h[i]+Rt.h[i]);`
	`};`

`Rd=vadduh(Rs,Rt):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat_16(Rs.uh[i]+Rt.uh[i]);`
	`};`

Type: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vaddh(Rs,Rt)`	`Word32 Q6_R_vaddh_RR(Word32 Rs, Word32 Rt)`
`Rd=vaddh(Rs,Rt):sat`	`Word32 Q6_R_vaddh_RR_sat(Word32 Rs, Word32 Rt)`
`Rd=vadduh(Rs,Rt):sat`	`Word32 Q6_R_vadduh_RR_sat(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vaddh(Rs,Rt)
1	1	1	1	0	1	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vaddh(Rs,Rt):sat
1	1	1	1	0	1	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vadduh(Rs,Rt):sat

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Field name	Description
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector average halfwords

VAVGH adds the two 16-bit halfwords of Rs to the two 16-bit halfwords of Rd, and shifts the result right by 1 bit. Optionally, a rounding constant is added before shifting.

VNAVGH subtracts the two 16-bit halfwords of Rt from the two 16-bit halfwords of Rs, and shifts the result right by 1 bit. For vector negative average with rounding, see the XTYPE VNAVGH instruction.

Syntax	Behavior

`Rd=vavgh(Rs,Rt)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rs.h[i]+Rt.h[i])>>1);`
	`};`

`Rd=vavgh(Rs,Rt):rnd`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rs.h[i]+Rt.h[i]+1)>>1);`
	`};`

`Rd=vnavgh(Rt,Rs)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rt.h[i]-Rs.h[i])>>1);`
	`};`

Type: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=vavgh(Rs,Rt)`	`Word32 Q6_R_vavgh_RR(Word32 Rs, Word32 Rt)`
`Rd=vavgh(Rs,Rt):rnd`	`Word32 Q6_R_vavgh_RR_rnd(Word32 Rs, Word32 Rt)`
`Rd=vnavgh(Rt,Rs)`	`Word32 Q6_R_vnavgh_RR(Word32 Rt, Word32 Rs)`

Encoding

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	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	1	-	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vavgh(Rs,Rt)
1	1	1	1	0	1	1	1	-	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vavgh(Rs,Rt):rnd
1	1	1	1	0	1	1	1	-	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vnavgh(Rt,Rs)

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
ICLASS	Instruction Class

Field name	Description
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector subtract halfwords

Subtract each of the two halfwords in 32-bit vector Rs from the corresponding halfword in vector Rt. Optionally saturate each 16-bit addition to either a signed or unsigned 16-bit value.

Applying saturation to the VSUBH instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to VSUBUH ensures that the unsigned result is in the range 0 to 0xffff. When saturation is not needed, VSUBH should be used.

Syntax	Behavior

`Rd=vsubh(Rt,Rs)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=[sat_16](Rt.h[i]-Rs.h[i]);`
	`};`

`Rd=vsubuh(Rt,Rs):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat_16(Rt.uh[i]-Rs.uh[i]);`
	`};`

Type: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF remains set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vsubh(Rt,Rs)`	`Word32 Q6_R_vsubh_RR(Word32 Rt, Word32 Rs)`
`Rd=vsubh(Rt,Rs):sat`	`Word32 Q6_R_vsubh_RR_sat(Word32 Rt, Word32 Rs)`
`Rd=vsubuh(Rt,Rs):sat`	`Word32 Q6_R_vsubuh_RR_sat(Word32 Rt, Word32 Rs)`

Encoding

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	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubh(Rt,Rs)
1	1	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubh(Rt,Rs):sat
1	1	1	1	0	1	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubuh(Rt,Rs):sat

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

11.4 CR

The CR instruction class includes instructions used for looping and control register transfers.

Logical reductions on predicates

The ANY8 instruction sets a destination predicate register to 0xff if *any* of the low 8 bits in source predicate register Ps are set. Otherwise, the predicate is set to 0x00.

The ALL8 instruction sets a destination predicate register to 0xff if *all* of the low 8 bits in the source predicate register Ps are set. Otherwise, the predicate is set to 0x00.

Syntax	Behavior
`Pd=all8(Ps)`	`(Ps==0xff) ? (Pd=0xff) : (Pd=0x00);`
`Pd=any8(Ps)`	`Ps ? (Pd=0xff) : (Pd=0x00);`

Type: CR (slots 2,3)

Notes

- This instruction can execute in slot 2 or slot 3, even though it is a CR instruction.

Intrinsics

`Pd=all8(Ps)`	`Byte Q6_p_all8_p(Byte Ps)`
`Pd=any8(Ps)`	`Byte Q6_p_any8_p(Byte Ps)`

Encoding

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	
ICLASS				rs	sm									s2		Parse														d2		
0	1	1	0	1	0	1	1	1	0	0	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=any8(Ps)
0	1	1	0	1	0	1	1	1	0	1	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=all8(Ps)

Field name	Description
`rs`	Read register Rs
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s

Loop instructions

loopN is a single instruction which sets up a hardware loop. The N in the instruction name indicates the set of loop registers to use. Loop0 is the innermost loop, while loop1 is the outer loop. This instruction first sets the Start Address (SA) register based on a PC-relative immediate add. The relative immediate is added to the PC and stored in SA. The Loop Count (LC) is set to either an unsigned immediate or to a register value.

endloopN is an instruction which marks the end of a hardware loop. If the Loop Count (LC) register indicates that a loop should continue to iterate, the LC register is decremented and the program flow changes to the address in the Start Address (SA) register. endloopN is encoded in bits 15:14 of the instruction it is appended to. Therefore, no distinct 32-bit encoding exists for this symbol.

Syntax	Behavior

`:endloop0`	`if (USR.LPCFG) {`
	`  if (USR.LPCFG==1) {`
	`  P3=0xff;`
	`  };`
	`  USR.LPCFG=USR.LPCFG-1;`
	`};`
	`if (LC0>1) {`
	`  PC=SA0;`
	`  LC0=LC0-1;`
	`};`

`:endloop0:endloop1`	`if (USR.LPCFG) {`
	`  if (USR.LPCFG==1) {`
	`  P3=0xff;`
	`  };`
	`  USR.LPCFG=USR.LPCFG-1;`
	`};`
	`if (LC0>1) {`
	`  PC=SA0;`
	`  LC0=LC0-1;`
	`} else {`
	`  if (LC1>1) {`
	`  PC=SA1;`
	`  LC1=LC1-1;`
	`  };`
	`};`

Syntax	Behavior

`:endloop1`	`if (LC1>1) {`
	`  PC=SA1;`
	`  LC1=LC1-1;`
	`};`

`loop0(#r7:2,#U10)`	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=0;`

`loop0(#r7:2,Rs)`	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=0;`

`loop1(#r7:2,#U10)`	`SA1=PC+#r;` `LC1=#U;`
`loop1(#r7:2,Rs)`	`SA1=PC+#r;` `LC1=Rs;`

Type: CR (slot 3)

Notes

- A maximum of one PC-plus-offset calculation is allowed per packet. Loop instructions count towards this maximum.
- This instruction cannot execute in the last address of a hardware loop.
- No program flow instructions are allowed in a packet with this instruction.
- The Next PC value is the address immediately following the last instruction in the packet containing the endloopN modifier.
- The PC value is the address of the start of the packet.
- A PC-relative address is formed by taking the encoded signed immediate value, shifting it left by 1 bit, and adding it to the current PC value.

Encoding

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	
ICLASS				rs	sm						s5					Parse																
0	1	1	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	-	-	-	loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	-	-	-	loop1(#r7:2,Rs)
ICLASS				rs	sm											Parse																
0	1	1	0	1	0	0	1	0	0	0	**I**	**I**	**I**	**I**	**I**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**i**	**i**	-	**I**	**I**	loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	0	0	1	**I**	**I**	**I**	**I**	**I**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**i**	**i**	-	**I**	**I**	loop1(#r7:2,#U10)

Field name	Description
rs	Read register Rs
sm	Supervisor mode only
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s

Pipelined loop instructions

sNloop is a single instruction which sets up a hardware loop with automatic predicate control. This features saves code size by enabling many software pipelined loops to be generated without prologue code. Upon executing this instruction, the P3 register is automatically cleared. After the loop has been executed N times (where N is selectable from 1-3), the P3 register is set. The intent is that store instructions in the loop are predicated with P3 and thus not enabled during the pipeline warm-up.

In the sNloop instruction the loop 0 (inner-loop) registers are used. This instruction first sets the Start Address (SA0) register based on a PC-relative immediate add. The relative immediate is added to the PC and stored in SA0. The Loop Count (LC0) is set to either an unsigned immediate or to a register value. The predicate P3 is cleared. The USR.LPCFG bits are set based on the N value.

It is illegal to group the spNloop instruction together with a program flow instruction in the same packet.

Syntax	Behavior

`p3=s1loop0(#r7:2,#U10)`	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=1;`
	`P3=0;`

`p3=s1loop0(#r7:2,Rs)`	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=1;`
	`P3=0;`

`p3=s2loop0(#r7:2,#U10)`	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=2;`
	`P3=0;`

`p3=s2loop0(#r7:2,Rs)`	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=2;`
	`P3=0;`

`p3=s3loop0(#r7:2,#U10)`	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=3;`
	`P3=0;`

`p3=s3loop0(#r7:2,Rs)`	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=3;`
	`P3=0;`

Type: CR (slot 3)

Notes

- A maximum of one PC-plus-offset calculation is allowed per packet. Loop instructions count towards this maximum.
- This instruction cannot execute in the last address of a hardware loop.
- The Next PC value is the address immediately following the last instruction in the packet
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the encoded signed immediate value, shifting it left by 1 bit, and adding it to the current PC value.

Encoding

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	
ICLASS				rs	sm						s5					Parse																
0	1	1	0	0	0	0	0	1	0	1	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	p3=sp1loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	1	1	0	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	p3=sp2loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	1	1	1	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	p3=sp3loop0(#r7:2,Rs)
ICLASS				rs	sm											Parse																
0	1	1	0	1	0	0	1	1	0	1	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	p3=sp1loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	1	1	0	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	p3=sp2loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	1	1	1	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	p3=sp3loop0(#r7:2,#U10)

Field name	Description
rs	Read register Rs
sm	Supervisor mode only
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s

Logical operations on predicates

These instructions perform bitwise logical operations on predicate registers.

Syntax	Behavior
`Pd=Ps`	`Assembler maps to: "Pd=or(Ps,Ps)"`
`Pd=and(Ps,Pt)`	`Pd=Ps & Pt;`
`Pd=and(Pt,!Ps)`	`Pd=Pt & (~Ps);`
`Pd=not(Ps)`	`Pd=~Ps;`
`Pd=or(Ps,Pt)`	`Pd=Ps | Pt;`
`Pd=or(Pt,!Ps)`	`Pd=Pt | (~Ps);`
`Pd=xor(Ps,Pt)`	`Pd=Ps ^ Pt;`

Type: CR (slots 2,3)

Notes

- This instruction can execute in slot 2 or slot 3, even though it is a CR instruction.

Intrinsics

`Pd=Ps`	`Byte Q6_p_equals_p(Byte Ps)`
`Pd=and(Ps,Pt)`	`Byte Q6_p_and_pp(Byte Ps, Byte Pt)`
`Pd=and(Pt,!Ps)`	`Byte Q6_p_and_pnp(Byte Pt, Byte Ps)`
`Pd=not(Ps)`	`Byte Q6_p_not_p(Byte Ps)`
`Pd=or(Ps,Pt)`	`Byte Q6_p_or_pp(Byte Ps, Byte Pt)`
`Pd=or(Pt,!Ps)`	`Byte Q6_p_or_pnp(Byte Pt, Byte Ps)`
`Pd=xor(Ps,Pt)`	`Byte Q6_p_xor_pp(Byte Ps, Byte Pt)`

Encoding

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	
ICLASS				rs	sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	0	0	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=and(Ps,Pt)
0	1	1	0	1	0	1	1	0	0	1	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=or(Ps,Pt)
0	1	1	0	1	0	1	1	0	1	0	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=xor(Ps,Pt)

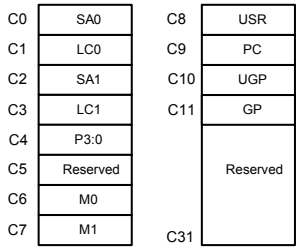
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	0	1	0	1	1	0	1	1	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=and(Pt,!Ps)
ICLASS				rs	sm									s2		Parse														d2		
0	1	1	0	1	0	1	1	1	1	0	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=not(Ps)
ICLASS				rs	sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	1	1	1	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=or(Pt,!Ps)

Field name	Description
`rs`	Read register Rs
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s
`t2`	Field to encode register t

User control register transfer

Move 32-bit values between a user control register and a general register. The user control registers include SA, LC, Predicates, M, USR, PC, and UGP. The figure shows the user control registers and their register field encodings.

Note that the PC register is not writable. A program flow instruction must be used to change the PC value.



Syntax	Behavior
`Cd=Rs`	`Cd=Rs;`
`Rd=Cs`	`Rd=Cs;`

Type: CR (slot 3)

Encoding

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	
ICLASS				rs	sm						s5					Parse											d5					
0	1	1	0	0	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Cd=Rs
0	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Cs

Field name	Description
`rs`	Read register Rs
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

11.5 JR

The JR instruction class includes branch instructions (jumps and calls) that obtain the target address from a register operand.

Call subroutine from register

Change the program flow to a subroutine. This instruction first transfers the Next Program Counter (NPC) value into the Link Register, and then jumps to a target address contained in a register.

This instruction can appear only in slot 2.

Syntax	Behavior

`callr Rs`	`LR=NPC;`
	`PC=Rs;`
	`;`

`if ([!]Pu) callr Rs`	`if ([!]Pu[0]) {`
	`  LR=NPC;`
	`  PC=Rs;`
	`;`
	`};`

Type: JR (slot 2)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', it is executed only if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', it is executed only if the least-significant bit of Pn is 0.

Encoding

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	
ICLASS											s5					Parse																	
0	1	0	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	callr Rs
ICLASS											s5					Parse						u2										
0	1	0	1	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) callr Rs
0	1	0	1	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) callr Rs

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
u2	Field to encode register u

Jump to address from register

Change the program flow to a target address. This instruction changes the Program Counter to a target address contained in a register.

This instruction can appear only in slot 2.

Syntax	Behavior

`if ([!]Pu) jumpr Rs`	`if ([!]Pu[0]) {`
	`  PC=Rs;`
	`};`

`jumpr Rs`	`PC=Rs;`

Type: JR (slot 2)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', it is executed only if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', it is executed only if the least-significant bit of Pn is 0.

Encoding

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	
ICLASS											s5					Parse																	
0	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	jumpr Rs
ICLASS											s5					Parse						u2										
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) jumpr Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) jumpr Rs

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
u2	Field to encode register u

11.6 J

The J instruction class includes branch instructions (jumps and calls) that obtain the target address from a (PC-relative) immediate address value.

Call subroutine

Change the program flow to a subroutine. This instruction first transfers the Next Program Counter (NPC) value into the Link Register, and then jumps to the target address.

This instruction can appear in slots 2 or 3.

Syntax	Behavior

`call #r22:2`	`LR=NPC;`
	`PC=PC+#r;`
	`;`

`if ([!]Pu) call #r15:2`	`if ([!]Pu[0]) {`
	`  LR=NPC;`
	`  PC=PC+#r;`
	`;`
	`};`

Type: J (slots 2,3)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', it is executed only if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', it is executed only if the least-significant bit of Pn is 0.
- The Next PC value is the address immediately following the last instruction in the packet
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the encoded signed immediate value, shifting it left by 1 bit, and adding it to the current PC value.

Encoding

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	
ICLASS																Parse																
0	1	0	1	1	0	1	i	i	i	i	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	-	call #r22:2
ICLASS																Parse			PT	D N		u2										
0	1	0	1	1	1	0	1	i	i	0	i	i	i	i	i	P	P	i	0	0	-	u	u	i	i	i	i	i	i	i	-	if (Pu) call #r15:2
0	1	0	1	1	1	0	1	i	i	1	i	i	i	i	i	P	P	i	0	0	-	u	u	i	i	i	i	i	i	i	-	if (!Pu) call #r15:2

Field name	Description
ICLASS	Instruction Class
DN	Dot-new

Field name	Description
PT	Predict-taken
Parse	Packet/Loop parse bits
u2	Field to encode register u

Jump to address

Change the program flow to a target address. This instruction changes the Program Counter to a target address which is relative to the PC address. The offset from the current PC address is contained in the instruction encoding.

This instruction can appear in slots 2 or 3.

Syntax	Behavior

`if ([!]Pu) jump #r15:2`	`if ([!]Pu[0]) {`
	`  PC=PC+#r;`
	`};`

`jump #r22:2`	`PC=PC+#r;`

Type: J (slots 2,3)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', it is executed only if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', it is executed only if the least-significant bit of Pn is 0.

Encoding

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	
ICLASS																Parse																
0	1	0	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	jump #r22:2
ICLASS																Parse			PT	D N		u2										
0	1	0	1	1	1	0	0	**i**	**i**	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	0	0	**-**	**u**	**u**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Pu) jump #r15:2
0	1	0	1	1	1	0	0	**i**	**i**	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	0	0	**-**	**u**	**u**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!Pu) jump #r15:2

Field name	Description
ICLASS	Instruction Class
DN	Dot-new
PT	Predict-taken
Parse	Packet/Loop parse bits
u2	Field to encode register u

Jump to address conditioned on new predicate

Perform speculated jump.

Jump if the LSB of the newly-generated predicate is true. The predicate must be generated in the same packet as the speculated jump instruction.

A speculated jump instruction includes a hint (true or not true) which specifies the expected value of the predicate. If the actual generated value of the predicate differs from this expected value, the jump instruction incurs a performance penalty.

This instruction can appear in slots 2 or 3.

Syntax	Behavior

`if ([!]Pu.new) jump:nt #r15:2`	`check_new()};`
	`{`
	`if([!]Pu.new[0]){`
	`  PC=PC+#r;`
	`;`
	`};`

`if ([!]Pu.new) jump:t #r15:2`	`check_new()};`
	`{`
	`if([!]Pu.new[0]){`
	`  PC=PC+#r;`
	`;`
	`};`

Type: J (slots 2,3)

Notes

- This instruction is conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', it is executed only if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', it is executed only if the least-significant bit of Pn is 0.

Encoding

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	
ICLASS																Parse			PT	D N		u2										
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	0	1	-	u	u	i	i	i	i	i	i	i	-	if (Pu.new) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	1	1	-	u	u	i	i	i	i	i	i	i	-	if (Pu.new) jump:t #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	0	1	-	u	u	i	i	i	i	i	i	i	-	if (!Pu.new) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	1	1	-	u	u	i	i	i	i	i	i	i	-	if (!Pu.new) jump:t #r15:2

Field name	Description
ICLASS	Instruction Class
DN	Dot-new
PT	Predict-taken
Parse	Packet/Loop parse bits
u2	Field to encode register u

11.7 LD

The LD instruction class includes the load instructions, which are used to load memory values into registers.

Load doubleword

Load a 64-bit doubleword from memory and place in a destination register pair.

Syntax	Behavior
`Rdd=memd(Rs)`	`EA=Rs+#0;` `Rdd = *EA;`
`Rdd=memd(Rs+#s11:3)`	`EA=Rs+#s;` `Rdd = *EA;`

`Rdd=memd(Rx++#s4:3)`	`EA=Rx;`
	`Rdd = *EA;`
	`Rx=Rx+#s;`

`Rdd=memd(Rx++#s4:3:circ(Mu))`	`EA=Rx;`
	`Rdd = *EA;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rdd=memd(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rdd = *EA;`
	`Rx=circ_add(Rx,I<<3,MuV);`

`Rdd=memd(Rx++Mu)`	`EA=Rx;`
	`Rdd = *EA;`
	`Rx=Rx+MuV;`

`Rdd=memd(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rdd = *EA;`
	`Rx=Rx+MuV;`

`Rdd=memd(#u16:3)`	`EA=(GP | #u);` `Rdd = *EA;`

Type: LD (slots 0,1)

Encoding

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	
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	i	i	1	1	1	0	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=memd(#u16:3)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	i	i	1	1	1	0	s	s	s	s	s	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=memd(Rs+#s11:3)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	1	1	0	x	x	x	x	x	P	P	u	0	-	-	0	i	i	i	i	d	d	d	d	d	Rdd=memd(Rx++#s4:3:circ(Mu))
1	0	0	1	1	0	0	1	1	1	0	x	x	x	x	x	P	P	u	0	-	-	1	-	-	-	-	d	d	d	d	d	Rdd=memd(Rx++I:circ(Mu))

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	
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++#s4:3)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++Mu)
1	0	0	1	1	1	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u1	Field to encode register u
x5	Field to encode register x

Load doubleword conditionally

Load a 64-bit doubleword from memory and place in a destination register pair.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pt) Rdd=memd(Rx++#s4:3)`	`if([!]Pt[0]){ EA=Rx; Rdd = *EA; Rx=Rx+#s; } else { NOP; };`
`if ([!]Pt[.new]) Rdd=memd(Rs+#u6:3)`	`if ([!]Pt[.new][0]) { EA=Rs+#u; Rdd = *EA; } else { NOP; };`

Type: LD (slots 0,1)

Encoding

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	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rdd=memd(Rs+#u6:3)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	-	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(Rx++#s4:3)
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	-	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(Rx++#s4:3)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned

Field name	Description
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
x5	Field to encode register x

Load byte

Load a signed byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then sign extended from 8 bits to 32.

Syntax	Behavior
`Rd=memb(Rs)`	`EA=Rs+#0;` `Rd = *EA;`
`Rd=memb(Rs+#s11:0)`	`EA=Rs+#s;` `Rd = *EA;`

`Rd=memb(Rx++#s4:0)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+#s;`

`Rd=memb(Rx++#s4:0:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rd=memb(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,I<<0,MuV);`

`Rd=memb(Rx++Mu)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memb(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memb(#u16:0)`	`EA=(GP | #u);` `Rd = *EA;`

Type: LD (slots 0,1)

Encoding

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	
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(#u16:0)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rs+#s11:0)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++#s4:0:circ(Mu))
1	0	0	1	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	1	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++I:circ(Mu))

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	
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++#s4:0)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++Mu)
1	0	0	1	1	1	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u1	Field to encode register u
x5	Field to encode register x

Load byte conditionally

Load a signed byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then sign-extended from 8 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pt) Rd=memb(Rx++#s4:0)`	`if([!]Pt[0]){ EA=Rx; Rd = *EA; Rx=Rx+#s; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memb(Rs)`	`if ([!]Pt[.new][0]) { EA=Rs+#0; Rd = *EA; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memb(Rs+#u6:0)`	`if ([!]Pt[.new][0]) { EA=Rs+#u; Rd = *EA; } else { NOP; };`

Type: LD (slots 0,1)

Encoding

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	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(Rs+#u6:0)
0	1	0	0	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memb(Rs+#u6:0)
0	1	0	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(Rs+#u6:0)
0	1	0	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memb(Rs+#u6:0)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**-**	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**-**	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(Rx++#s4:0)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
x5	Field to encode register x

Load halfword

Load a signed halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is then sign extended from 16 bits to 32.

Syntax	Behavior
`Rd=memh(Rs)`	`EA=Rs+#0;` `Rd = *EA;`
`Rd=memh(Rs+#s11:1)`	`EA=Rs+#s;` `Rd = *EA;`

`Rd=memh(Rx++#s4:1)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+#s;`

`Rd=memh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rd=memh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,I<<1,MuV);`

`Rd=memh(Rx++Mu)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memh(#u16:1)`	`EA=(GP | #u);` `Rd = *EA;`

Type: LD (slots 0,1)

Encoding

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	
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(#u16:1)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rs+#s11:1)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++I:circ(Mu))

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	
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++#s4:1)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++Mu)
1	0	0	1	1	1	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u1	Field to encode register u
x5	Field to encode register x

Load halfword conditionally

Load a signed halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is then sign extended from 16 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pt) Rd=memh(Rx++#s4:1)`	`if([!]Pt[0]){ EA=Rx; Rd = *EA; Rx=Rx+#s; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memh(Rs)`	`if ([!]Pt[.new][0]) { EA=Rs+#0; Rd = *EA; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memh(Rs+#u6:1)`	`if ([!]Pt[.new][0]) { EA=Rs+#u; Rd = *EA; } else { NOP; };`

Type: LD (slots 0,1)

Encoding

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	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	1	0	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (Pt) Rd=memh(Rs+#u6:1)
0	1	0	0	0	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (Pt.new) Rd=memh(Rs+#u6:1)
0	1	0	0	0	1	0	1	0	1	0	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (!Pt) Rd=memh(Rs+#u6:1)
0	1	0	0	0	1	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (!Pt.new) Rd=memh(Rs+#u6:1)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	1	0	x	x	x	x	x	P	P	1	-	0	t	t	i	i	i	i	d	d	d	d	d	if (Pt) Rd=memh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	0	x	x	x	x	x	P	P	1	-	1	t	t	i	i	i	i	d	d	d	d	d	if (!Pt) Rd=memh(Rx++#s4:1)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
x5	Field to encode register x

Load unsigned byte

Load an unsigned byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then zero extended from 8 bits to 32.

Syntax	Behavior
`Rd=memub(Rs)`	`EA=Rs+#0;` `Rd = *EA;`
`Rd=memub(Rs+#s11:0)`	`EA=Rs+#s;` `Rd = *EA;`

`Rd=memub(Rx++#s4:0)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+#s;`

`Rd=memub(Rx++#s4:0:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rd=memub(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,I<<0,MuV);`

`Rd=memub(Rx++Mu)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memub(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memub(#u16:0)`	`EA=(GP | #u);` `Rd = *EA;`

Type: LD (slots 0,1)

Encoding

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	
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(#u16:0)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rs+#s11:0)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++#s4:0:circ(Mu))
1	0	0	1	1	0	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++I:circ(Mu))

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	
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++#s4:0)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++Mu)
1	0	0	1	1	1	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u1	Field to encode register u
x5	Field to encode register x

Load unsigned byte conditionally

Load an unsigned byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then zero extended from 8 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pt) Rd=memub(Rx++#s4:0)`	`if([!]Pt[0]){ EA=Rx; Rd = *EA; Rx=Rx+#s; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memub(Rs)`	`if ([!]Pt[.new][0]) { EA=Rs+#0; Rd = *EA; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memub(Rs+#u6:0)`	`if ([!]Pt[.new][0]) { EA=Rs+#u; Rd = *EA; } else { NOP; };`

Type: LD (slots 0,1)

Encoding

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	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	0	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (Pt) Rd=memub(Rs+#u6:0)
0	1	0	0	0	0	1	1	0	0	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (Pt.new) Rd=memub(Rs+#u6:0)
0	1	0	0	0	1	0	1	0	0	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (!Pt) Rd=memub(Rs+#u6:0)
0	1	0	0	0	1	1	1	0	0	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (!Pt.new) Rd=memub(Rs+#u6:0)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	0	1	x	x	x	x	x	P	P	1	-	0	t	t	i	i	i	i	d	d	d	d	d	if (Pt) Rd=memub(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	1	x	x	x	x	x	P	P	1	-	1	t	t	i	i	i	i	d	d	d	d	d	if (!Pt) Rd=memub(Rx++#s4:0)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
x5	Field to encode register x

Load unsigned halfword

Load an unsigned halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is zero extended from 16 bits to 32.

Syntax	Behavior
`Rd=memuh(Rs+#s11:1)`	`EA=Rs+#s;` `Rd = *EA;`

`Rd=memuh(Rx++#s4:1)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+#s;`

`Rd=memuh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rd=memuh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,I<<1,MuV);`

`Rd=memuh(Rx++Mu)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memuh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memuh(#u16:1)`	`EA=(GP | #u);` `Rd = *EA;`

Type: LD (slots 0,1)

Encoding

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	
ICLASS								Type		UN						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(#u16:1)
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rs+#s11:1)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++I:circ(Mu))

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	
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++#s4:1)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++Mu)
1	0	0	1	1	1	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u1	Field to encode register u
x5	Field to encode register x

Load unsigned halfword conditionally

Load an unsigned halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is zero extended from 16 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pt) Rd=memuh(Rx++#s4:1)`	`if([!]Pt[0]){ EA=Rx; Rd = *EA; Rx=Rx+#s; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memuh(Rs)`	`if ([!]Pt[.new][0]) { EA=Rs+#0; Rd = *EA; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memuh(Rs+#u6:1)`	`if ([!]Pt[.new][0]) { EA=Rs+#u; Rd = *EA; } else { NOP; };`

Type: LD (slots 0,1)

Encoding

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	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (Pt) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (Pt.new) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (!Pt) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	1	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	i	i	i	i	i	i	d	d	d	d	d	if (!Pt.new) Rd=memuh(Rs+#u6:1)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	1	1	x	x	x	x	x	P	P	1	-	0	t	t	i	i	i	i	d	d	d	d	d	if (Pt) Rd=memuh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	1	x	x	x	x	x	P	P	1	-	1	t	t	i	i	i	i	d	d	d	d	d	if (!Pt) Rd=memuh(Rx++#s4:1)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
x5	Field to encode register x

Load word

Load a 32-bit word from memory and place in a destination register.

Syntax	Behavior
`Rd=memw(Rs)`	`EA=Rs+#0;` `Rd = *EA;`
`Rd=memw(Rs+#s11:2)`	`EA=Rs+#s;` `Rd = *EA;`

`Rd=memw(Rx++#s4:2)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+#s;`

`Rd=memw(Rx++#s4:2:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rd=memw(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=circ_add(Rx,I<<2,MuV);`

`Rd=memw(Rx++Mu)`	`EA=Rx;`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memw(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rd = *EA;`
	`Rx=Rx+MuV;`

`Rd=memw(#u16:2)`	`EA=(GP | #u);` `Rd = *EA;`

Type: LD (slots 0,1)

Encoding

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	
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	**i**	**i**	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(#u16:2)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rs+#s11:2)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					
1	0	0	1	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++I:circ(Mu))

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	
ICLASS				Amode			Type			UN	x5					Parse												d5					
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++#s4:2)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	1	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++Mu)
1	0	0	1	1	1	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u1	Field to encode register u
x5	Field to encode register x

Load word conditionally

Load a 32-bit word from memory and place in a destination register.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pt) Rd=memw(Rx++#s4:2)`	`if([!]Pt[0]){ EA=Rx; Rd = *EA; Rx=Rx+#s; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memw(Rs)`	`if ([!]Pt[.new][0]) { EA=Rs+#0; Rd = *EA; } else { NOP; };`
`if ([!]Pt[.new]) Rd=memw(Rs+#u6:2)`	`if ([!]Pt[.new][0]) { EA=Rs+#u; Rd = *EA; } else { NOP; };`

Type: LD (slots 0,1)

Encoding

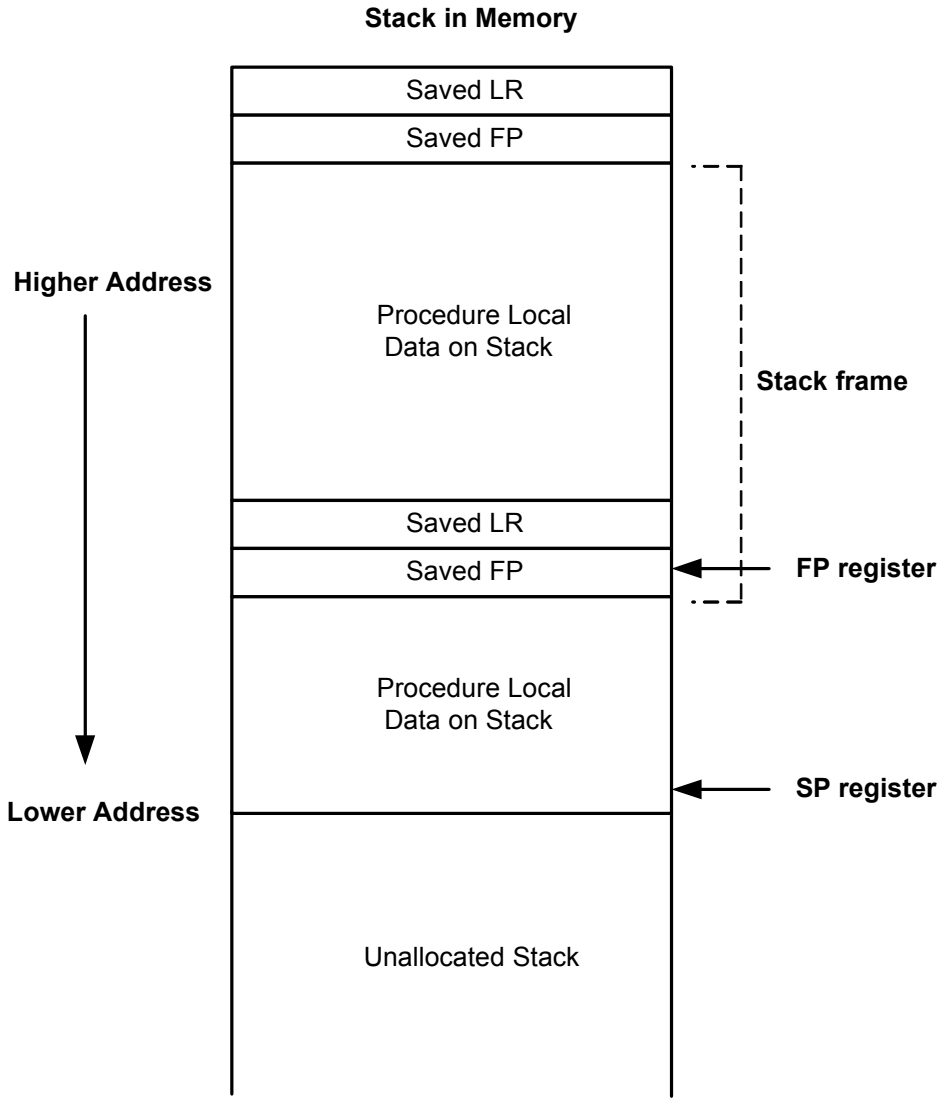
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(Rs+#u6:2)
0	1	0	0	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memw(Rs+#u6:2)
0	1	0	0	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(Rs+#u6:2)
0	1	0	0	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memw(Rs+#u6:2)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	-	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(Rx++#s4:2)
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	-	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(Rx++#s4:2)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
x5	Field to encode register x

Deallocate stack frame

Deallocate a stack frame from the call stack. The instruction first loads the saved FP and saved LR values from the address at FP. It then points SP back to the previous frame.

The stack layout is shown in the following figure.



Syntax	Behavior

`deallocframe`	`EA=FP;`
	`tmp = *EA;`
	`LR=tmp.w[1];`
	`FP=tmp.w[0];`
	`SP=EA+8;`

Type: LD (slots 0,1)

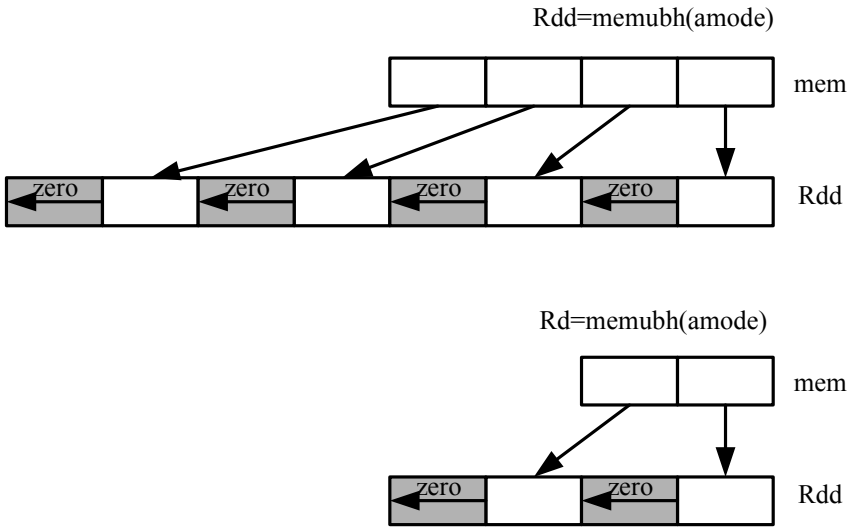
Encoding

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	
ICLASS				Amode			Type			UN						Parse																	
1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	0	**P**	**P**	0	-	-	-	-	-	-	-	-	1	1	1	1	0	deallocframe

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits

Load and unpack bytes to halfwords

Load contiguous bytes from memory and vector unpack them into halfwords.



Syntax	Behavior

`Rd=memubh(Rs)`	`EA=Rs+#0;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rd=memubh(Rs+#s11:1)`	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rd=memubh(Rx++#s4:1)`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=Rx+#s;`

`Rd=memubh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rd=memubh(Rx++I:circ(Mu))`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=circ_add(Rx,I<<1,MuV);`

Syntax	Behavior

`Rd=memubh(Rx++Mu)`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=Rx+MuV;`

`Rd=memubh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=Rx+MuV;`

`Rdd=memubh(Rs)`	`EA=Rs+#0;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rdd=memubh(Rs+#s11:2)`	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

Syntax	Behavior

`Rdd=memubh(Rx++#s4:2)`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=Rx+#s;`

`Rdd=memubh(Rx++#s4:2:circ(Mu))`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=circ_add(Rx,#s,MuV);`

`Rdd=memubh(Rx++I:circ(Mu))`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=circ_add(Rx,I<<2,MuV);`

`Rdd=memubh(Rx++Mu)`	`EA=Rx;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=Rx+MuV;`

`Rdd=memubh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Rx=Rx+MuV;`

Type: LD (slots 0,1)

Encoding

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	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rs+#s11:1)
1	0	0	1	0	**i**	**i**	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rs+#s11:2)
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	0	0	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++I:circ(Mu))
1	0	0	1	1	0	0	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++#s4:1)
1	0	0	1	1	0	1	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++#s4:2)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++Mu)
1	0	0	1	1	1	0	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++Mu)
1	0	0	1	1	1	1	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++Mu:brev)
1	0	0	1	1	1	1	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u1	Field to encode register u
x5	Field to encode register x

11.8 ST

The ST instruction class includes the store instructions, which are used to store register values into memory.

Store doubleword

Store a 64-bit register pair in memory at the effective address.

Syntax	Behavior
`memd(Rs)=Rtt`	`EA=Rs+#0;` `*EA = Rtt;`
`memd(Rs+#s11:3)=Rtt`	`EA=Rs+#s;` `*EA = Rtt;`

`memd(Rx++#s4:3)=Rtt`	`EA=Rx;`
	`*EA = Rtt;`
	`Rx=Rx+#s;`

`memd(Rx++#s4:3:circ(Mu))=Rtt`	`EA=Rx;`
	`*EA = Rtt;`
	`Rx=circ_add(Rx,#s,MuV);`

`memd(Rx++I:circ(Mu))=Rtt`	`EA=Rx;`
	`*EA = Rtt;`
	`Rx=circ_add(Rx,I<<3,MuV);`

`memd(Rx++Mu)=Rtt`	`EA=Rx;`
	`*EA = Rtt;`
	`Rx=Rx+MuV;`

`memd(Rx++Mu:brev)=Rtt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`*EA = Rtt;`
	`Rx=Rx+MuV;`

`memd(#u16:3)=Rtt`	`EA=GP | #u;` `*EA = Rtt;`

Type: ST (slot 0)

Encoding

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	
ICLASS								Type								Parse			t5														
0	1	0	0	1	i	i	0	1	1	0	i	i	i	i	i	P	P	i	t	t	t	t	t	i	i	i	i	i	i	i	i	memd(#u16:3)=Rtt
ICLASS				Amode			Type			UN	s5					Parse			t5													
1	0	1	0	0	i	i	1	1	1	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	i	i	i	memd(Rs+#s11:3)=Rtt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	-	-	-	-	-	1	-	memd(Rx++I:circ(Mu))=Rtt
1	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	i	i	i	i	-	0	-	memd(Rx++#s4:3:circ(Mu))=Rtt

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	
ICLASS				Amode			Type			U N	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	**-**	**-**	memd(Rx++#s4:3)=Rtt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memd(Rx++Mu)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memd(Rx++Mu:brev)=Rtt

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store doubleword conditionally

Store a 64-bit register pair in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pv) memd(Rs)=Rtt`	`if ([!]Pv[0]){ EA=Rs+#0; *EA = Rtt; } else { NOP; };`
`if ([!]Pv) memd(Rs+#u6:3)=Rtt`	`if ([!]Pv[0]){ EA=Rs+#u; *EA = Rtt; } else { NOP; };`
`if ([!]Pv) memd(Rx++#s4:3)=Rtt`	`if ([!]Pv[0]){ EA=Rx; *EA = Rtt; Rx=Rx+#s; } else { NOP; };`

Type: ST (slot 0)

Encoding

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	
ICLASS					Sense	PredNew		Type			s5					Parse			t5														
0	1	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (Pv) memd(Rs+#u6:3)=Rtt
0	1	0	0	0	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (!Pv) memd(Rs+#u6:3)=Rtt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memd(Rx++#s4:3)=Rtt
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memd(Rx++#s4:3)=Rtt

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store byte

Store the least-significant byte in a source register at the effective address.

Syntax	Behavior
`memb(Rs)=Rt`	`EA=Rs+#0;` `*EA = Rt&0xff;`
`memb(Rs+#s11:0)=Rt`	`EA=Rs+#s;` `*EA = Rt&0xff;`

`memb(Rx++#s4:0)=Rt`	`EA=Rx;`
	`*EA = Rt&0xff;`
	`Rx=Rx+#s;`

`memb(Rx++#s4:0:circ(Mu))=Rt`	`EA=Rx;`
	`*EA = Rt&0xff;`
	`Rx=circ_add(Rx,#s,MuV);`

`memb(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`*EA = Rt&0xff;`
	`Rx=circ_add(Rx,I<<0,MuV);`

`memb(Rx++Mu)=Rt`	`EA=Rx;`
	`*EA = Rt&0xff;`
	`Rx=Rx+MuV;`

`memb(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`*EA = Rt&0xff;`
	`Rx=Rx+MuV;`

`memb(#u16:0)=Rt`	`EA=GP | #u;` `*EA = Rt&0xff;`

Type: ST (slot 0)

Encoding

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	
ICLASS								Type								Parse			t5														
0	1	0	0	1	i	i	0	0	0	0	i	i	i	i	i	P	P	i	t	t	t	t	t	i	i	i	i	i	i	i	i	memb(#u16:0)=Rt
ICLASS				Amode			Type			U N	s5					Parse			t5													
1	0	1	0	0	i	i	1	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	i	i	i	memb(Rs+#s11:0)=Rt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	-	-	-	-	-	1	-	memb(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	i	i	i	i	-	0	-	memb(Rx++#s4:0:circ(Mu))=Rt

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	
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	-	-	-	memb(Rx++#s4:0)=Rt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	-	-	-	-	-	-	-	memb(Rx++Mu)=Rt
1	0	1	0	1	1	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	-	-	-	-	-	-	-	memb(Rx++Mu:brev)=Rt

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store byte conditionally

Store the least-significant byte in a source register at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pv) memb(Rs)=Rt`	`if ([!]Pv[0]){ EA=Rs+#0; *EA = Rt&0xff; } else { NOP; };`
`if ([!]Pv) memb(Rs+#u6:0)=Rt`	`if ([!]Pv[0]){ EA=Rs+#u; *EA = Rt&0xff; } else { NOP; };`
`if ([!]Pv) memb(Rx++#s4:0)=Rt`	`if ([!]Pv[0]){ EA=Rx; *EA = Rt&0xff; Rx=Rx+#s; } else { NOP; };`

Type: ST (slot 0)

Encoding

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	
ICLASS					Sense	Pred New		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (Pv) memb(Rs+#u6:0)=Rt
0	1	0	0	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (!Pv) memb(Rs+#u6:0)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rx++#s4:0)=Rt
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(Rx++#s4:0)=Rt

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store halfword

Store the upper or lower 16-bits of a source register at the effective address.

Syntax	Behavior
`memh(Rs)=Rt.H`	`EA=Rs+#0;` `*EA = Rt.h[1];`
`memh(Rs)=Rt`	`EA=Rs+#0;` `*EA = Rt&0xffff;`
`memh(Rs+#s11:1)=Rt.H`	`EA=Rs+#s;` `*EA = Rt.h[1];`
`memh(Rs+#s11:1)=Rt`	`EA=Rs+#s;` `*EA = Rt&0xffff;`

`memh(Rx++#s4:1)=Rt.H`	`EA=Rx;`
	`*EA = Rt.h[1];`
	`Rx=Rx+#s;`

`memh(Rx++#s4:1)=Rt`	`EA=Rx;`
	`*EA = Rt&0xffff;`
	`Rx=Rx+#s;`

`memh(Rx++#s4:1:circ(Mu))=Rt.H`	`EA=Rx;`
	`*EA = Rt.h[1];`
	`Rx=circ_add(Rx,#s,MuV);`

`memh(Rx++#s4:1:circ(Mu))=Rt`	`EA=Rx;`
	`*EA = Rt&0xffff;`
	`Rx=circ_add(Rx,#s,MuV);`

`memh(Rx++I:circ(Mu))=Rt.H`	`EA=Rx;`
	`*EA = Rt.h[1];`
	`Rx=circ_add(Rx,I<<1,MuV);`

`memh(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`*EA = Rt&0xffff;`
	`Rx=circ_add(Rx,I<<1,MuV);`

`memh(Rx++Mu)=Rt.H`	`EA=Rx;`
	`*EA = Rt.h[1];`
	`Rx=Rx+MuV;`

Syntax	Behavior

`memh(Rx++Mu)=Rt`	`EA=Rx;`
	`*EA = Rt&0xffff;`
	`Rx=Rx+MuV;`

`memh(Rx++Mu:brev)=Rt.H`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`*EA = Rt.h[1];`
	`Rx=Rx+MuV;`

`memh(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`*EA = Rt&0xffff;`
	`Rx=Rx+MuV;`

`memh(#u16:1)=Rt.H`	`EA=GP | #u;` `*EA = Rt.h[1];`
`memh(#u16:1)=Rt`	`EA=GP | #u;` `*EA = Rt&0xffff;`

Type: ST (slot 0)

Encoding

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	
ICLASS								Type								Parse			t5														
0	1	0	0	1	**i**	**i**	0	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(#u16:1)=Rt
0	1	0	0	1	**i**	**i**	0	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(#u16:1)=Rt.H
ICLASS				Amode			Type			UN	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(Rs+#s11:1)=Rt
1	0	1	0	0	**i**	**i**	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(Rs+#s11:1)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memh(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memh(Rx++I:circ(Mu))=Rt.H
1	0	1	0	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1:circ(Mu))=Rt.H
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	**-**	**-**	memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	**-**	**-**	memh(Rx++#s4:1)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu)=Rt
1	0	1	0	1	1	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu)=Rt.H
1	0	1	0	1	1	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu:brev)=Rt
1	0	1	0	1	1	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu:brev)=Rt.H

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store halfword conditionally

Store the upper or lower 16-bits of a source register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv) memh(Rs)=Rt.H`	`if ([!]Pv[0]){`
	`  EA=Rs+#0;`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv) memh(Rs)=Rt`	`if ([!]Pv[0]){`
	`  EA=Rs+#0;`
	`  *EA = Rt&0xffff;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv)`
`memh(Rs+#u6:1)=Rt.H`
	`if ([!]Pv[0]){`
	`  EA=Rs+#u;`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv)`
`memh(Rs+#u6:1)=Rt`
	`if ([!]Pv[0]){`
	`  EA=Rs+#u;`
	`  *EA = Rt&0xffff;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv)`
`memh(Rx++#s4:1)=Rt.H`
	`if ([!]Pv[0]){`
	`  EA=Rx;`
	`  *EA = Rt.h[1];`
	`  Rx=Rx+#s;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv)`
`memh(Rx++#s4:1)=Rt`
	`if ([!]Pv[0]){`
	`  EA=Rx;`
	`  *EA = Rt&0xffff;`
	`  Rx=Rx+#s;`
	`} else {`
	`  NOP;`
	`};`

Type: ST (slot 0)

Encoding

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	
ICLASS					Sense	PredNew		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (Pv) memh(Rs+#u6:1)=Rt
0	1	0	0	0	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (Pv) memh(Rs+#u6:1)=Rt.H
0	1	0	0	0	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Rt
0	1	0	0	0	1	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Rt.H
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Rt.H

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store word

Store a 32-bit register in memory at the effective address.

Syntax	Behavior
`memw(Rs)=Rt`	`EA=Rs+#0;` `*EA = Rt;`
`memw(Rs+#s11:2)=Rt`	`EA=Rs+#s;` `*EA = Rt;`

`memw(Rx++#s4:2)=Rt`	`EA=Rx;`
	`*EA = Rt;`
	`Rx=Rx+#s;`

`memw(Rx++#s4:2:circ(Mu))=Rt`	`EA=Rx;`
	`*EA = Rt;`
	`Rx=circ_add(Rx,#s,MuV);`

`memw(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`*EA = Rt;`
	`Rx=circ_add(Rx,I<<2,MuV);`

`memw(Rx++Mu)=Rt`	`EA=Rx;`
	`*EA = Rt;`
	`Rx=Rx+MuV;`

`memw(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`*EA = Rt;`
	`Rx=Rx+MuV;`

`memw(#u16:2)=Rt`	`EA=GP | #u;` `*EA = Rt;`

Type: ST (slot 0)

Encoding

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	
ICLASS								Type								Parse			t5														
0	1	0	0	1	**i**	**i**	0	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(#u16:2)=Rt
ICLASS				Amode			Type			U N	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(Rs+#s11:2)=Rt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	-	-	-	-	-	1	-	memw(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	-	0	-	memw(Rx++#s4:2:circ(Mu))=Rt

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	
ICLASS				Amode			Type			U N	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	-	-	-	memw(Rx++#s4:2)=Rt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	-	-	-	-	-	-	-	memw(Rx++Mu)=Rt
1	0	1	0	1	1	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	-	-	-	-	-	-	-	memw(Rx++Mu:brev)=Rt

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store word conditionally

Store a 32-bit register is stored in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv) memw(Rs)=Rt`	`if ([!]Pv[0]){`
	`  EA=Rs+#0;`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv) memw(Rs+#u6:2)=Rt`	`if ([!]Pv[0]){`
	`  EA=Rs+#u;`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv) memw(Rx++#s4:2)=Rt`	`if ([!]Pv[0]){`
	`  EA=Rx;`
	`  *EA = Rt;`
	`  Rx=Rx+#s;`
	`} else {`
	`  NOP;`
	`};`

Type: ST (slot 0)

Encoding

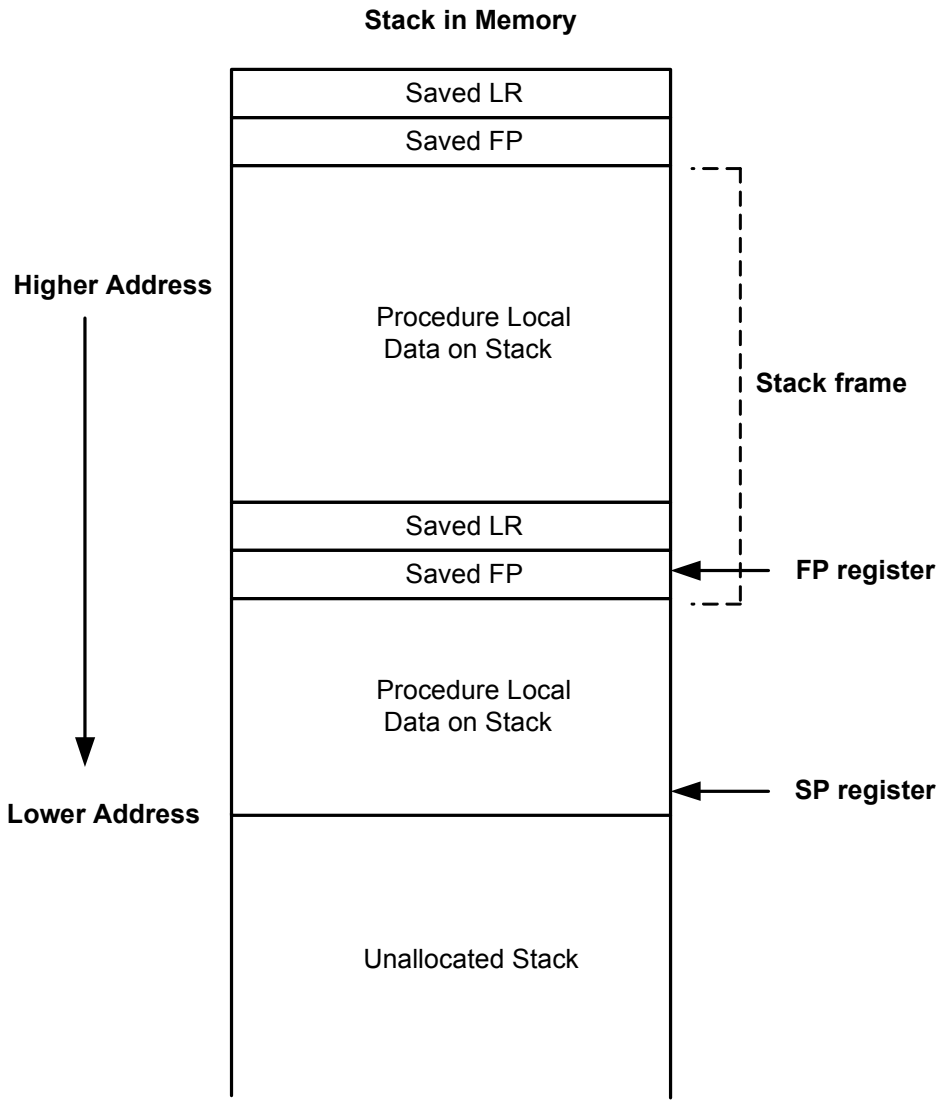
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (Pv) memw(Rs+#u6:2)=Rt
0	1	0	0	0	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**-**	**v**	**v**	if (!Pv) memw(Rs+#u6:2)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(Rx++#s4:2)=Rt
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**-**	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memw(Rx++#s4:2)=Rt

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Allocate stack frame

Allocate a stack frame on the call stack. This instruction first pushes LR and FP to the top of stack. It then subtracts an unsigned immediate from SP in order to allocate room for local variables. FP is set to the address of the old frame pointer on the stack.

The stack layout is shown in the following figure.



Syntax	Behavior

`allocframe(#u11:3)`	`EA=SP-8;`
	`*EA = ((LR << 32) | FP);`
	`FP=EA;`
	`SP=EA-#u;`

Type: ST (slot 0)

Encoding

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	
ICLASS				Amode			Type			U N						Parse																	
1	0	1	0	0	0	0	0	1	0	0	1	1	1	0	1	**P**	**P**	0	0	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	allocframe(#u11:3)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
Amode	Amode
Type	Type
UN	Unsigned

11.9 SYSTEM

The SYSTEM instruction class includes instructions which enable access to system-level processor resources.

11.9.1 System / User

The SYSTEM user instructions enable access to system-level processor resources.

Load locked word

Perform atomic memory operation.

Syntax	Behavior

`Rd=memw_locked(Rs)`	`EA=Rs;`
	`Rd = *EA;`
	`;`

Type: LD (slots 0,1)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=memw_locked(Rs)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Store conditional word

Perform atomic memory operation.

Syntax	Behavior

`memw_locked(Rs,Pd)=Rt`	`EA=Rs;`
	`if (lock_valid) {`
	`  *EA = Rt;`
	`  Pd = 0xff;`
	`  lock_valid = 0;`
	`} else {`
	`  Pd = 0;`
	`};`
	`;`

Type: ST (slot 0)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS				Amode			Type			UN	s5					Parse			t5											d2		
1	0	1	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	memw_locked(Rs,Pd)=Rt

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Amode	Amode
Type	Type
UN	Unsigned

Zero a cache line

For write-back cacheable pages, first look up the data cache at address Rs:

- If the line is in the cache, the line is cleared (written with all zeros).
- If the line is not in the cache, it is allocated and then cleared.
- In the case of a miss, the line is allocated and cleared directly in the cache without first fetching the line as is the normal case for a write-back allocate-on-miss store.

This instruction is useful for efficiently handling write-only data by pre-allocating lines in the cache.

The address should be 32-byte aligned. If not, an unaligned error exception will be raised.

This instruction is defined only for write-back cacheable pages only. For write-through pages and uncached pages, it raises an error exception.

Syntax	Behavior
`dczeroa(Rs)`	`EA=Rs;` `dcache_zero_addr(EA);`

Type: ST (slot 0)

Notes

- A packet containing this instruction must have slot 1 either empty or executing a NOP.

Encoding

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	
ICLASS				Amode			Type			UN	s5					Parse																	
1	0	1	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dczeroa(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
Amode	Amode
Type	Type
UN	Unsigned

Memory barrier

Establish a memory barrier to ensure proper ordering between accesses before the barrier instruction and accesses after the barrier instruction. All accesses before the barrier will be globally observable before any access after the barrier can be observed.

The use of this instruction is MSM-system-dependent.

Syntax	Behavior
`barrier`	`memory_barrier;`

Type: ST (slot 0)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS				Amode			Type			UN						Parse																	
1	0	1	0	1	0	0	0	0	0	0	-	-	-	-	-	P	P	-	-	-	-	-	-	-	-	-	-	-	-	-	-	barrier

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
Amode	Amode
Type	Type
UN	Unsigned

Breakpoint

Cause the program to enter debug mode if enabled by ISDB. Execution control is handed to ISDB and the program will not proceed until directed by the debugger.

If ISDB is disabled, this instruction is treated as a NOP.

Syntax	Behavior
`brkpt`	`Enter debug mode;`

Type: CR (slot 3)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS				rs	sm											Parse																
0	1	1	0	1	1	0	0	0	0	1	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	0	0	0	-	-	-	-	-	brkpt

Field name	Description
`rs`	Read register Rs
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Data cache prefetch

Prefetch the data at address Rs.

This instruction is a hint to the memory system, and is handled in an implementation-dependent manner.

Syntax	Behavior
`dcfetch(Rs)`	`dcache_fetch(Rs);`

Type: LD (slots 0)

Intrinsics

`dcfetch(Rs)`	`void Q6_dcfetch_A(Address a)`

Encoding

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	
ICLASS				Amode			Type			UN	s5					Parse																
1	0	0	1	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	-	-	dcfetch(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Amode`	Amode
`Type`	Type
`UN`	Unsigned
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s

Data cache maintenance user operations

Perform maintenance operations on the data cache.

`dccleaninva` looks up the data cache at address Rs. If this address is in the cache and has dirty data, then the data is flushed out to memory and the line is then invalidated.

`dccleana` looks up the data cache at address Rs. If this address is in the cache and has dirty data, then the data is flushed out to memory.

`dcinva` looks up the data cache at address Rs. If this address is in the cache, then the line containing the data is invalidated.

Syntax	Behavior
`dccleana(Rs)`	`EA=Rs;` `dcache_clean_addr(EA);`

`dccleaninva(Rs)`	`EA=Rs;`
	`dcache_clean_addr(EA);`
	`dcache_inv_addr(EA);`

`dcinva(Rs)`	`EA=Rs;` `dcache_inv_addr(EA);`

Type: ST (slot 0)

Notes

- A packet containing this instruction must have slot 1 either empty or executing a NOP.

Encoding

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	
ICLASS				Amode			Type			UN	s5					Parse																	
1	0	1	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dccleana(Rs)
1	0	1	0	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dcinva(Rs)
1	0	1	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dccleaninva(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s
`Amode`	Amode
`Type`	Type
`UN`	Unsigned

Instruction cache maintenance user operations

Look up the address in Rs in the instruction cache. If the address is found, invalidate the corresponding cache line.

Syntax	Behavior
`icinva(Rs)`	`EA=Rs;` `icache_inv_addr(EA);`

Type: JR (slot 2)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS											s5					Parse																
0	1	0	1	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	0	-	-	-	-	-	-	-	-	-	-	-	icinva(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s

Instruction synchronization

Ensure that all prior instructions have committed before continuing to the next instruction.

This instruction should be executed after the following events (when it is desired for subsequent instructions to observe the results of the event):

- After modifying the TLB with a TLBW instruction
- After modifying the SSR register
- After modifying the SYSCFG register
- After any instruction cache maintenance operation
- After modifying the TID register

Syntax	Behavior
`isync`	`memory_synch;`

Type: JR (slots 2)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS																Parse																
0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	**P**	**P**	0	-	-	-	0	0	0	0	0	0	0	0	1	0	isync

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits

Memory thread synchronization

Synchronize memory.

All outstanding memory operations, including cached and uncached loads and stores, are completed before the processor continues to the next instruction. This ensures that certain memory operations are performed in the desired order (for example, when accessing I/O devices).

After performing a `syncht` operation, the processor ceases fetching and executing instructions from the program until all outstanding memory operations of that program are completed.

In multithreaded or multicore environments, `syncht` is not concerned with other execution contexts.

The use of this instruction is MSM-system-dependent.

Syntax	Behavior
`syncht`	`memory_synch;`

Type: ST (slot 0)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS				Amode			Type			U N						Parse																	
1	0	1	0	1	0	0	0	0	1	0	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	syncht

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`Amode`	Amode
`Type`	Type
`UN`	Unsigned

Trap

Cause a precise exception.

Executing a trap instruction sets the EX bit in SSR to 1, which disables interrupts and enables supervisor mode. The program then jumps to the vector location (either `trap0` location or `trap1` location). The instruction specifies an 8-bit immediate field. This field is copied into the system status register cause field.

Upon returning from the service routine with a RTE, execution resumes at the packet after the TRAP instruction.

These instructions are generally intended for user code to switch to request services from the operating system. Two trap instructions are provided so the OS can optimize for fast service routines and slower service routines.

Syntax	Behavior
`trap0(#u8)`	`SSR.CAUSE = #u;` `TRAP "0";`
`trap1(#u8)`	`SSR.CAUSE = #u;` `TRAP "1";`

Type: JR (slots 2)

Notes

- This is a solo instruction. It cannot be grouped with other instructions in a packet.

Encoding

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	
ICLASS																Parse																
0	1	0	1	0	1	0	0	0	-	-	-	-	-	-	-	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	**i**	-	-	trap0(#u8)
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	-	-	-	**i**	**i**	**i**	-	-	trap1(#u8)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

11.10 XTYPE

The XTYPE instruction class includes a variety of instruction types, most of which operate on 64-bit values.

11.10.1 XTYPE / Arithmetic & Logical

The XTYPE arithmetic and logical instructions operate on 32- and 64-bit data.

Absolute value

Take the absolute value of the source register and place in destination register. The 32-bit version of the operation is available with optional saturation. The single case of saturation is if the source register is equal to 0x80000000, then the destination saturates to 0x7fffffff.

Syntax	Behavior
`Rd=abs(Rs)[:sat]`	`Rd = [sat_32](ABS(sxt`$_{32->64}$`(Rs)));`
`Rdd=abs(Rss)`	`Rdd = ABS(Rss);`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=abs(Rs)`	`Word32 Q6_R_abs_R(Word32 Rs)`
`Rd=abs(Rs):sat`	`Word32 Q6_R_abs_R_sat(Word32 Rs)`
`Rdd=abs(Rss)`	`Word64 Q6_P_abs_P(Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=abs(Rss)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=abs(Rs)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=abs(Rs):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Add and accumulate

Add Rs and Rt or a signed immediate, then add or subtract the resulting value with Rx. The result is saved in Rx.

Syntax	Behavior
`Rx+=add(Rs,#s8)`	`Rx=Rx + Rs + #s;`
`Rx+=add(Rs,Rt)`	`Rx=Rx + Rs + Rt;`
`Rx-=add(Rs,#s8)`	`Rx=Rx - (Rs + #s);`
`Rx-=add(Rs,Rt)`	`Rx=Rx - (Rs + Rt);`

Type: M (slots 2,3)

Intrinsics

`Rx+=add(Rs,#s8)`	`Word32 Q6_R_addacc_RI(Word32 Rx, Word32 Rs, Word32 Is)`
`Rx+=add(Rs,Rt)`	`Word32 Q6_R_addacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=add(Rs,#s8)`	`Word32 Q6_R_addnac_RI(Word32 Rx, Word32 Rs, Word32 Is)`
`Rx-=add(Rs,Rt)`	`Word32 Q6_R_addnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			x5					
1	1	1	0	0	0	1	0	0	-	-	s	s	s	s	s	P	P	-	i	i	i	i	i	i	i	i	x	x	x	x	x	Rx+=add(Rs,#s8)
1	1	1	0	0	0	1	0	1	-	-	s	s	s	s	s	P	P	-	i	i	i	i	i	i	i	i	x	x	x	x	x	Rx-=add(Rs,#s8)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	0	1	x	x	x	x	x	Rx+=add(Rs,Rt)
1	1	1	0	1	1	1	1	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	0	1	x	x	x	x	x	Rx-=add(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits

Field name	Description
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Add

The first form of this instruction adds two 32-bit registers. If the result overflows 32 bits, it is saturated to 0x7fffffff for a positive result, or 0x80000000 for a negative result. Note that 32-bit non-saturating register add is an ALU32 instruction and can be executed in any slot.

The second form adds 64-bit registers Rss and Rtt and puts the result in Rdd.

Syntax	Behavior
`Rd=add(Rs,Rt):sat`	`Rd=sat_32(Rs+Rt);`
`Rdd=add(Rss,Rtt)`	`Rdd=Rss+Rtt;`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=add(Rs,Rt):sat`	`Word32 Q6_R_add_RR_sat(Word32 Rs, Word32 Rt)`
`Rdd=add(Rss,Rtt)`	`Word64 Q6_P_add_PP(Word64 Rss, Word64 Rtt)`

Encoding

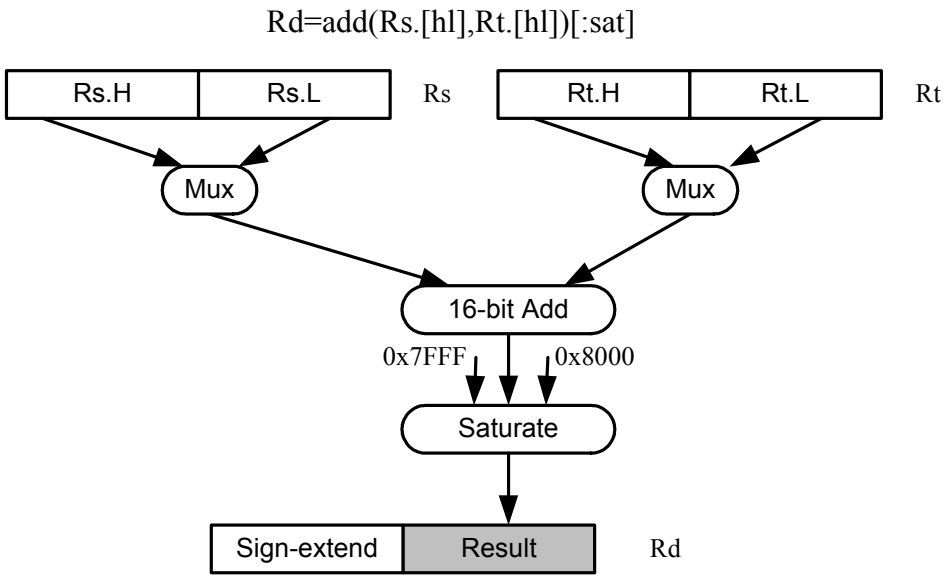
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	1	d	d	d	d	d	Rdd=add(Rss,Rtt)
1	1	0	1	-	1	0	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	-	-	d	d	d	d	d	Rd=add(Rs,Rt):sat

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Add halfword

Perform a 16-bit add with optional saturation, and place the result in either the upper or lower half of a register. If the result goes in the upper half, then the sources can be any high or low halfword of Rs and Rt. The lower 16 bits of the result are zeroed.

If the result is to be placed in the lower 16 bits of Rd, then the Rs source can be either high or low, but the other source must be the low halfword of Rt. In this case, the upper halfword of Rd is the sign-extension of the low halfword.



Syntax	Behavior
`Rd=add(Rt.L,Rs.[HL])[:sat]`	`Rd=[sat_16](Rt.h[0]+Rs.h[01]);`
`Rd=add(Rt.[HL],Rs.[HL])[:sat]:<<16`	`Rd=([sat_16](Rt.h[01]+Rs.h[01]))<<16;`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

Rd=add(Rt.H,Rs.H):<<16	Word32 Q6_R_add_RhRh_s16(Word32 Rt, Word32 Rs)
Rd=add(Rt.H,Rs.H):sat:<<16	Word32 Q6_R_add_RhRh_sat_s16(Word32 Rt, Word32 Rs)
Rd=add(Rt.H,Rs.L):<<16	Word32 Q6_R_add_RhRl_s16(Word32 Rt, Word32 Rs)
Rd=add(Rt.H,Rs.L):sat:<<16	Word32 Q6_R_add_RhRl_sat_s16(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.H)	Word32 Q6_R_add_RlRh(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.H):<<16	Word32 Q6_R_add_RlRh_s16(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.H):sat	Word32 Q6_R_add_RlRh_sat(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.H):sat:<<16	Word32 Q6_R_add_RlRh_sat_s16(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.L)	Word32 Q6_R_add_RlRl(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.L):<<16	Word32 Q6_R_add_RlRl_s16(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.L):sat	Word32 Q6_R_add_RlRl_sat(Word32 Rt, Word32 Rs)
Rd=add(Rt.L,Rs.L):sat:<<16	Word32 Q6_R_add_RlRl_sat_s16(Word32 Rt, Word32 Rs)

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L)
1	1	0	1	-	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H)
1	1	0	1	-	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):sat
1	1	0	1	-	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):sat
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):<<16
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):<<16
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.L):<<16
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.H):<<16
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):sat:<<16
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):sat:<<16
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.L):sat:<<16
1	1	0	1	-	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.H):sat:<<16

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Compare

Compare two 64-bit register pairs for unsigned greater than, greater than, or equal. The 8-bit predicate register Pd is set to all 1's or all 0's, depending on the result.

Syntax	Behavior
`Pd=cmp.eq(Rss,Rtt)`	`Pd=(Rss==Rtt) ? 0xff : 0x00;`
`Pd=cmp.gt(Rss,Rtt)`	`Pd=(Rss>Rtt) ? 0xff : 0x00;`
`Pd=cmp.gtu(Rss,Rtt)`	`Pd=(Rss.u64>Rtt.u64) ? 0xff : 0x00;`

Type: ALU64 (slots 2,3)

Intrinsics

`Pd=cmp.eq(Rss,Rtt)`	`Byte Q6_p_cmp_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=cmp.gt(Rss,Rtt)`	`Byte Q6_p_cmp_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=cmp.gtu(Rss,Rtt)`	`Byte Q6_p_cmp_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp						d2		
1	1	0	1	-	0	1	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	-	-	-	**d**	**d**	Pd=cmp.eq(Rss,Rtt)
1	1	0	1	-	0	1	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	-	-	-	**d**	**d**	Pd=cmp.gt(Rss,Rtt)
1	1	0	1	-	0	1	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	-	-	-	**d**	**d**	Pd=cmp.gtu(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Logical operations

Perform bitwise logical AND, OR, XOR, and NOT operations.

The source and destination registers are 64-bit.

For 32-bit logical operations, see the ALU32 logical instructions.

Syntax	Behavior
`Rdd=and(Rss,Rtt)`	`Rdd=Rss&Rtt;`
`Rdd=or(Rss,Rtt)`	`Rdd=Rss|Rtt;`
`Rdd=xor(Rss,Rtt)`	`Rdd=Rss^Rtt;`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=and(Rss,Rtt)`	`Word64 Q6_P_and_PP(Word64 Rss, Word64 Rtt)`
`Rdd=or(Rss,Rtt)`	`Word64 Q6_P_or_PP(Word64 Rss, Word64 Rtt)`
`Rdd=xor(Rss,Rtt)`	`Word64 Q6_P_xor_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5						MinOp			d5					
1	1	0	1	-	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=and(Rss,Rtt)
1	1	0	1	-	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=or(Rss,Rtt)
1	1	0	1	-	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=xor(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Maximum word

Select the maximum of two 32-bit source registers, and place in destination register Rd.

Syntax	Behavior
`Rd=max(Rs,Rt)`	`Rd = max(Rs,Rt);`
`Rd=maxu(Rs,Rt)`	`Rd = max(Rs.uw[0],Rt.uw[0]);`

Type: ALU64 (slots 2,3)

Intrinsics

`Rd=max(Rs,Rt)`	`Word32 Q6_R_max_RR(Word32 Rs, Word32 Rt)`
`Rd=maxu(Rs,Rt)`	`Word32 Q6_R_maxu_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=max(Rs,Rt)
1	1	0	1	-	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=maxu(Rs,Rt)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Minimum word

Select the signed or unsigned minimum of two 32-bit source registers, and place in destination register Rd.

Syntax	Behavior
`Rd=min(Rt,Rs)`	`Rd = min(Rt,Rs);`
`Rd=minu(Rt,Rs)`	`Rd = min(Rt.uw[0],Rs.uw[0]);`

Type: ALU64 (slots 2,3)

Intrinsics

`Rd=min(Rt,Rs)`	`Word32 Q6_R_min_RR(Word32 Rt, Word32 Rs)`
`Rd=minu(Rt,Rs)`	`Word32 Q6_R_minu_RR(Word32 Rt, Word32 Rs)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=min(Rt,Rs)
1	1	0	1	-	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=minu(Rt,Rs)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Negate

The first form of this instruction performs a negate on a 32-bit register with saturation. If the input is 0x80000000, the result is saturated to 0x7fffffff. Note that the non-saturating 32-bit register negate is a ALU32-class instruction and can be executed on any slot.

The second form of this instruction negates a 64-bit source register and puts the result in destination Rdd.

Syntax	Behavior
`Rd=neg(Rs):sat`	`Rd = sat_32(-Rs.s64);`
`Rdd=neg(Rss)`	`Rdd = -Rss;`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=neg(Rs):sat`	`Word32 Q6_R_neg_R_sat(Word32 Rs)`
`Rdd=neg(Rss)`	`Word64 Q6_P_neg_P(Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	-	s	s	s	s	s	P	P	-	-	-	-	-	-	1	0	1	d	d	d	d	d	Rdd=neg(Rss)
1	0	0	0	1	1	0	0	1	0	-	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	0	d	d	d	d	d	Rd=neg(Rs):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Logical NOT

Perform bitwise logical AND, OR, XOR, and NOT operations.

Source and destination registers are 64-bits.

For 32-bit logical operations, see the ALU32 logical instructions.

Syntax	Behavior
`Rdd=not(Rss)`	`Rdd=~Rss;`

Type: S (slots 2,3)

Intrinsics

`Rdd=not(Rss)`	`Word64 Q6_P_not_P(Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=not(Rss)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Subtract

The first form of this instruction subtracts the 32-bit register Rs from the 32-bit register Rt. If the result overflows 32 bits, it is saturated to 0x7fffffff for a positive result, or 0x80000000 for a negative result. Note that the non-saturating 32-bit register subtract is an ALU32 instruction and can be executed on any slot.

The second instruction form subtracts 64-bit register Rtt from register Rss.

Syntax	Behavior
`Rd=sub(Rt,Rs):sat`	`Rd=sat_32(Rt - Rs);`
`Rdd=sub(Rtt,Rss)`	`Rdd=Rtt-Rss;`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sub(Rt,Rs):sat`	`Word32 Q6_R_sub_RR_sat(Word32 Rt, Word32 Rs)`
`Rdd=sub(Rtt,Rss)`	`Word64 Q6_P_sub_PP(Word64 Rtt, Word64 Rss)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	1	d	d	d	d	d	Rdd=sub(Rtt,Rss)
1	1	0	1	-	1	0	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	-	-	d	d	d	d	d	Rd=sub(Rt,Rs):sat

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Subtract and accumulate

Subtract Rs from Rt, then add the resulting value with Rx. The result is saved in Rx.

Syntax	Behavior
`Rx+=sub(Rt,Rs)`	`Rx=Rx + Rt - Rs;`

Type: M (slots 2,3)

Intrinsics

`Rx+=sub(Rt,Rs)`	`Word32 Q6_R_subacc_RR(Word32 Rx, Word32 Rt, Word32 Rs)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=sub(Rt,Rs)

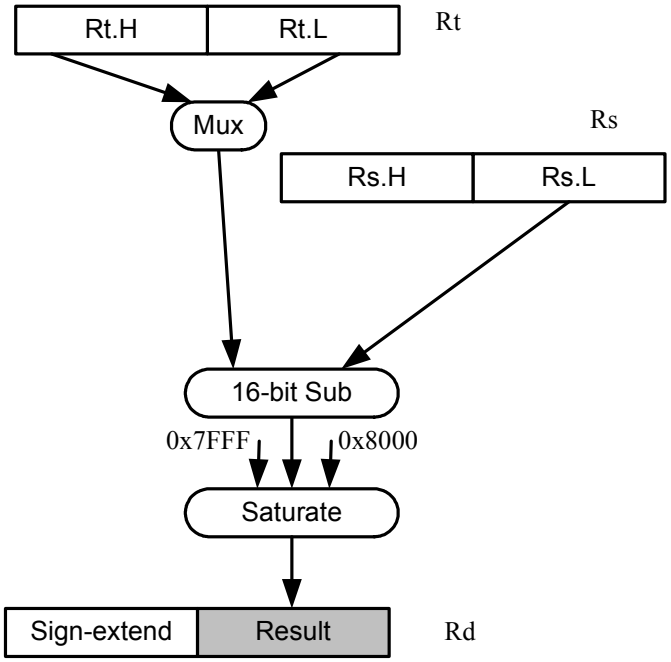
Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Subtract halfword

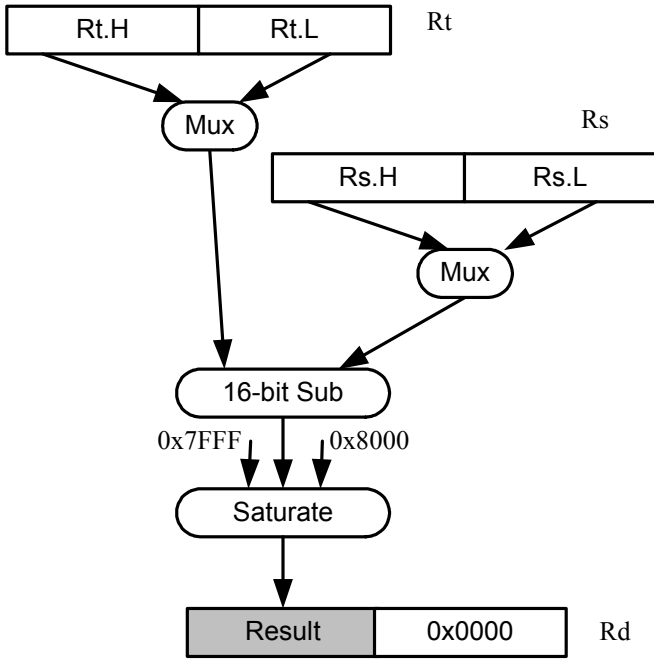
Perform a 16-bit subtract with optional saturation and place the result in either the upper or lower half of a register. If the result goes in the upper half, then the sources can be any high or low halfword of Rs and Rt. The lower 16 bits of the result are zeroed.

If the result is to be placed in the lower 16 bits of Rd, then the Rs source can be either high or low, but the other source must be the low halfword of Rt. In this case, the upper halfword of Rd is the sign-extension of the low halfword.

Rd=sub(Rt.[hl],Rs.l)[:sat]



Rd=sub(Rt.[hl],Rs.[hl])[:sat]:<<16



Syntax	Behavior
`Rd=sub(Rt.L,Rs.[HL])[:sat]`	`Rd=[sat_16](Rt.h[0]-Rs.h[01]);`
`Rd=sub(Rt.[HL],Rs.[HL])[:sat]:<<16`	`Rd=([sat_16](Rt.h[01]-Rs.h[01]))<<16;`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

Rd=sub(Rt.H,Rs.H):<<16	Word32 Q6_R_sub_RhRh_s16(Word32 Rt, Word32 Rs)
Rd=sub(Rt.H,Rs.H):sat:<<16	Word32 Q6_R_sub_RhRh_sat_s16(Word32 Rt, Word32 Rs)
Rd=sub(Rt.H,Rs.L):<<16	Word32 Q6_R_sub_RhRl_s16(Word32 Rt, Word32 Rs)
Rd=sub(Rt.H,Rs.L):sat:<<16	Word32 Q6_R_sub_RhRl_sat_s16(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.H)	Word32 Q6_R_sub_RlRh(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.H):<<16	Word32 Q6_R_sub_RlRh_s16(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.H):sat	Word32 Q6_R_sub_RlRh_sat(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.H):sat:<<16	Word32 Q6_R_sub_RlRh_sat_s16(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.L)	Word32 Q6_R_sub_RlRl(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.L):<<16	Word32 Q6_R_sub_RlRl_s16(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.L):sat	Word32 Q6_R_sub_RlRl_sat(Word32 Rt, Word32 Rs)
Rd=sub(Rt.L,Rs.L):sat:<<16	Word32 Q6_R_sub_RlRl_sat_s16(Word32 Rt, Word32 Rs)

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	1	0	1	0	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rd=sub(Rt.L,Rs.L)
1	1	0	1	-	1	0	1	0	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	d	d	d	d	d	Rd=sub(Rt.L,Rs.H)
1	1	0	1	-	1	0	1	0	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	d	d	d	d	d	Rd=sub(Rt.L,Rs.L):sat
1	1	0	1	-	1	0	1	0	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	d	d	d	d	d	Rd=sub(Rt.L,Rs.H):sat
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	0	d	d	d	d	d	Rd=sub(Rt.L,Rs.L):<<16
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	1	d	d	d	d	d	Rd=sub(Rt.L,Rs.H):<<16
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	0	d	d	d	d	d	Rd=sub(Rt.H,Rs.L):<<16
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	1	d	d	d	d	d	Rd=sub(Rt.H,Rs.H):<<16
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	0	d	d	d	d	d	Rd=sub(Rt.L,Rs.L):sat:<<16
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	1	d	d	d	d	d	Rd=sub(Rt.L,Rs.H):sat:<<16
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	0	d	d	d	d	d	Rd=sub(Rt.H,Rs.L):sat:<<16
1	1	0	1	-	1	0	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	1	d	d	d	d	d	Rd=sub(Rt.H,Rs.H):sat:<<16

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Sign extend word to doubleword

Sign-extend a 32-bit word to a 64-bit doubleword.

Syntax	Behavior
`Rdd=sxtw(Rs)`	`Rdd = sxt`$_{32->64}$`(Rs);`

Type: S (slots 2,3)

Intrinsics

`Rdd=sxtw(Rs)`	`Word64 Q6_P_sxtw_R(Word32 Rs)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=sxtw(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Vector absolute difference

For each element in the source vector Rss, subtract the corresponding element in source vector Rtt. Take the absolute value of the results, and store into Rdd.

This instruction is available for halfwords and words.

Syntax	Behavior

`Rdd=vabsdiffh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=ABS(Rtt.h[i] - Rss.h[i]);`
	`};`

`Rdd=vabsdiffw(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=ABS(Rtt.w[i] - Rss.w[i]);`
	`};`

Type: M (slots 2,3)

Intrinsics

`Rdd=vabsdiffh(Rtt,Rss)`	`Word64 Q6_P_vabsdiffh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vabsdiffw(Rtt,Rss)`	`Word64 Q6_P_vabsdiffw_PP(Word64 Rtt, Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffw(Rtt,Rss)
1	1	1	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffh(Rtt,Rss)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

XOR and XOR with destination

XOR the two source operands, then XOR the result with the destination register Rx.

This instruction effectively performs an add/subtract-and-accumulate operation in polynomial arithmetic. It is useful in encryption and error coding algorithms.

Syntax	Behavior
`Rx^=xor(Rs,Rt)`	`Rx^=Rs^Rt;`

Type: M (slots 2,3)

Intrinsics

`Rx^=xor(Rs,Rt)`	`Word32 Q6_R_xorxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	1	1	x	x	x	x	x	Rx^=xor(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

11.10.2 XTYPE / Bit

The XTYPE bit instructions include bit-level test and manipulation instructions.

Count leading

Count leading zeros (`cl0`) counts the number of consecutive zeros starting with the most significant bit.

Count leading ones (`cl1`) counts the number of consecutive ones starting with the most significant bit.

Count leading bits (`clb`) counts both leading ones and leading zeros and then selects the maximum.

The `NORMAMT` instruction returns the number of leading bits minus one.

For a two's-complement number, the number of leading zeros is zero for negative numbers. The number of leading ones is zero for positive numbers.

The number of leading bits can be used to judge the magnitude of the value.

Syntax	Behavior
`Rd=cl0(Rs)`	`Rd = count_leading_ones(~Rs);`
`Rd=cl0(Rss)`	`Rd = count_leading_ones(~Rss);`
`Rd=cl1(Rs)`	`Rd = count_leading_ones(Rs);`
`Rd=cl1(Rss)`	`Rd = count_leading_ones(Rss);`
`Rd=clb(Rs)`	`Rd = max(count_leading_ones(Rs),count_leading_ones(~Rs));`
`Rd=clb(Rss)`	`Rd = max(count_leading_ones(Rss),count_leading_ones(~Rss));`

`Rd=normamt(Rs)`	`if (Rs == 0) {`
	`  Rd = 0;`
	`} else {`
	`  Rd = (max(count_leading_ones(Rs),count_leading_ones(~Rs)))-1;`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rd=cl0(Rs)`	`Word32 Q6_R_cl0_R(Word32 Rs)`
`Rd=cl0(Rss)`	`Word32 Q6_R_cl0_P(Word64 Rss)`
`Rd=cl1(Rs)`	`Word32 Q6_R_cl1_R(Word32 Rs)`
`Rd=cl1(Rss)`	`Word32 Q6_R_cl1_P(Word64 Rss)`
`Rd=clb(Rs)`	`Word32 Q6_R_clb_R(Word32 Rs)`
`Rd=clb(Rss)`	`Word32 Q6_R_clb_P(Word64 Rss)`
`Rd=normamt(Rs)`	`Word32 Q6_R_normamt_R(Word32 Rs)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=clb(Rss)
1	0	0	0	1	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=cl0(Rss)
1	0	0	0	1	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=cl1(Rss)
1	0	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=clb(Rs)
1	0	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=cl0(Rs)
1	0	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cl1(Rs)
1	0	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=normamt(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Count trailing

Count trailing zeros (`ct0`) counts the number of consecutive zeros starting with the least significant bit.

Count trailing ones (`ct1`) counts the number of consecutive ones starting with the least significant bit.

Syntax	Behavior
`Rd=ct0(Rs)`	`Rd = count_leading_ones(~reverse_bits(Rs));`
`Rd=ct1(Rs)`	`Rd = count_leading_ones(reverse_bits(Rs));`

Type: S (slots 2,3)

Intrinsics

`Rd=ct0(Rs)`	`Word32 Q6_R_ct0_R(Word32 Rs)`
`Rd=ct1(Rs)`	`Word32 Q6_R_ct1_R(Word32 Rs)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=ct0(Rs)
1	0	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=ct1(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Compare bit mask

If all bits in the mask in Rt or a short immediate are set (BITSSET) or clear (BITSCLEAR), set the Pd to true. Otherwise, set the bits in Pd to false.

Syntax	Behavior
`Pd=bitsclr(Rs,#u6)`	`Pd=((Rs&#u)==0) ? 0xff : 0x00;`
`Pd=bitsclr(Rs,Rt)`	`Pd=((Rs&Rt)==0) ? 0xff : 0x00;`
`Pd=bitsset(Rs,Rt)`	`Pd=((Rs&Rt)==Rt) ? 0xff : 0x00;`

Type: S (slots 2,3)

Intrinsics

`Pd=bitsclr(Rs,#u6)`	`Byte Q6_p_bitsclr_RI(Word32 Rs, Word32 Iu)`
`Pd=bitsclr(Rs,Rt)`	`Byte Q6_p_bitsclr_RR(Word32 Rs, Word32 Rt)`
`Pd=bitsset(Rs,Rt)`	`Byte Q6_p_bitsset_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse															d2		
1	0	0	0	0	1	0	1	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	-	-	-	-	-	-	d	d	Pd=bitsclr(Rs,#u6)
ICLASS				RegType				Maj			s5					Parse			t5											d2		
1	1	0	0	0	1	1	1	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	-	-	-	d	d	Pd=bitsset(Rs,Rt)
1	1	0	0	0	1	1	1	1	0	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	-	-	-	d	d	Pd=bitsclr(Rs,Rt)

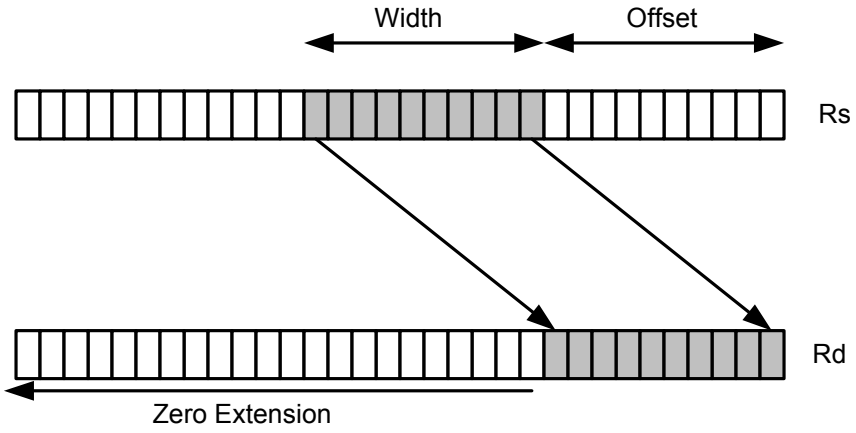
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
Maj	Major Opcode
RegType	Register Type
RegType	Register Type

Extract unsigned

Extract a bitfield from the source register (or register pair) and deposit into the least significant bits of the destination register (or register pair). The other, more significant bits in the destination are cleared. The width of the extracted field is obtained from the first immediate or from the most-significant word of Rtt. The field offset is obtained from either the second immediate or from the least-significant word of Rtt.

For register-based extract, where Rtt supplies the offset and width, the offset value is treated as a signed 7-bit number. In the case that this value is negative, the source register Rss is shifted left (the reverse direction). Width number of bits are then taken from the least-significant portion of this result.

If the shift amount and/or offset captures data beyond the most significant end of the input, then these bits will be taken as zero.



Syntax	Behavior

`Rd=extractu(Rs,#u5,#U5)`	`width=#u;`
	`offset=#U;`
	`Rd = zxt`$_{width->32}$`((Rs >> offset));`

`Rd=extractu(Rs,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rd = zxt`$_{width->64}$`((offset>0)?(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))>>>offset):(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))<<offset));`

`Rdd=extractu(Rss,#u6,#U6)`	`width=#u;`
	`offset=#U;`
	`Rdd = zxt`$_{width->64}$`((Rss >> offset));`

`Rdd=extractu(Rss,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rdd = zxt`$_{width->64}$`((offset>0)?(Rss>>>offset):(Rss<<offset));`

Type: S (slots 2,3)

Intrinsics

`Rd=extractu(Rs,#u5,#U5)`	`Word32 Q6_R_extractu_RII(Word32 Rs, Word32 Iu, Word32 IU)`
`Rd=extractu(Rs,Rtt)`	`Word32 Q6_R_extractu_RP(Word32 Rs, Word64 Rtt)`
`Rdd=extractu(Rss,#u6,#U6)`	`Word64 Q6_P_extractu_PII(Word64 Rss, Word32 Iu, Word32 IU)`
`Rdd=extractu(Rss,Rtt)`	`Word64 Q6_P_extractu_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	1	**l**	**l**	**l**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**l**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rdd=extractu(Rss,#u6,#U6)
1	0	0	0	1	1	0	1	0	**l**	**l**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**l**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rd=extractu(Rs,#u5,#U5)
ICLASS				RegType				Maj			s5					Parse			t5						Min			d5					
1	1	0	0	0	0	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=extractu(Rss,Rtt)
ICLASS				RegType							s5					Parse			t5									d5					
1	1	0	0	1	0	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=extractu(Rs,Rtt)

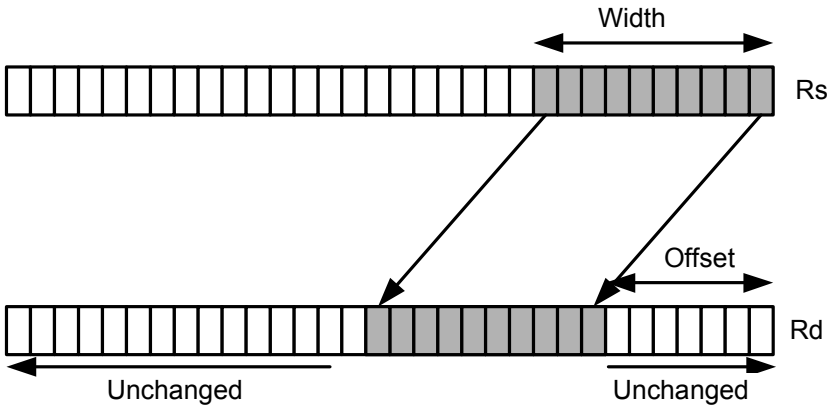
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

Insert bitfield

Replace a bitfield in the destination register (or register pair) with bits from the least significant portion of Rs/Rss. The number of bits is obtained from the first immediate or the most-significant word of Rtt. The bits are shifted by the second immediate or the least significant word of Rtt.

If register Rtt specifies the offset, then the low 7-bits of Rtt are treated as a signed 7-bit value. If this value is negative, the result is zero.

Shift amounts and offsets that are too large may push bits beyond the end of the destination register, in this case the bits will not appear in the destination register.



Syntax	Behavior

Rx=insert(Rs,#u5,#U5)	width=#u;
	offset=#U;
	Rx &= ~(((1<<width)-1)<<offset);
	Rx |= ((Rs & ((1<<width)-1)) << offset);

Rx=insert(Rs,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	mask = ((1<<width)-1);
	if (offset < 0) {
	Rx = 0;
	} else {
	Rx &= ~(mask<<offset);
	Rx |= ((Rs & mask) << offset);
	};

Rxx=insert(Rss,#u6,#U6)	width=#u;
	offset=#U;
	Rxx &= ~(((1<<width)-1)<<offset);
	Rxx |= ((Rss & ((1<<width)-1)) << offset);

Rxx=insert(Rss,Rtt)	width=zxt$_{6->32}$((Rtt.w[1]));
	offset=sxt$_{7->32}$((Rtt.w[0]));
	mask = ((1<<width)-1);
	if (offset < 0) {
	Rxx = 0;
	} else {
	Rxx &= ~(mask<<offset);
	Rxx |= ((Rss & mask) << offset);
	};

Type: S (slots 2,3)

Intrinsics

Rx=insert(Rs,#u5,#U5)	Word32 Q6_R_insert_RII(Word32 Rx, Word32 Rs, Word32 Iu, Word32 IU)
Rx=insert(Rs,Rtt)	Word32 Q6_R_insert_RP(Word32 Rx, Word32 Rs, Word64 Rtt)
Rxx=insert(Rss,#u6,#U6)	Word64 Q6_P_insert_PII(Word64 Rxx, Word64 Rss, Word32 Iu, Word32 IU)
Rxx=insert(Rss,Rtt)	Word64 Q6_P_insert_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	1	**I**	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**x**	**x**	**x**	**x**	**x**	Rxx=insert(Rss,#u6,#U6)
1	0	0	0	1	1	1	1	0	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**x**	**x**	**x**	**x**	**x**	Rx=insert(Rs,#u5,#U5)
ICLASS				RegType							s5					Parse			t5								x5					
1	1	0	0	1	0	0	0	**-**	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**x**	**x**	**x**	**x**	**x**	Rx=insert(Rs,Rtt)
1	1	0	0	1	0	1	0	**-**	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**x**	**x**	**x**	**x**	**x**	Rxx=insert(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
RegType	Register Type

Interleave/deinterleave

For `interleave`, bit I of Rs gets placed in bit (I*2)+1 of Rdd, and bit I of Rt gets placed in bit (I*2) of Rdd.

For `deinterleave`, the even bits of the source register are placed in the even register of the result pair, and the odd bits of the source register are placed in the odd register of the result pair.

`r1:0 = deinterleave(r1:0)` is the inverse of `r1:0 = interleave(r1:0)`.

Syntax	Behavior
`Rdd=deinterleave(Rss)`	`Rdd = deinterleave(ODD,EVEN);`
`Rdd=interleave(Rss)`	`Rdd = interleave(Rss.w[1],Rss.w[0]);`

Type: S (slots 2,3)

Intrinsics

`Rdd=deinterleave(Rss)`	`Word64 Q6_P_deinterleave_P(Word64 Rss)`
`Rdd=interleave(Rss)`	`Word64 Q6_P_interleave_P(Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=deinterleave(Rss)
1	0	0	0	0	0	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=interleave(Rss)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Linear Feedback-Shift Iteration

Count the number of ones of the logical AND of the two source input values, and take the least significant value of that sum.

For the `parity` instruction, the result is placed in a register

For the `lfs` instruction, the first source value is shifted right by one bit, and the parity is placed in the MSB.

Syntax	Behavior
`Rdd=lfs(Rss,Rtt)`	`Rdd = (Rss.u64 >> 1) | ((1&count_ones(Rss & Rtt)).u64<<63) ;`

Type: S (slots 2,3)

Intrinsics

`Rdd=lfs(Rss,Rtt)`	`Word64 Q6_P_lfs_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	d	d	d	d	d	Rdd=lfs(Rss,Rtt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Masked parity

Count the number of ones of the logical AND of the two source input values, and take the least significant value of that sum. The first source value is shifted right by one bit, and the parity is placed in the MSB.

Syntax	Behavior
`Rd=parity(Rss,Rtt)`	`Rd = 1&count_ones(Rss & Rtt);`

Type: ALU64 (slots 2,3)

Intrinsics

`Rd=parity(Rss,Rtt)`	`Word32 Q6_R_parity_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType						s5					Parse			t5								d5					
1	1	0	1	-	0	0	0	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=parity(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Bit reverse

Reverse the order of bits. The most significant bit gets swapped with the least significant bit, bit 30 gets swapped with bit 1, and so on.

Syntax	Behavior
`Rd=brev(Rs)`	`Rd = reverse_bits(Rs);`

Type: S (slots 2,3)

Intrinsics

`Rd=brev(Rs)`	`Word32 Q6_R_brev_R(Word32 Rs)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	0	1	-	s	s	s	s	s	P	P	-	-	-	-	-	-	1	1	0	d	d	d	d	d	Rd=brev(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Set/clear/toggle bit

Set (to 1), clear (to 0), or toggle a single bit in the source, and place the resulting value in the destination. The bit to be manipulated can be indicated using an immediate or register value.

In the case that a register is used to indicate the bit position, and the value specified is out of range, the destination register will be unchanged.

Syntax	Behavior
`Rd=clrbit(Rs,#u5)`	`Rd = (Rs & (~(1<<#u)));`
`Rd=clrbit(Rs,Rt)`	`Rd = (Rs & (~((sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)))));`
`Rd=setbit(Rs,#u5)`	`Rd = (Rs | (1<<#u));`
`Rd=setbit(Rs,Rt)`	`Rd = (Rs | (sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)));`
`Rd=togglebit(Rs,#u5)`	`Rd = (Rs ^ (1<<#u));`
`Rd=togglebit(Rs,Rt)`	`Rd = (Rs ^ (sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)));`

Type: S (slots 2,3)

Intrinsics

`Rd=clrbit(Rs,#u5)`	`Word32 Q6_R_clrbit_RI(Word32 Rs, Word32 Iu)`
`Rd=clrbit(Rs,Rt)`	`Word32 Q6_R_clrbit_RR(Word32 Rs, Word32 Rt)`
`Rd=setbit(Rs,#u5)`	`Word32 Q6_R_setbit_RI(Word32 Rs, Word32 Iu)`
`Rd=setbit(Rs,Rt)`	`Word32 Q6_R_setbit_RR(Word32 Rs, Word32 Rt)`
`Rd=togglebit(Rs,#u5)`	`Word32 Q6_R_togglebit_RI(Word32 Rs, Word32 Iu)`
`Rd=togglebit(Rs,Rt)`	`Word32 Q6_R_togglebit_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=setbit(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=clrbit(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=togglebit(Rs,#u5)
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=setbit(Rs,Rt)
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=clrbit(Rs,Rt)
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=togglebit(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

Test bit

Extract a bit from a register. If the bit is true (1), set all the bits of the predicate register destination to 1. If the bit is false (0), set all the bits of the predicate register destination to 0. The bit to be tested can be indicated using an immediate or register value.

In the case that a register is used to indicate the bit to test, and the value specified is out of range, the predicate result will be zero.

Syntax	Behavior
`Pd=tstbit(Rs,#u5)`	`Pd = ((Rs & (1<<#u)) != 0) ? 0xff : 0x00;`
`Pd=tstbit(Rs,Rt)`	`Pd = ((zxt`$_{32->64}$`(Rs) & (sxt`$_{7->32}$`(Rt)>0)?(zxt`$_{32->64}$`(1)<<sxt`$_{7->32}$`(Rt)):(zxt`$_{32->64}$`(1)>>>sxt`$_{7->32}$`(Rt))) != 0) ? 0xff : 0x00;`

Type: S (slots 2,3)

Intrinsics

`Pd=tstbit(Rs,#u5)`	`Byte Q6_p_tstbit_RI(Word32 Rs, Word32 Iu)`
`Pd=tstbit(Rs,Rt)`	`Byte Q6_p_tstbit_RR(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse														d2		
1	0	0	0	0	1	0	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=tstbit(Rs,#u5)
ICLASS				RegType				Maj			s5					Parse			t5										d2		
1	1	0	0	0	1	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=tstbit(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
Maj	Major Opcode
RegType	Register Type
RegType	Register Type

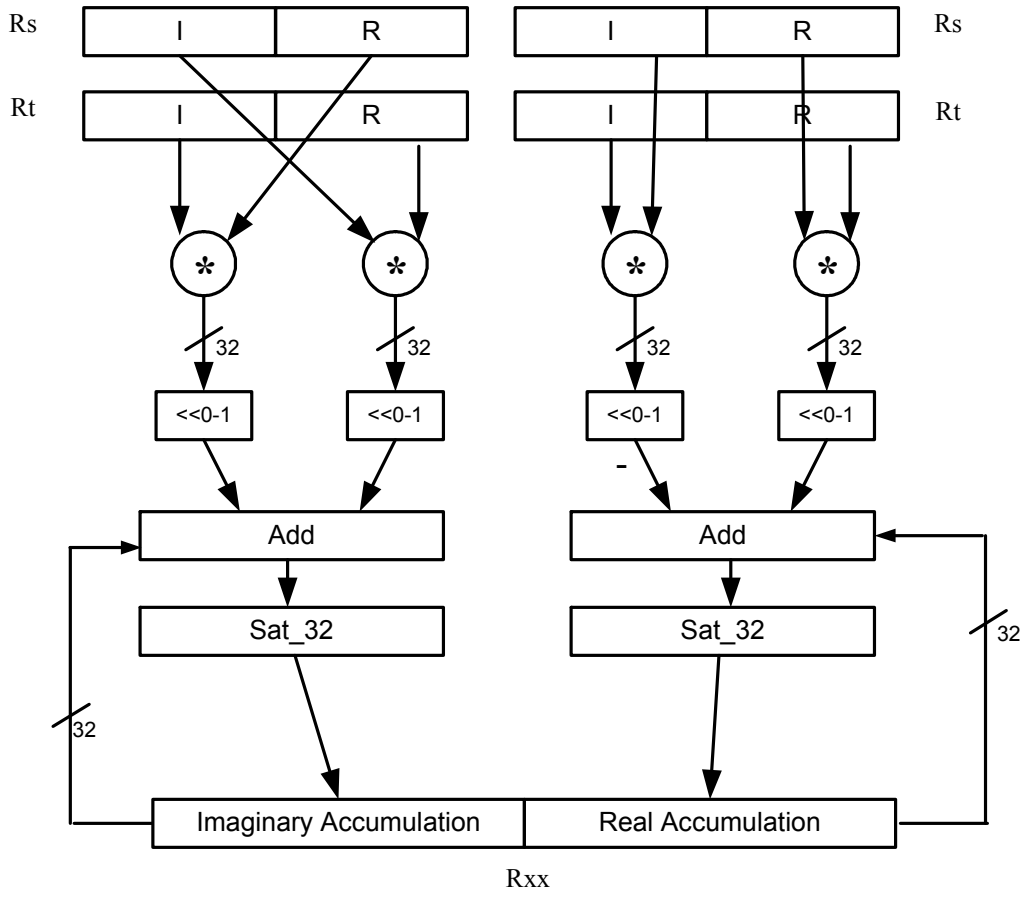
11.10.3 XTYPE / Complex

The XTYPE complex instructions perform complex math operations on imaginary values.

Complex multiply

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. Optionally, scale the result by 0-1 bits. Optionally, add a complex accumulator. Saturate the real and imaginary portions to 32-bits. The output has a real 32-bit value in the low word and an imaginary 32-bit value in the high word. The Rt input can be optionally conjugated. Another option is that the result can be subtracted from the destination rather than accumulated.

Rxx+=cmpy(Rs,Rt):sat



Syntax	Behavior
`Rdd=cmpy(Rs,Rt)[:<<1]:sat`	`Rdd.w[1]=sat_32((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1]);` `Rdd.w[0]=sat_32((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]);`
`Rdd=cmpy(Rs,Rt*)[:<<1]:sat`	`Rdd.w[1]=sat_32((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1]);` `Rdd.w[0]=sat_32((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=cmpy(Rs,Rt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1]);` `Rxx.w[0]=sat_32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=cmpy(Rs,Rt*)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1]);` `Rxx.w[0]=sat_32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx-=cmpy(Rs,Rt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] - ((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1]));` `Rxx.w[0]=sat_32(Rxx.w[0] - ((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]));`
`Rxx-=cmpy(Rs,Rt*)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] - ((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1]));` `Rxx.w[0]=sat_32(Rxx.w[0] - ((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]));`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpy_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpy_RR_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpy_RR_conj_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpy_RR_conj_sat(Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpyacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpyacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpyacc_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpyacc_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpynac_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpynac_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpynac_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpynac_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

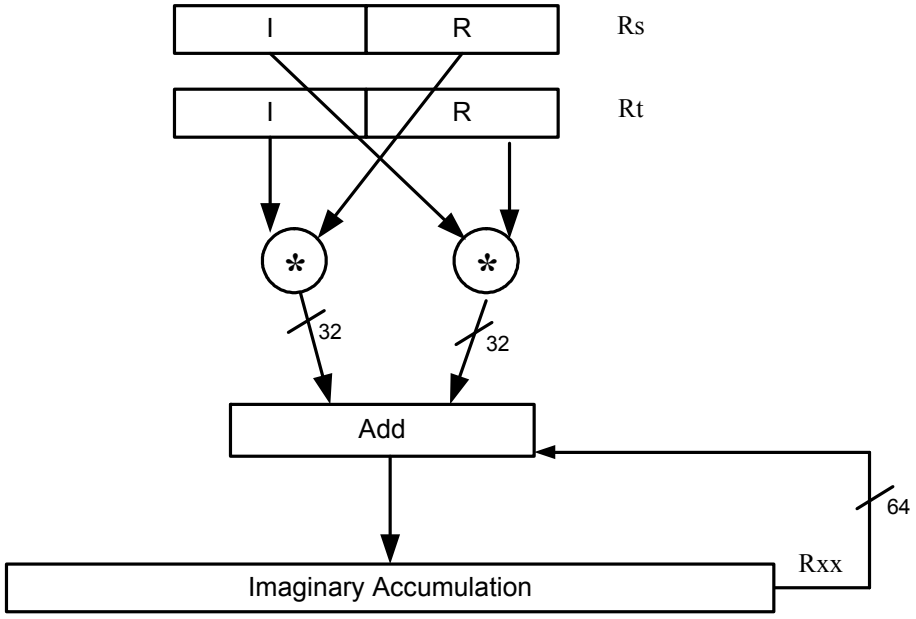
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	0	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpy(Rs,Rt*)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpy(Rs,Rt*)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=cmpy(Rs,Rt*)[:<<N]:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Complex multiply real or imaginary

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. Take either the real or imaginary result and optionally accumulate with a 64-bit destination.

Rxx+=cmpyi(Rs,Rt)



Syntax	Behavior
`Rdd=cmpyi(Rs,Rt)`	`Rdd = (Rs.h[1] * Rt.h[0]) + (Rs.h[0] * Rt.h[1]);`
`Rdd=cmpyr(Rs,Rt)`	`Rdd = (Rs.h[0] * Rt.h[0]) - (Rs.h[1] * Rt.h[1]);`
`Rxx+=cmpyi(Rs,Rt)`	`Rxx = Rxx + (Rs.h[1] * Rt.h[0]) + (Rs.h[0] * Rt.h[1]);`
`Rxx+=cmpyr(Rs,Rt)`	`Rxx = Rxx + (Rs.h[0] * Rt.h[0]) - (Rs.h[1] * Rt.h[1]);`

Type: M (slots 2,3)

Intrinsics

`Rdd=cmpyi(Rs,Rt)`	`Word64 Q6_P_cmpyi_RR(Word32 Rs, Word32 Rt)`
`Rdd=cmpyr(Rs,Rt)`	`Word64 Q6_P_cmpyr_RR(Word32 Rs, Word32 Rt)`
`Rxx+=cmpyi(Rs,Rt)`	`Word64 Q6_P_cmpyiacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpyr(Rs,Rt)`	`Word64 Q6_P_cmpyracc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

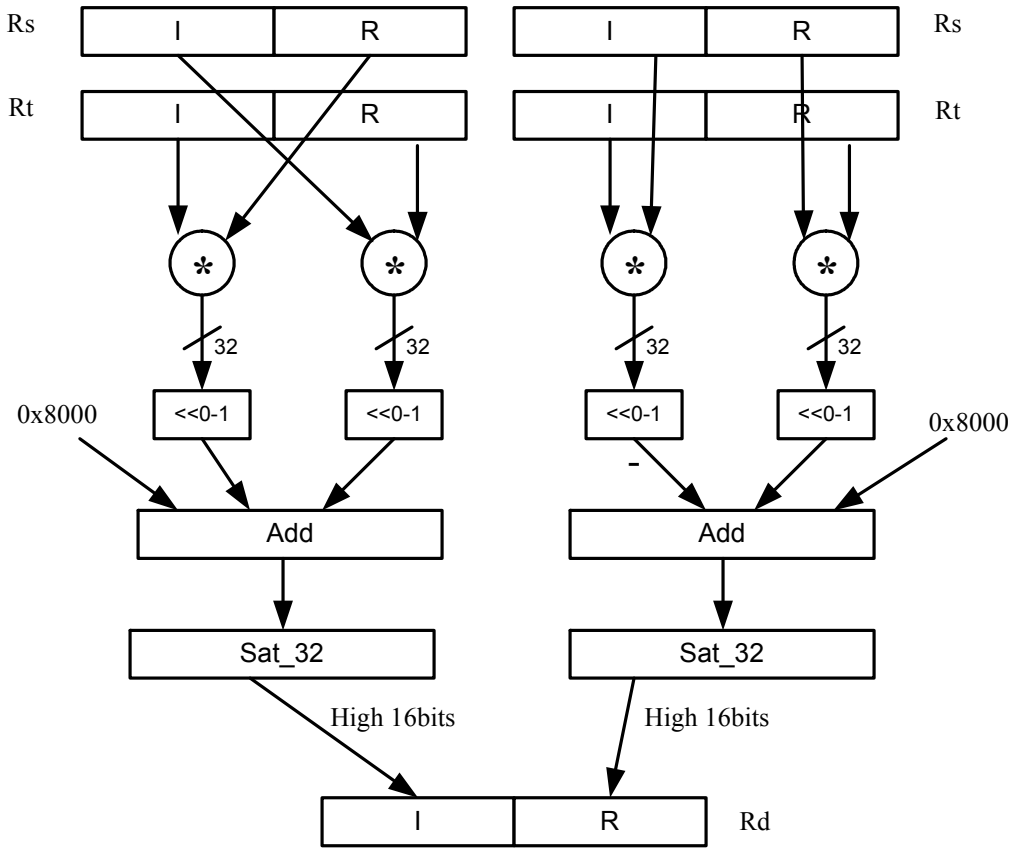
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyi(Rs,Rt)
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyr(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyi(Rs,Rt)
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyr(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Complex multiply with round and pack

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. The Rt input is optionally conjugated. The multiplier results are optionally scaled by 0-1 bits. A rounding constant is added to each real and imaginary sum. The real and imaginary parts are individually saturated to 32bits. The upper 16 bits of each 32-bit result are packed in a 32-bit destination register.

Rd=cmpy(Rs,Rt):rnd:sat



Syntax	Behavior
`Rd=cmpy(Rs,Rt)[:<<1]:rnd:sat`	`Rd.h[1]=(sat_32((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1] + 0x8000)).h[1]; Rd.h[0]=(sat_32((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];`
`Rd=cmpy(Rs,Rt*)[:<<1]:rnd:sat`	`Rd.h[1]=(sat_32((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1] + 0x8000)).h[1]; Rd.h[0]=(sat_32((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=cmpy(Rs,Rt):<<1:rnd:sat`	`Word32 Q6_R_cmpy_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt):rnd:sat`	`Word32 Q6_R_cmpy_RR_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt*):<<1:rnd:sat`	`Word32 Q6_R_cmpy_RR_conj_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt*):rnd:sat`	`Word32 Q6_R_cmpy_RR_conj_rnd_sat(Word32 Rs, Word32 Rt)`

Encoding

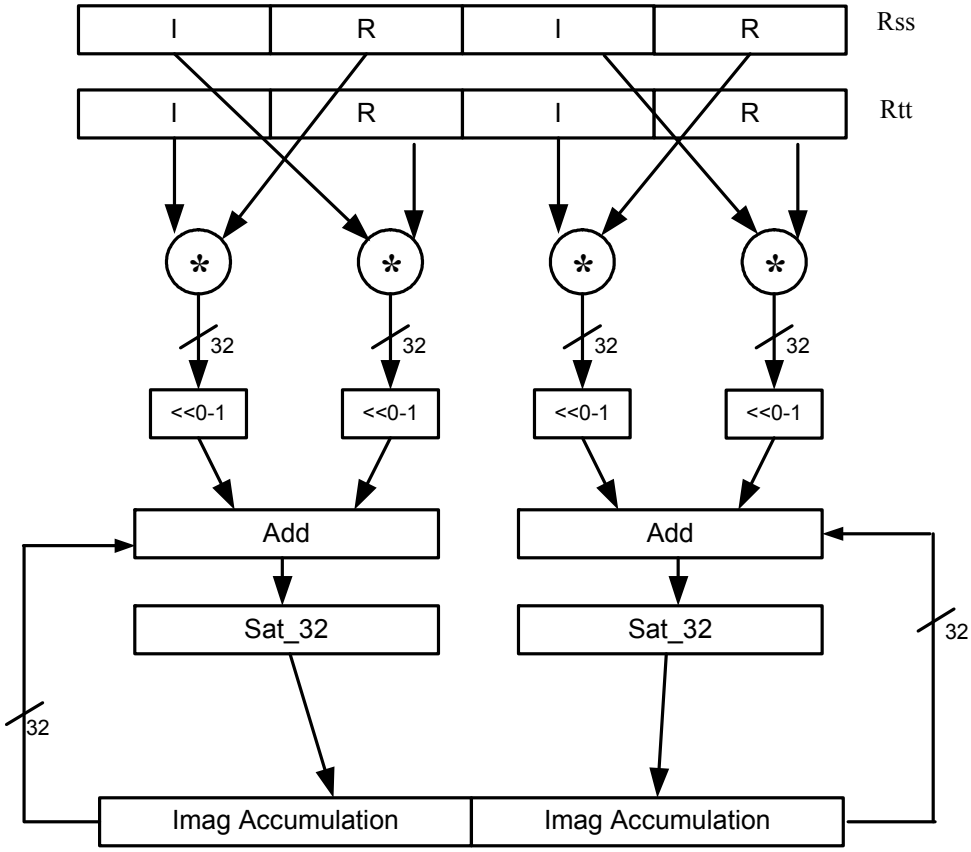
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp		d5						
1	1	1	0	1	1	0	1	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpy(Rs,Rt)[:<<N]:rnd:sat
1	1	1	0	1	1	0	1	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpy(Rs,Rt*)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector complex multiply real or imaginary

The inputs Rss and Rtt are a vector of two complex values. Each complex value is composed of a 16-bit imaginary portion in the upper halfword and a 16-bit real portion in the lower halfword. Generate two complex results, either the real result or the imaginary result. These results are optionally shifted left by 0-1 bits, and optionally accumulated with the destination register.

Rxx+=vcmpyi(Rss,Rtt):sat



Syntax	Behavior
`Rdd=vcmpyi(Rss,Rtt)[:<<1]:sat`	`Rdd.w[0]=sat_32((Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1])[<<1]);` `Rdd.w[1]=sat_32((Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3])[<<1]);`
`Rdd=vcmpyr(Rss,Rtt)[:<<1]:sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1])[<<1]);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3])[<<1]);`
`Rxx+=vcmpyi(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3])<<0);`
`Rxx+=vcmpyr(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3])<<0);`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vcmpyi(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vcmpyi_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyi(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyi_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyr(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vcmpyr_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyr(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyr_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vcmpyi(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyiacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vcmpyr(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyracc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vcmpyr(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vcmpyi(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vcmpyr(Rss,Rtt):sat
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vcmpyi(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector complex conjugate

Perform a vector complex conjugate of both complex values in vector Rss. This is done by negating the imaginary halfwords, and placing the result in destination Rdd.

Syntax	Behavior

`Rdd=vconj(Rss):sat`	`Rdd.h[1]=sat_16(-Rss.h[1]);`
	`Rdd.h[0]=Rss.h[0];`
	`Rdd.h[3]=sat_16(-Rss.h[3]);`
	`Rdd.h[2]=Rss.h[2];`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vconj(Rss):sat`	`Word64 Q6_P_vconj_P_sat(Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse							MinOp			d5						
1	0	0	0	0	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vconj(Rss):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector complex rotate

Take the least significant bits of Rt, and use these bits to rotate each of the two complex values in the source vector a multiple of 90 degrees. Bits 0 and 1 control the rotation factor for word 0, and bits 2 and 3 control the rotation factor for word 1.

If the rotation control bits are 0, the rotation is 0: the real and imaginary halves of the source appear unchanged and unmoved in the destination.

If the rotation control bits are 1, the rotation is -pi/2: the real half of the destination gets the imaginary half of the source, and the imaginary half of the destination gets the negative real half of the source.

If the rotation control bits are 2, the rotation is pi/2: the real half of the destination gets the negative imaginary half of the source, and the imaginary half of the destination gets the real half of the source.

If the rotation control bits are 3, the rotation is pi: the real half of the destination gets the negative real half of the source, and the imaginary half of the destination gets the negative imaginary half of the source.

Syntax	Behavior

`Rdd=vcrotate(Rss,Rt)`	`tmp = Rt[1:0];`
	`if (tmp == 0) {`
	`  Rdd.h[0]=Rss.h[0];`
	`  Rdd.h[1]=Rss.h[1];`
	`} else if (tmp == 1) {`
	`  Rdd.h[0]=Rss.h[1];`
	`  Rdd.h[1]=sat_16(-Rss.h[0]);`
	`} else if (tmp == 2) {`
	`  Rdd.h[0]=sat_16(-Rss.h[1]);`
	`  Rdd.h[1]=Rss.h[0];`
	`} else {`
	`  Rdd.h[0]=sat_16(-Rss.h[0]);`
	`  Rdd.h[1]=sat_16(-Rss.h[1]);`
	`};`
	`tmp = Rt[3:2];`
	`if (tmp == 0) {`
	`  Rdd.h[2]=Rss.h[2];`
	`  Rdd.h[3]=Rss.h[3];`
	`} else if (tmp == 1) {`
	`  Rdd.h[2]=Rss.h[3];`
	`  Rdd.h[3]=sat_16(-Rss.h[2]);`
	`} else if (tmp == 2) {`
	`  Rdd.h[2]=sat_16(-Rss.h[3]);`
	`  Rdd.h[3]=Rss.h[2];`
	`} else {`
	`  Rdd.h[2]=sat_16(-Rss.h[2]);`
	`  Rdd.h[3]=sat_16(-Rss.h[3]);`
	`};`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vcrotate(Rss,Rt)`	`Word64 Q6_P_vcrotate_PR(Word64 Rss, Word32 Rt)`

Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vcrotate(Rss,Rt)

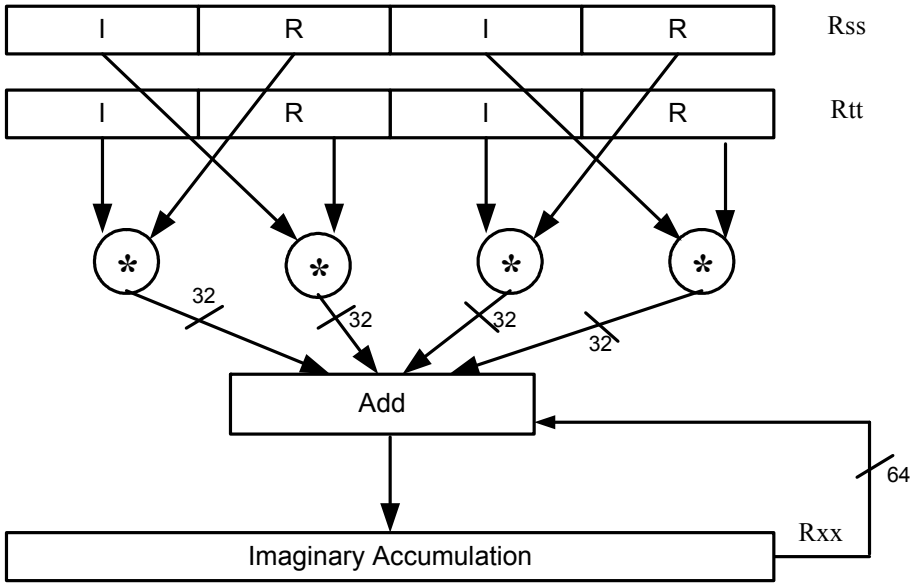
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Vector reduce complex multiply real or imaginary

The input vectors are two packed complex values, each with a real low halfword and imaginary high halfword. Compute either the real or imaginary products, add the intermediate results together and optionally accumulate with the destination. The Rtt input is optionally conjugated (negate the imaginary portion) before multiplication.

Using `vrcmpyr` and `vrcmpyi`, it is possible to sustain an average of one full complex multiply per cycle in a complex FIR, while also keeping both the real and imaginary accumulators in full-precision 64-bit values.

Rxx+=vrcmpyi(Rss,Rtt)



Syntax	Behavior
Rdd=vrcmpyi(Rss,Rtt)	Rdd = (Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3]);
Rdd=vrcmpyi(Rss,Rtt*)	Rdd = (Rss.h[1] * Rtt.h[0]) - (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) - (Rss.h[2] * Rtt.h[3]);
Rdd=vrcmpyr(Rss,Rtt)	Rdd = (Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3]);
Rdd=vrcmpyr(Rss,Rtt*)	Rdd = (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);
Rxx+=vrcmpyi(Rss,Rtt)	Rxx = Rxx + (Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3]);
Rxx+=vrcmpyi(Rss,Rtt*)	Rxx = Rxx + (Rss.h[1] * Rtt.h[0]) - (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) - (Rss.h[2] * Rtt.h[3]);
Rxx+=vrcmpyr(Rss,Rtt)	Rxx = Rxx + (Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3]);
Rxx+=vrcmpyr(Rss,Rtt*)	Rxx = Rxx + (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);

Type: M (slots 2,3)

Intrinsics

Rdd=vrcmpyi(Rss,Rtt)	Word64 Q6_P_vrcmpyi_PP(Word64 Rss, Word64 Rtt)
Rdd=vrcmpyi(Rss,Rtt*)	Word64 Q6_P_vrcmpyi_PP_conj(Word64 Rss, Word64 Rtt)
Rdd=vrcmpyr(Rss,Rtt)	Word64 Q6_P_vrcmpyr_PP(Word64 Rss, Word64 Rtt)

`Rdd=vrcmpyr(Rss,Rtt*)`	`Word64 Q6_P_vrcmpyr_PP_conj(Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyi(Rss,Rtt)`	`Word64 Q6_P_vrcmpyiacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyi(Rss,Rtt*)`	`Word64 Q6_P_vrcmpyiacc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyr(Rss,Rtt)`	`Word64 Q6_P_vrcmpyracc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyr(Rss,Rtt*)`	`Word64 Q6_P_vrcmpyracc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyi(Rss,Rtt)
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyr(Rss,Rtt)
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyi(Rss,Rtt*)
1	1	1	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyr(Rss,Rtt*)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyi(Rss,Rtt)
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyr(Rss,Rtt)
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyi(Rss,Rtt*)
1	1	1	0	1	0	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyr(Rss,Rtt*)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

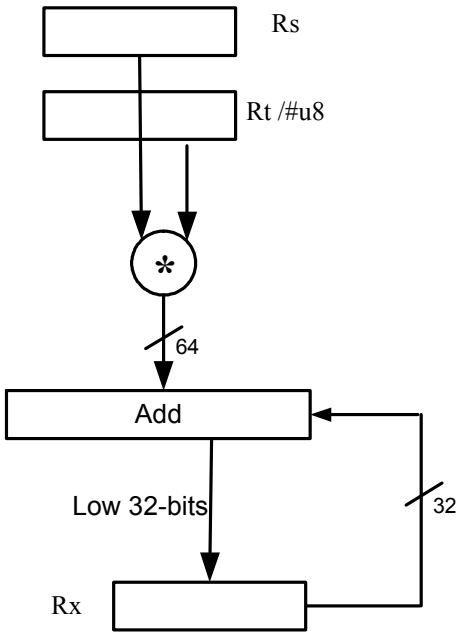
11.10.4 XTYPE / Higher Precision Multiply

The XTYPE higher-precision multiply instructions perform 32x16 and 32x32 multiplication.

Multiply and use lower result

Multiply the signed 32-bit integer in Rs by either the signed 32-bit integer in Rt or an unsigned immediate value. The 64-bit result is optionally accumulated with the 32-bit destination. The least-significant 32 bits of the result are written to the single destination register.

This multiply produces the correct results for the ANSI C multiplication of two signed or unsigned integers with an integer result.



Syntax	Behavior
`Rd=+mpyi(Rs,#u8)`	`Rd=Rs*#u;`
`Rd=-mpyi(Rs,#u8)`	`Rd=Rs*-#u;`
`Rd=mpyi(Rs,#m9)`	`if ("#m9<0") { Assembler maps to: "Rd=-mpyi(Rs,#m9*(-1))"; } else { Assembler maps to: "Rd=+mpyi(Rs,#m9)"; };`

Syntax	Behavior
Rd=mpyi(Rs,Rt)	Rd=Rs*Rt;
Rd=mpyui(Rs,Rt)	Assembler maps to: "Rd=mpyi(Rs,Rt)"
Rx+=mpyi(Rs,#u8)	Rx=Rx + (Rs*#u);
Rx+=mpyi(Rs,Rt)	Rx=Rx + Rs*Rt;
Rx-=mpyi(Rs,#u8)	Rx=Rx - (Rs*#u);

Type: M (slots 2,3)

Intrinsics

Rd=mpyi(Rs,#m9)	Word32 Q6_R_mpyi_RI(Word32 Rs, Word32 Im)
Rd=mpyi(Rs,Rt)	Word32 Q6_R_mpyi_RR(Word32 Rs, Word32 Rt)
Rd=mpyui(Rs,Rt)	Word32 Q6_R_mpyui_RR(Word32 Rs, Word32 Rt)
Rx+=mpyi(Rs,#u8)	Word32 Q6_R_mpyiacc_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx+=mpyi(Rs,Rt)	Word32 Q6_R_mpyiacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyi(Rs,#u8)	Word32 Q6_R_mpyinac_RI(Word32 Rx, Word32 Rs, Word32 Iu)

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	1	1	0	0	0	0	0	0	-	-	s	s	s	s	s	P	P	-	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=+mpyi(Rs,#u8)
1	1	1	0	0	0	0	0	1	-	-	s	s	s	s	s	P	P	-	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=-mpyi(Rs,#u8)
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	1	1	0	0	0	0	1	0	-	-	s	s	s	s	s	P	P	-	i	i	i	i	i	i	i	i	x	x	x	x	x	Rx+=mpyi(Rs,#u8)
1	1	1	0	0	0	0	1	1	-	-	s	s	s	s	s	P	P	-	i	i	i	i	i	i	i	i	x	x	x	x	x	Rx-=mpyi(Rs,#u8)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	1	0	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	0	d	d	d	d	d	Rd=mpyi(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			x5					
1	1	1	0	1	1	1	1	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	0	0	x	x	x	x	x	Rx+=mpyi(Rs,Rt)

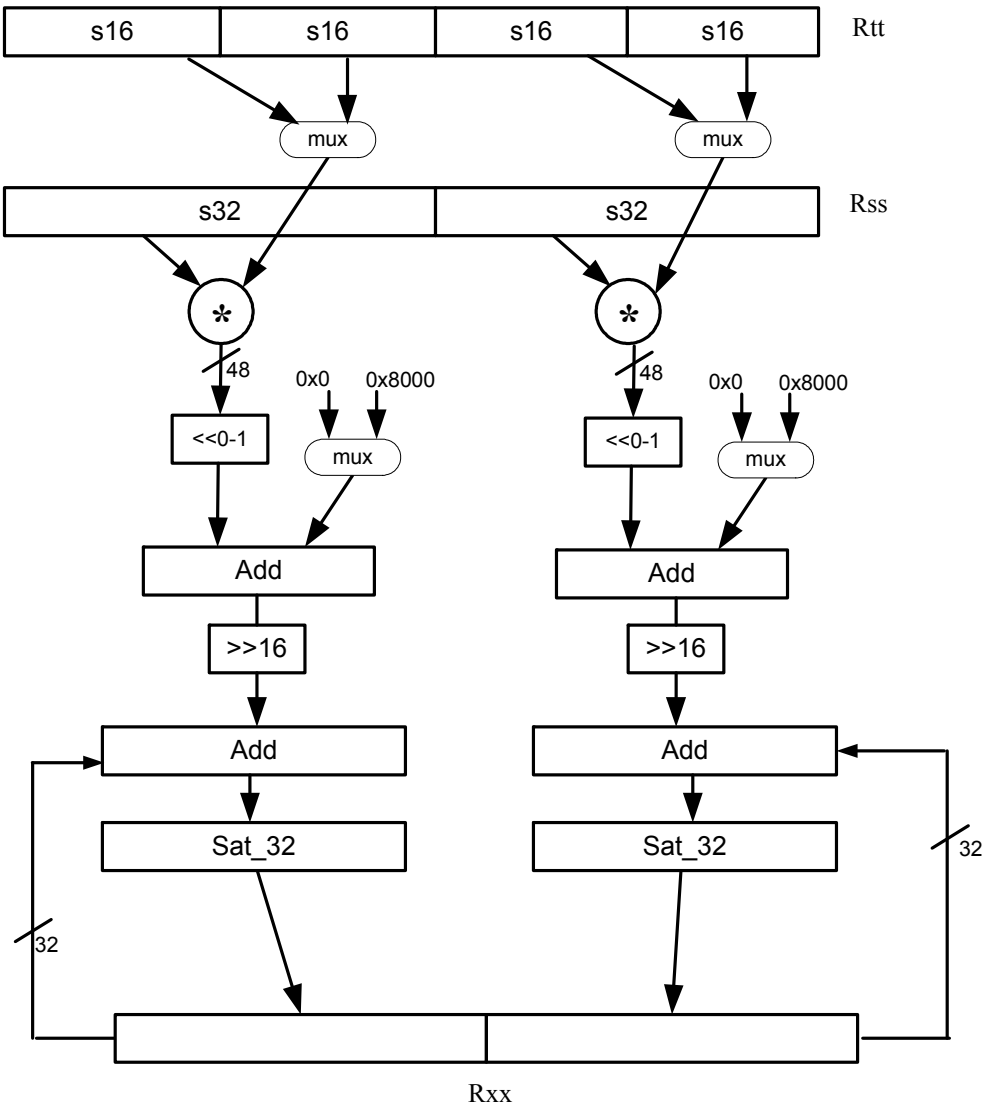
Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Field name	Description
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply word by signed half (32x16)

Perform mixed precision vector multiply operations. A 32-bit word from vector Rss is multiplied by a 16-bit halfword (either even or odd) from vector Rtt. The multiplication is performed as a signed 32x16, which produces a 48-bit result. This result is optionally scaled left by one bit. This result is then shifted right by 16 bits, optionally accumulated and then saturated to 32 bits.

This operation is available in vector form (`vmpyweh`/`vmpywoh`) and non-vector form (`mpyweh`/`mpywoh`).



Syntax	Behavior
`Rdd=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[2])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[0])[<<1]+0x8000)>>16);`
`Rdd=vmpyweh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[2])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[0])[<<1])>>16);`
`Rdd=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[3])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[1])[<<1]+0x8000)>>16);`
`Rdd=vmpywoh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[3])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[1])[<<1])>>16);`
`Rxx+=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[2])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[0])[<<1]+0x8000)>>16));`
`Rxx+=vmpyweh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[2])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[0])[<<1])>>16));`
`Rxx+=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[3])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[1])[<<1]+0x8000)>>16 ));`
`Rxx+=vmpywoh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[3])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[1])[<<1])>>16 ));`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

Rdd=vmpyweh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpyweh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweh_PP_s1_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpyweh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweh_PP_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpyweh(Rss,Rtt):sat	Word64 Q6_P_vmpyweh_PP_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywoh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywoh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywoh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywoh_PP_s1_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywoh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywoh_PP_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywoh(Rss,Rtt):sat	Word64 Q6_P_vmpywoh_PP_sat(Word64 Rss, Word64 Rtt)
Rxx+=vmpyweh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywehacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpyweh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpyweh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywehacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpyweh(Rss,Rtt):sat	Word64 Q6_P_vmpywehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywoh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywohacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywoh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywohacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywoh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywohacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywoh(Rss,Rtt):sat	Word64 Q6_P_vmpywohacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)

Encoding

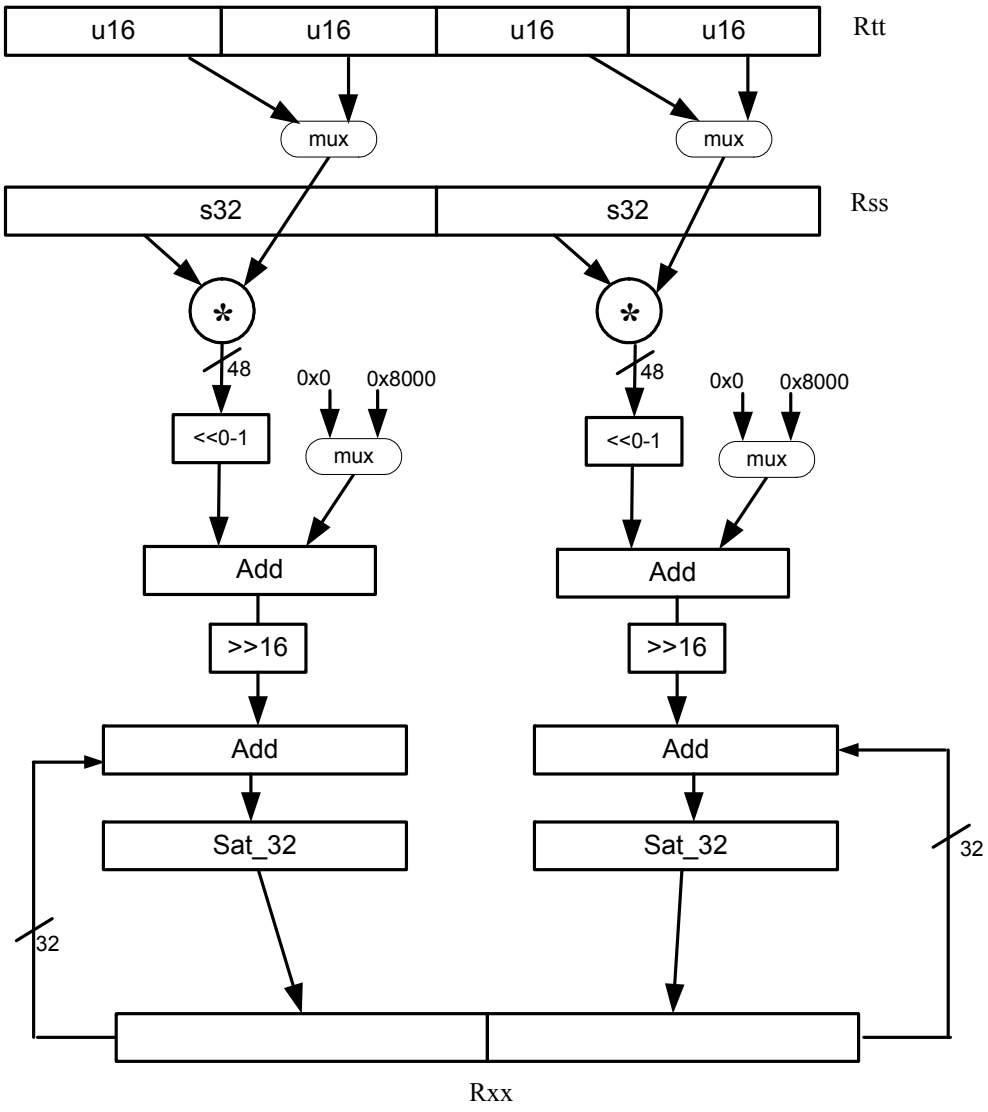
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywoh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywoh(Rss,Rtt)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywoh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywoh(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply word by unsigned half (32x16)

Perform mixed precision vector multiply operations. A 32-bit signed word from vector Rss is multiplied by a 16-bit unsigned halfword (either odd or even) from vector Rtt. This multiplication produces a 48-bit result. This result is optionally scaled left by one bit, and then a rounding constant is optionally added to the lower 16 bits. This result is then shifted right by 16 bits, optionally accumulated and then saturated to 32 bits.

This is a dual vector operation, and is performed for both high and low word of Rss.



Syntax	Behavior
`Rdd=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[2])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[0])[<<1]+0x8000)>>16);`
`Rdd=vmpyweuh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[2])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[0])[<<1])>>16);`
`Rdd=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[3])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[1])[<<1]+0x8000)>>16);`
`Rdd=vmpywouh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[3])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[1])[<<1])>>16);`
`Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[2])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[0])[<<1]+0x8000)>>16));`
`Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[2])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[0])[<<1])>>16));`
`Rxx+=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[3])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[1])[<<1]+0x8000)>>16 ));`
`Rxx+=vmpywouh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[3])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[1])[<<1])>>16 ));`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

Rdd=vmpyweuh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweuh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpyweuh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweuh_PP_s1_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpyweuh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweuh_PP_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpyweuh(Rss,Rtt):sat	Word64 Q6_P_vmpyweuh_PP_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywouh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywouh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywouh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywouh_PP_s1_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywouh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywouh_PP_rnd_sat(Word64 Rss, Word64 Rtt)
Rdd=vmpywouh(Rss,Rtt):sat	Word64 Q6_P_vmpywouh_PP_sat(Word64 Rss, Word64 Rtt)
Rxx+=vmpyweuh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweuhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpyweuh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweuhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpyweuh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweuhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpyweuh(Rss,Rtt):sat	Word64 Q6_P_vmpyweuhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywouh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywouhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywouh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywouhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywouh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywouhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)
Rxx+=vmpywouh(Rss,Rtt):sat	Word64 Q6_P_vmpywouhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)

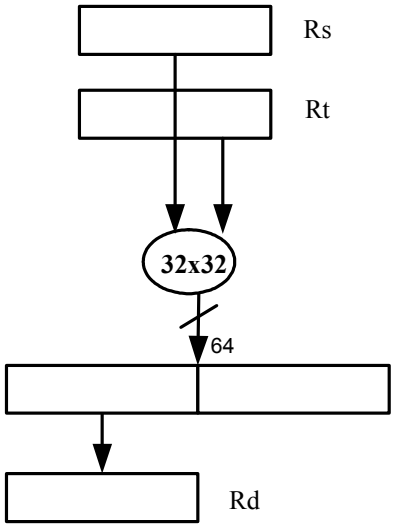
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweuh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywouh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweuh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	0	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywouh(Rss,Rtt)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweuh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywouh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweuh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywouh(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Multiply and use upper result

Multiply two signed or unsigned 32-bit words. Store the upper 32 bits of this result to a single destination register. Optional rounding is available.



Syntax	Behavior
`Rd=mpy(Rs,Rt.H):<<1:rnd:sat`	`Rd = sat_32(((Rs * Rt.h[1])<<1+0x8000)>>16);`
`Rd=mpy(Rs,Rt.L):<<1:rnd:sat`	`Rd = sat_32(((Rs * Rt.h[0])<<1+0x8000)>>16);`
`Rd=mpy(Rs,Rt)`	`Rd=(Rs * Rt)>>32;`
`Rd=mpy(Rs,Rt):rnd`	`Rd=((Rs * Rt)+0x80000000)>>32;`
`Rd=mpyu(Rs,Rt)`	`Rd=(Rs.uw[0] * Rt.uw[0])>>32;`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=mpy(Rs,Rt.H):<<1:rnd:sat`	`Word32 Q6_R_mpy_RRh_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt.L):<<1:rnd:sat`	`Word32 Q6_R_mpy_RRl_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt)`	`Word32 Q6_R_mpy_RR(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt):rnd`	`Word32 Q6_R_mpy_RR_rnd(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs,Rt)`	`Word32 Q6_R_mpyu_RR(Word32 Rs, Word32 Rt)`

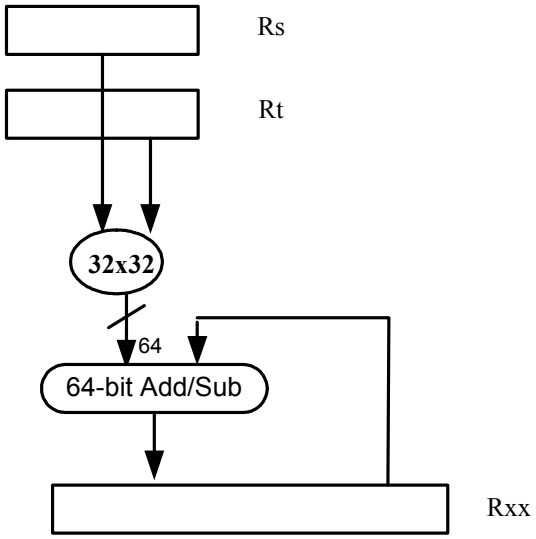
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt)
1	1	1	0	1	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt):rnd
1	1	1	0	1	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs,Rt)
1	1	1	0	1	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.H):<<1:rnd:sat
1	1	1	0	1	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.L):<<1:rnd:sat

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Multiply and use full result

Multiply two signed or unsigned 32-bit words. Optionally add or subtract this value from the 64-bit accumulator. The result is a full-precision 64-bit value.



Syntax	Behavior
`Rdd=mpy(Rs,Rt)`	`Rdd=(Rs * Rt);`
`Rdd=mpyu(Rs,Rt)`	`Rdd=(Rs.uw[0] * Rt.uw[0]);`
`Rxx[+-]=mpy(Rs,Rt)`	`Rxx= Rxx[+-](Rs * Rt);`
`Rxx[+-]=mpyu(Rs,Rt)`	`Rxx= Rxx[+-](Rs.uw[0] * Rt.uw[0]);`

Type: M (slots 2,3)

Intrinsics

`Rdd=mpy(Rs,Rt)`	`Word64 Q6_P_mpy_RR(Word32 Rs, Word32 Rt)`
`Rdd=mpyu(Rs,Rt)`	`Word64 Q6_P_mpyu_RR(Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs,Rt)`	`Word64 Q6_P_mpyacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs,Rt)`	`Word64 Q6_P_mpyuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs,Rt)`	`Word64 Q6_P_mpynac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs,Rt)`	`Word64 Q6_P_mpyunac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs,Rt)
1	1	1	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs,Rt)
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs,Rt)
1	1	1	0	0	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs,Rt)
1	1	1	0	0	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

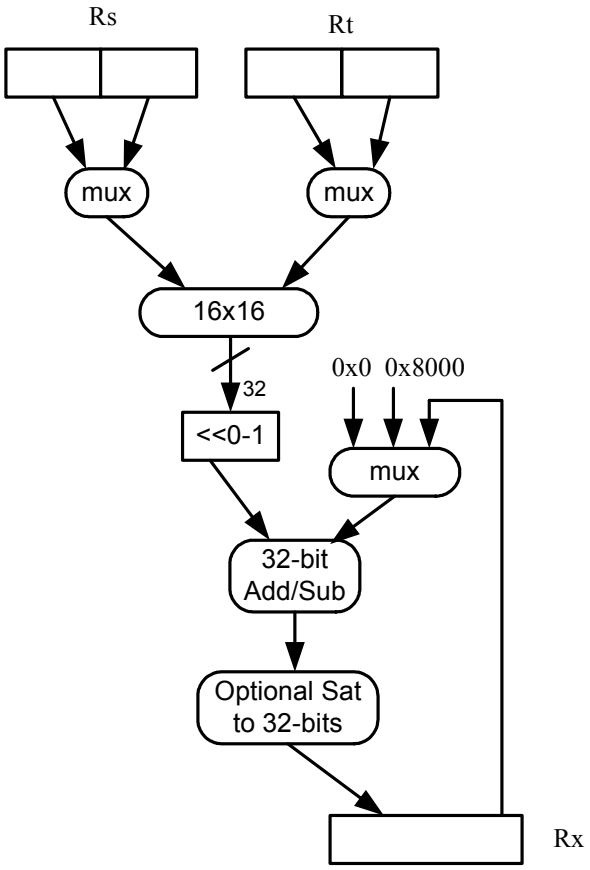
11.10.5 XTYPE / Single Precision Multiply

The XTYPE single-precision multiply instructions perform 16x16 multiplication.

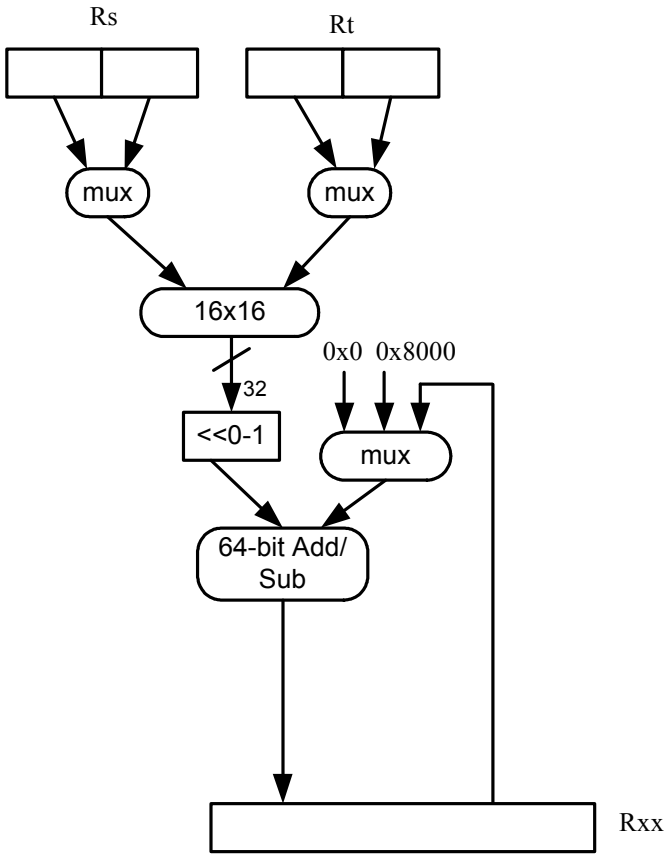
Scalar 16x16 multiply signed

Multiply two signed halfwords. Optionally shift the multiplier result by 1 bit. This result can be accumulated or rounded. The destination/accumulator can be either 32- or 64-bit. For 32-bit results, saturation is optional.

Rx+=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]
Rd = mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd][:sat]



Rxx+=mpy(Rs.[HL],Rt.[HL])[:<<1]
Rdd = mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd]



Syntax	Behavior
`Rd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd][:sat]`	`Rd=[sat_32]([round]((Rs.h[01] * Rt.h[01])[<<1]));`
`Rdd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd]`	`Rdd=[round]((Rs.h[01] * Rt.h[01])[<<1]);`
`Rx+=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]`	`Rx=[sat_32](Rx+ (Rs.h[01] * Rt.h[01])[<<1]);`
`Rx-=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]`	`Rx=[sat_32](Rx- (Rs.h[01] * Rt.h[01])[<<1]);`
`Rxx+=mpy(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx+ (Rs.h[01] * Rt.h[01])[<<1];`
`Rxx-=mpy(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx- (Rs.h[01] * Rt.h[01])[<<1];`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=mpy(Rs.H,Rt.H)`	`Word32 Q6_R_mpy_RhRh(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.H):<<1`	`Word32 Q6_R_mpy_RhRh_s1(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.H):<<1:rnd`	`Word32 Q6_R_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.H):<<1:rnd:sat`	`Word32 Q6_R_mpy_RhRh_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.H):<<1:sat`	`Word32 Q6_R_mpy_RhRh_s1_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.H):rnd`	`Word32 Q6_R_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.H):rnd:sat`	`Word32 Q6_R_mpy_RhRh_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.H):sat`	`Word32 Q6_R_mpy_RhRh_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs.H,Rt.L)`	`Word32 Q6_R_mpy_RhRl(Word32 Rs, Word32 Rt)`

Rd=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpy_RhRl_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RhRl_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpy_RhRl_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):rnd	Word32 Q6_R_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):rnd:sat	Word32 Q6_R_mpy_RhRl_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpy_RhRl_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpy_RlRh(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpy_RlRh_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1:rnd	Word32 Q6_R_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1:rnd:sat	Word32 Q6_R_mpy_RlRh_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpy_RlRh_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):rnd	Word32 Q6_R_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):rnd:sat	Word32 Q6_R_mpy_RlRh_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpy_RlRh_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpy_RlRl(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpy_RlRl_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RlRl_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpy_RlRl_s1_sat(Word32 Rs, Word32 Rt)

Rd=mpy(Rs.L,Rt.L):rnd	Word32 Q6_R_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):rnd:sat	Word32 Q6_R_mpy_RlRl_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpy_RlRl_sat(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpy_RhRh(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpy_RhRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):rnd	Word64 Q6_P_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L)	Word64 Q6_P_mpy_RhRl(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):<<1	Word64 Q6_P_mpy_RhRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):rnd	Word64 Q6_P_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H)	Word64 Q6_P_mpy_RlRh(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):<<1	Word64 Q6_P_mpy_RlRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):rnd	Word64 Q6_P_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L)	Word64 Q6_P_mpy_RlRl(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):<<1	Word64 Q6_P_mpy_RlRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):rnd	Word64 Q6_P_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpyacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)

Rx+=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpyacc_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpyacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpyacc_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpyacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpyacc_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpyacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpyacc_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpynac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpynac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpynac_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpynac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpynac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)

Rx-=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpynac_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpynac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpynac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpynac_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpynac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpynac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpynac_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpyacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.H,Rt.L)	Word64 Q6_P_mpyacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.L,Rt.H)	Word64 Q6_P_mpyacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.L,Rt.L)	Word64 Q6_P_mpyacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpy(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpynac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpynac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)

`Rxx-=mpy(Rs.H,Rt.L)`	`Word64 Q6_P_mpynac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpynac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.H)`	`Word64 Q6_P_mpynac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.H):<<1`	`Word64 Q6_P_mpynac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.L)`	`Word64 Q6_P_mpynac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.L):<<1`	`Word64 Q6_P_mpynac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	d5					
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.L)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.H)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.L)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.H)[:<<N]:rnd
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5					
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	d5					
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:sat

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	
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5					
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.H)[:<<N]:sat

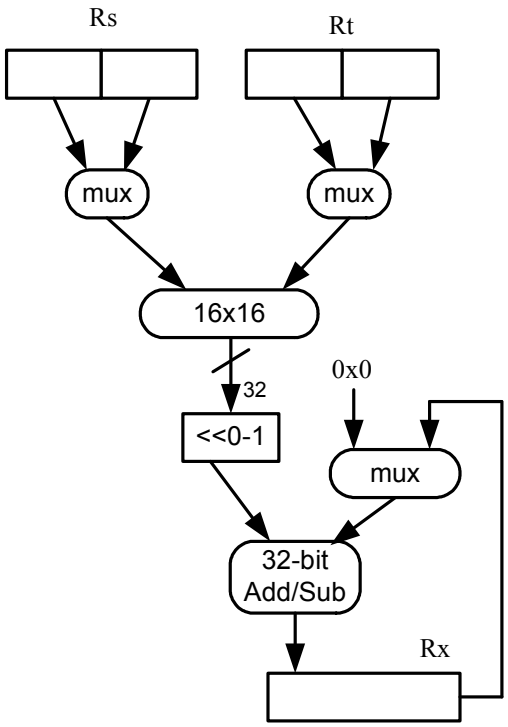
Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
sH	Rs is High
tH	Rt is High
sH	Rs is High

Field name	Description
tH	Rt is High
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

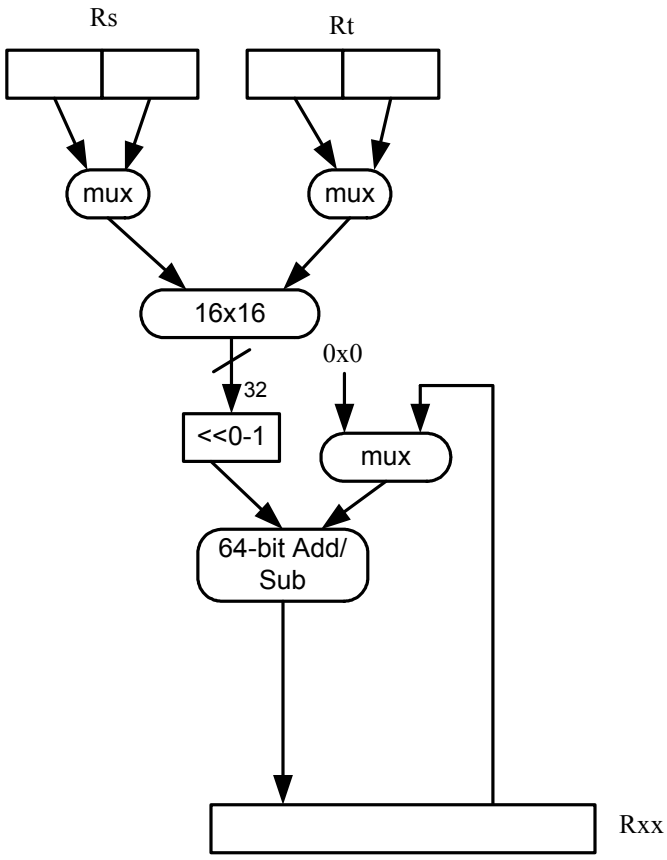
Scalar 16x16 multiply unsigned

Multiply two unsigned halfwords. Scale the result by 0-3 bits. Optionally, add or subtract the result from the accumulator.

Rx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rd = mpyu(Rs.[HL],Rt.[HL])[:<<1]



Rxx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rdd = mpyu(Rs.[HL],Rt.[HL])[:<<1]



Syntax	Behavior
`Rd=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rd=(Rs.uh[01] * Rt.uh[01])[<<1];`

Syntax	Behavior
`Rdd=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rdd=(Rs.uh[01] * Rt.uh[01])[<<1];`
`Rx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rx=Rx+ (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rx-=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rx=Rx- (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rxx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx+ (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rxx-=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx- (Rs.uh[01] * Rt.uh[01])[<<1];`

Type: M (slots 2,3)

Intrinsics

`Rd=mpyu(Rs.H,Rt.H)`	`Word32 Q6_R_mpyu_RhRh(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs.H,Rt.H):<<1`	`Word32 Q6_R_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs.H,Rt.L)`	`Word32 Q6_R_mpyu_RhRl(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs.H,Rt.L):<<1`	`Word32 Q6_R_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs.L,Rt.H)`	`Word32 Q6_R_mpyu_RlRh(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs.L,Rt.H):<<1`	`Word32 Q6_R_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs.L,Rt.L)`	`Word32 Q6_R_mpyu_RlRl(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs.L,Rt.L):<<1`	`Word32 Q6_R_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)`
`Rdd=mpyu(Rs.H,Rt.H)`	`Word64 Q6_P_mpyu_RhRh(Word32 Rs, Word32 Rt)`
`Rdd=mpyu(Rs.H,Rt.H):<<1`	`Word64 Q6_P_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)`
`Rdd=mpyu(Rs.H,Rt.L)`	`Word64 Q6_P_mpyu_RhRl(Word32 Rs, Word32 Rt)`
`Rdd=mpyu(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)`

Rdd=mpyu(Rs.L,Rt.H)	Word64 Q6_P_mpyu_RlRh(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.L)	Word64 Q6_P_mpyu_RlRl(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyuacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyuacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyuacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyuacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.H)	Word32 Q6_R_mpyuacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyuacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.L)	Word32 Q6_R_mpyuacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyuacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyunac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyunac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyunac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyunac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.H)	Word32 Q6_R_mpyunac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyunac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.L)	Word32 Q6_R_mpyunac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyunac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)

Rxx+=mpyu(Rs.H,Rt.H)	Word64 Q6_P_mpyuacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyuacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.H,Rt.L)	Word64 Q6_P_mpyuacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyuacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.H)	Word64 Q6_P_mpyuacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyuacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.L)	Word64 Q6_P_mpyuacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx+=mpyu(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyuacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.H)	Word64 Q6_P_mpyunac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyunac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.L)	Word64 Q6_P_mpyunac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyunac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.H)	Word64 Q6_P_mpyunac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyunac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.L)	Word64 Q6_P_mpyunac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)
Rxx-=mpyu(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyunac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	d5						
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5						
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	d5						
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5						
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.H,Rt.H)[:<<N]

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
sH	Rs is High

Field name	Description
tH	Rt is High
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

11.10.6 XTYPE / Permute

The XTYPE permute instructions perform various permutation operations.

Saturate

Saturate a single scalar value.

`sath` saturates a signed 32-bit number to a signed 16-bit number, which is sign-extended back to 32 bits and placed in the destination register. The minimum negative value of the result is 0xffff8000 and the maximum positive value is 0x00007fff.

`satuh` saturates a signed 32-bit number to an unsigned 16-bit number, which is zero-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0 and the maximum value is 0x0000ffff.

`satub` saturates a signed 32-bit number to an unsigned 8-bit number, which is zero-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0 and the maximum value is 0x000000ff.

`satb` saturates a signed 32-bit number to an signed 8-bit number, which is sign-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0xffffff80 and the maximum value is 0x000000ff.

Syntax	Behavior
`Rd=sat(Rss)`	`Rd = sat_32(Rss);`
`Rd=satb(Rs)`	`Rd = sat_8(Rs);`
`Rd=sath(Rs)`	`Rd = sat_16(Rs);`
`Rd=satub(Rs)`	`Rd = usat_8(Rs);`
`Rd=satuh(Rs)`	`Rd = usat_16(Rs);`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sat(Rss)`	`Word32 Q6_R_sat_P(Word64 Rss)`
`Rd=satb(Rs)`	`Word32 Q6_R_satb_R(Word32 Rs)`
`Rd=sath(Rs)`	`Word32 Q6_R_sath_R(Word32 Rs)`
`Rd=satub(Rs)`	`Word32 Q6_R_satub_R(Word32 Rs)`
`Rd=satuh(Rs)`	`Word32 Q6_R_satuh_R(Word32 Rs)`

Encoding

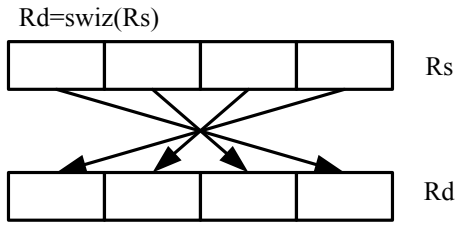
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=sat(Rss)
1	0	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sath(Rs)
1	0	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=satuh(Rs)
1	0	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=satub(Rs)
1	0	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=satb(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Swizzle bytes

Swizzle the bytes of a word.

This instructon is useful in converting between little and big endian formats.



Syntax	Behavior

`Rd=swiz(Rs)`	`Rd.b[0]=Rs.b[3];`
	`Rd.b[1]=Rs.b[2];`
	`Rd.b[2]=Rs.b[1];`
	`Rd.b[3]=Rs.b[0];`

Type: S (slots 2,3)

Intrinsics

`Rd=swiz(Rs)`	`Word32 Q6_R_swiz_R(Word32 Rs)`

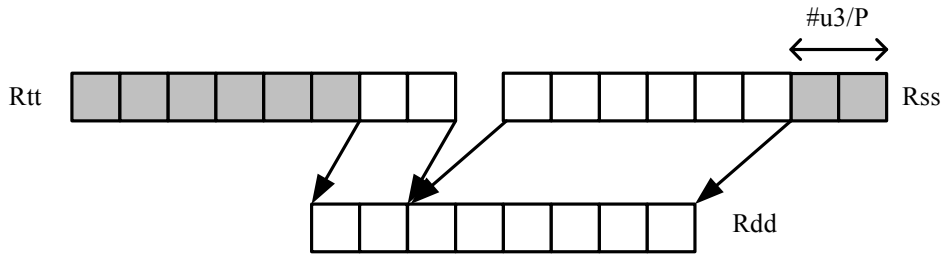
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=swiz(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Vector align

Align a vector. Use the immediate amount, or the least significant 3 bits of a Predicate register, as the number of bytes to align. Shift the Rss register pair right by this number of bytes. Fill the vacated positions with the least significant elements from Rtt.



Syntax	Behavior
`Rdd=valignb(Rtt,Rss,#u3)`	`Rdd = (Rss >>> #u*8)|(Rtt << ((8-#u)*8));`
`Rdd=valignb(Rtt,Rss,Pu)`	`Rdd = Rss >>> (Pu&0x7)*8|(Rtt << (8-(Pu&0x7))*8);`

Type: S (slots 2,3)

Intrinsics

`Rdd=valignb(Rtt,Rss,#u3)`	`Word64 Q6_P_valignb_PPI(Word64 Rtt, Word64 Rss, Word32 Iu)`
`Rdd=valignb(Rtt,Rss,Pu)`	`Word64 Q6_P_valignb_PPp(Word64 Rtt, Word64 Rss, Byte Pu)`

Encoding

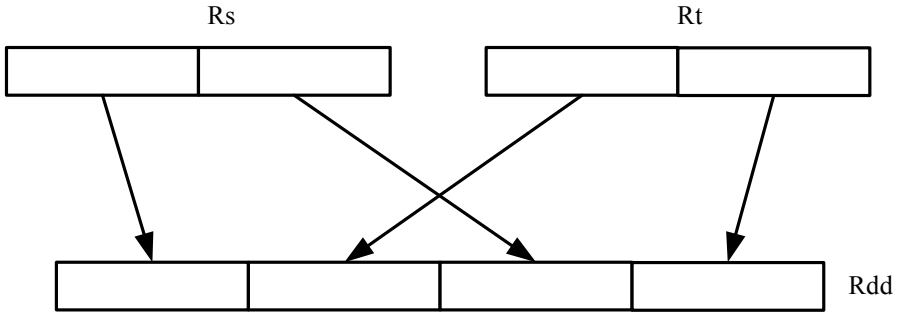
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	0	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	i	i	i	d	d	d	d	d	Rdd=valignb(Rtt,Rss,#u3)
ICLASS				RegType				Maj			s5					Parse			t5						u2		d5					
1	1	0	0	0	0	1	0	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=valignb(Rtt,Rss,Pu)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Field name	Description
u2	Field to encode register u
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector pack high and low halfwords

Pack together the most-significant halfwords from Rs and Rt in the most-significant word of register pair Rdd. The least-significant halfwords from Rs and Rt are packed together in the least-significant halfword of Rdd.



Syntax	Behavior

`Rdd=packhl(Rs,Rt)`	`Rdd.h[0]=Rt.h[0];`
	`Rdd.h[1]=Rs.h[0];`
	`Rdd.h[2]=Rt.h[1];`
	`Rdd.h[3]=Rs.h[1];`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=packhl(Rs,Rt)`	`Word64 Q6_P_packhl_RR(Word32 Rs, Word32 Rt)`

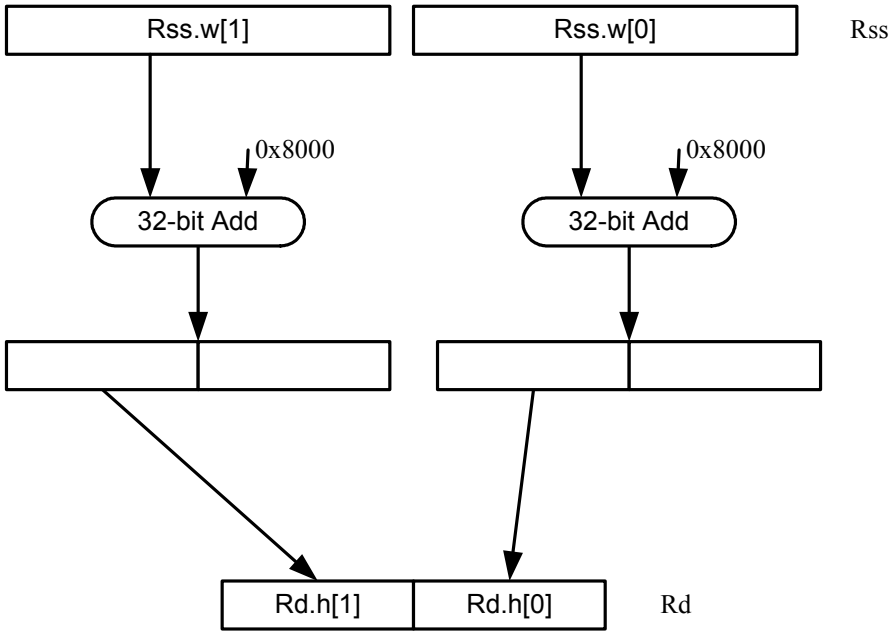
Encoding

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	
ICLASS					RegType						s5					Parse			t5								d5					
1	1	0	1	-	1	0	0	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rdd=packhl(Rs,Rt)

Field name	Description
`RegType`	Register Type
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector round and pack

Add the constant 0x00008000 to each word in the 64-bit source vector Rss. Optionally saturate this addition to 32bits. Pack the high halfwords of the result into the corresponding halfword of the 32-bit destination register.



Syntax	Behavior

`Rd=vrndwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=(Rss.w[i]+0x08000).h[1];`
	`};`

`Rd=vrndwh(Rss):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=sat_32(Rss.w[i]+0x08000).h[1];`
	`};`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vrndwh(Rss)`	`Word32 Q6_R_vrndwh_P(Word64 Rss)`
`Rd=vrndwh(Rss):sat`	`Word32 Q6_R_vrndwh_P_sat(Word64 Rss)`

Encoding

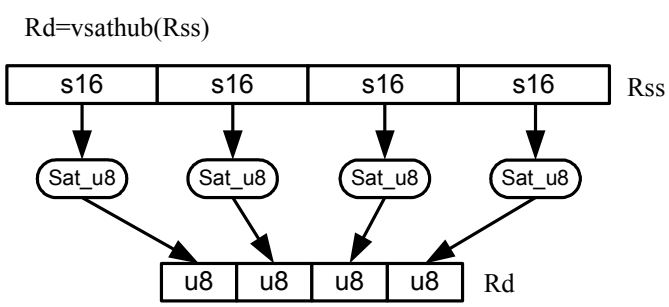
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=vrndwh(Rss)
1	0	0	0	1	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vrndwh(Rss):sat

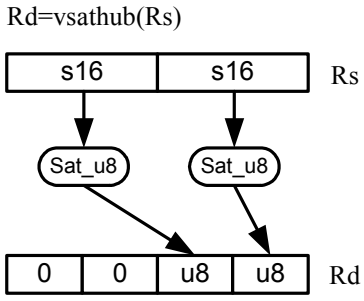
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

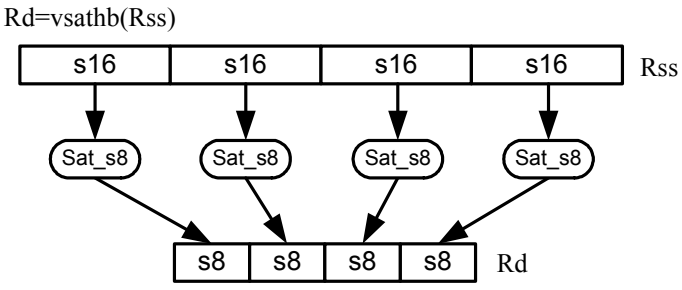
Vector saturate and pack

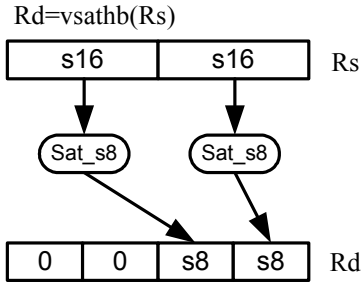
For each element in the vector, saturate the value to the next smaller size.

`vsathub` saturates signed halfwords to unsigned bytes, while `vsathb` saturates signed halfwords to signed bytes.

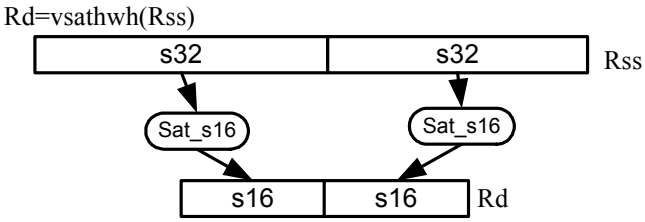


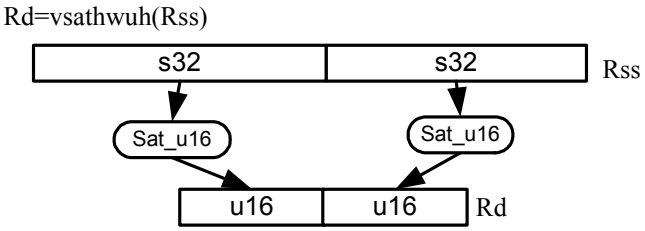






`vsatwh` saturates signed words to signed halfwords, while `vsatwuh` saturates signed words to unsigned halfwords. The resulting values are packed together into destination register Rd.





Syntax	Behavior

`Rd=vsathb(Rs)`	`Rd.b[0]=sat_8(Rs.h[0]);`
	`Rd.b[1]=sat_8(Rs.h[1]);`
	`Rd.b[2]=0;`
	`Rd.b[3]=0;`

`Rd=vsathb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=sat_8(Rss.h[i]);`
	`};`

`Rd=vsathub(Rs)`	`Rd.b[0]=usat_8(Rs.h[0]);`
	`Rd.b[1]=usat_8(Rs.h[1]);`
	`Rd.b[2]=0;`
	`Rd.b[3]=0;`

`Rd=vsathub(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat_8(Rss.h[i]);`
	`};`

`Rd=vsatwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=sat_16(Rss.w[i]);`
	`};`

`Rd=vsatwuh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat_16(Rss.w[i]);`
	`};`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vsathb(Rs)`	`Word32 Q6_R_vsathb_R(Word32 Rs)`
`Rd=vsathb(Rss)`	`Word32 Q6_R_vsathb_P(Word64 Rss)`
`Rd=vsathub(Rs)`	`Word32 Q6_R_vsathub_R(Word32 Rs)`
`Rd=vsathub(Rss)`	`Word32 Q6_R_vsathub_P(Word64 Rss)`
`Rd=vsatwh(Rss)`	`Word32 Q6_R_vsatwh_P(Word64 Rss)`
`Rd=vsatwuh(Rss)`	`Word32 Q6_R_vsatwuh_P(Word64 Rss)`

Encoding

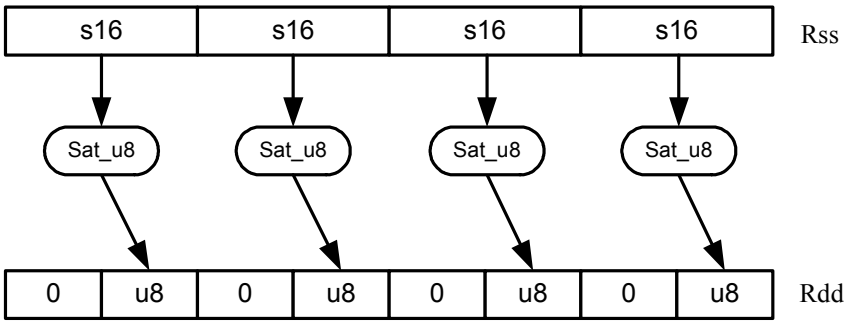
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathub(Rss)
1	0	0	0	1	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vsatwh(Rss)
1	0	0	0	1	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=vsatwuh(Rss)
1	0	0	0	1	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathb(Rss)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathb(Rs)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathub(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

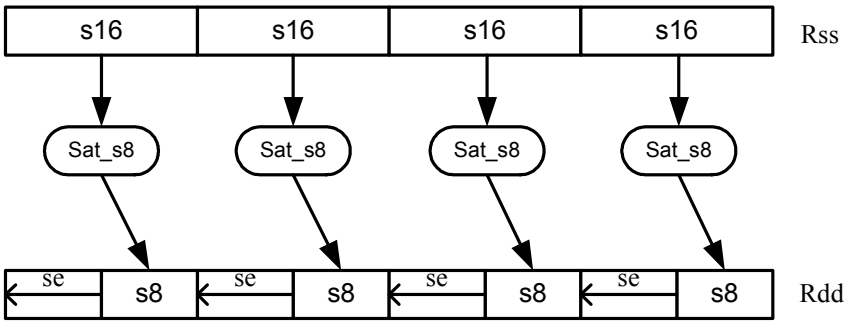
Vector saturate without pack

Saturate each element of source vector Rss to the next smaller size. `vsathub` saturates signed halfwords to unsigned bytes. `vsatwh` saturates signed words to signed halfwords, and `vsatwuh` saturates signed words to unsigned halfwords. The resulting values are placed in destination register Rdd in unpacked form.

Rdd=vsathub(Rss)



Rdd=vsathb(Rss)



Syntax	Behavior

`Rdd=vsathb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_8(Rss.h[i]);`
	`};`

`Rdd=vsathub(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat_8(Rss.h[i]);`
	`};`

`Rdd=vsatwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat_16(Rss.w[i]);`
	`};`

`Rdd=vsatwuh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=usat_16(Rss.w[i]);`
	`};`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsathb(Rss)`	`Word64 Q6_P_vsathb_P(Word64 Rss)`
`Rdd=vsathub(Rss)`	`Word64 Q6_P_vsathub_P(Word64 Rss)`
`Rdd=vsatwh(Rss)`	`Word64 Q6_P_vsatwh_P(Word64 Rss)`
`Rdd=vsatwuh(Rss)`	`Word64 Q6_P_vsatwuh_P(Word64 Rss)`

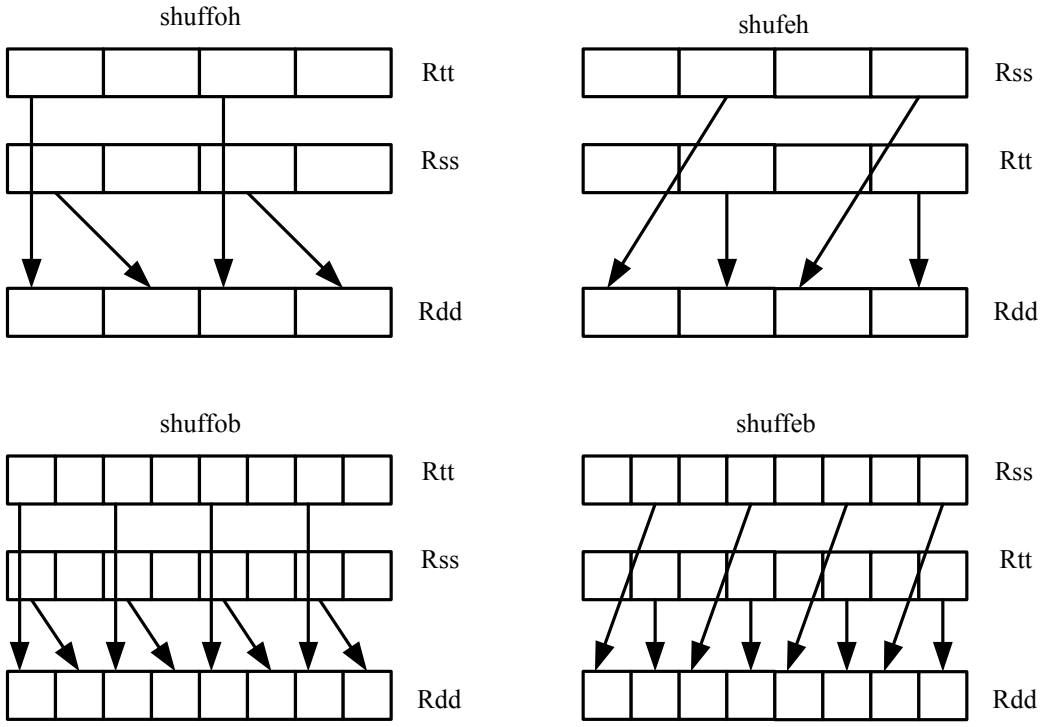
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsathub(Rss)
1	0	0	0	0	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsatwuh(Rss)
1	0	0	0	0	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsatwh(Rss)
1	0	0	0	0	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsathb(Rss)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector shuffle

Shuffle odd halfwords (`shuffoh`) takes the odd halfwords from Rtt and the odd halfwords from Rss and merges them together into vector Rdd. Shuffle even halfwords (`shuffeh`) performs the same operation on every even halfword in Rss and Rtt. The same operation is available for odd and even bytes.



Syntax	Behavior

`Rdd=shuffeb(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i*2]=Rtt.b[i*2];`
	`  Rdd.b[i*2+1]=Rss.b[i*2];`
	`};`

`Rdd=shuffeh(Rss,Rtt)`	`for (i=0;i<2;i++) {`
	`  Rdd.h[i*2]=Rtt.h[i*2];`
	`  Rdd.h[i*2+1]=Rss.h[i*2];`
	`};`

`Rdd=shuffob(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i*2]=Rss.b[i*2+1];`
	`  Rdd.b[i*2+1]=Rtt.b[i*2+1];`
	`};`

`Rdd=shuffoh(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.h[i*2]=Rss.h[i*2+1];`
	`  Rdd.h[i*2+1]=Rtt.h[i*2+1];`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rdd=shuffeb(Rss,Rtt)`	`Word64 Q6_P_shuffeb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=shuffeh(Rss,Rtt)`	`Word64 Q6_P_shuffeh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=shuffob(Rtt,Rss)`	`Word64 Q6_P_shuffob_PP(Word64 Rtt, Word64 Rss)`
`Rdd=shuffoh(Rtt,Rss)`	`Word64 Q6_P_shuffoh_PP(Word64 Rtt, Word64 Rss)`

Encoding

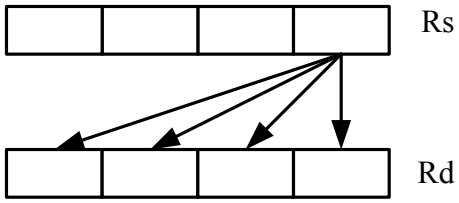
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffeb(Rss,Rtt)
1	1	0	0	0	0	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffob(Rtt,Rss)
1	1	0	0	0	0	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffeh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffoh(Rtt,Rss)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector splat bytes

Vector splat byte replicates the low 8-bits from register Rs into each of the four bytes of destination register Rd.

Rd=vsplatb(Rs)



Syntax	Behavior

`Rd=vsplatb(Rs)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rs.b[0];`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rd=vsplatb(Rs)`	`Word32 Q6_R_vsplatb_R(Word32 Rs)`

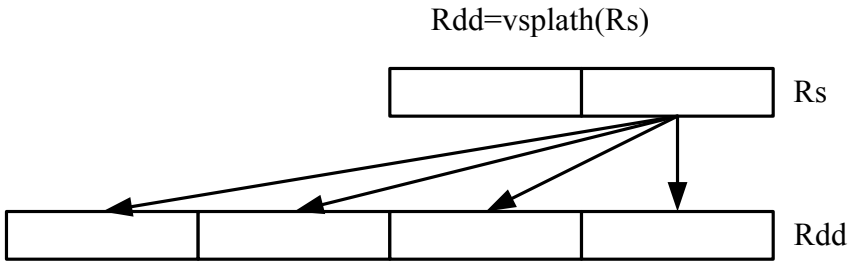
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse							MinOp			d5					
1	0	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vsplatb(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Vector splat halfwords

Vector splat halfword replicates the low 16-bits from register Rs into each of the four halfwords of destination Rdd.



Syntax	Behavior

`Rdd=vsplath(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.h[0];`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rdd=vsplath(Rs)`	`Word64 Q6_P_vsplath_R(Word32 Rs)`

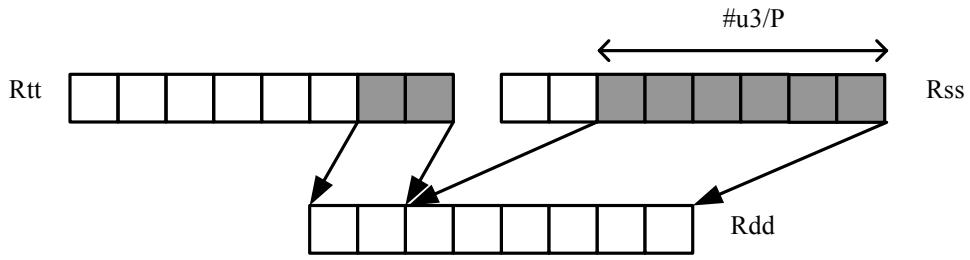
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsplath(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Vector splice

Concatenate the low (8-N) bytes of vector Rtt with the low N bytes of vector Rss. This instruction is useful for vectorizing unaligned stores.



Syntax	Behavior
`Rdd=vspliceb(Rss,Rtt,#u3)`	`Rdd = Rtt << #u*8 | zxt`$_{\#u*8->64}$`(Rss);`
`Rdd=vspliceb(Rss,Rtt,Pu)`	`Rdd = Rtt << (Pu&7)*8 | zxt`$_{(Pu\&7)*8->64}$`(Rss);`

Type: S (slots 2,3)

Intrinsics

`Rdd=vspliceb(Rss,Rtt,#u3)`	`Word64 Q6_P_vspliceb_PPI(Word64 Rss, Word64 Rtt, Word32 Iu)`
`Rdd=vspliceb(Rss,Rtt,Pu)`	`Word64 Q6_P_vspliceb_PPp(Word64 Rss, Word64 Rtt, Byte Pu)`

Encoding

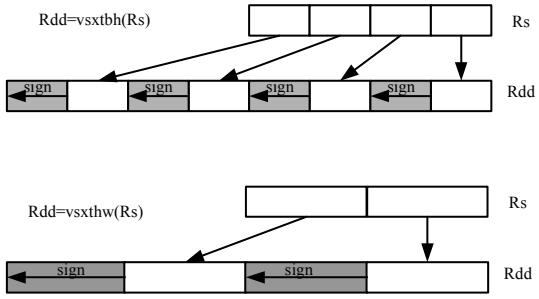
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	0	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	i	i	i	d	d	d	d	d	Rdd=vspliceb(Rss,Rtt,#u3)
ICLASS				RegType				Maj			s5					Parse			t5						u2		d5					
1	1	0	0	0	0	1	0	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=vspliceb(Rss,Rtt,Pu)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Field name	Description
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector sign extend

The first form of this instruction sign-extends each byte of a single register source to halfwords, and places the result in the destination register pair. The second form sign-extends each halfword of a single register source to words, and places the result in the destination register pair.



Syntax	Behavior

`Rdd=vsxtbh(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.b[i];`
	`};`

`Rdd=vsxthw(Rs)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=Rs.h[i];`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rdd=vsxtbh(Rs)`	`Word64 Q6_P_vsxtbh_R(Word32 Rs)`
`Rdd=vsxthw(Rs)`	`Word64 Q6_P_vsxthw_R(Word32 Rs)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsxtbh(Rs)
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsxthw(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

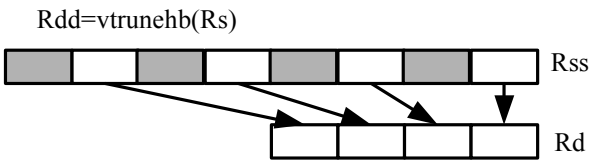
Vector truncate

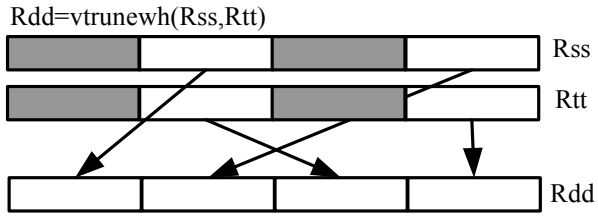
This instruction has two forms: one for truncating byte vectors, and one for truncating halfword vectors.

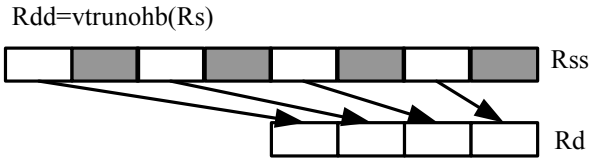
For each halfword in a vector, `vtrunehb` takes the even (lower) byte and ignores the other byte. The resulting values are packed into destination register Rd.

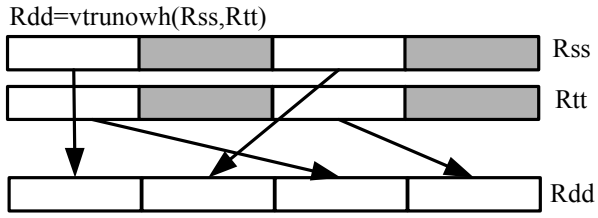
`vtrunohb` takes each odd byte of the source vector. `vtrunewh` uses two source register pairs, Rss and Rtt. The even (lower) halfwords of Rss are packed in the upper word of Rdd, while the lower halfwords of Rtt are packed in the lower word of Rdd.

`vturnowh` performs the same operation, but uses the odd (upper) halfwords of the source vectors instead.









Syntax	Behavior

`Rd=vtrunehb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rss.b[i*2];`
	`};`

`Rd=vtrunohb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rss.b[i*2+1];`
	`};`

`Rdd=vtrunewh(Rss,Rtt)`	`Rdd.h[0]=Rtt.h[0];`
	`Rdd.h[1]=Rtt.h[2];`
	`Rdd.h[2]=Rss.h[0];`
	`Rdd.h[3]=Rss.h[2];`

`Rdd=vtrunowh(Rss,Rtt)`	`Rdd.h[0]=Rtt.h[1];`
	`Rdd.h[1]=Rtt.h[3];`
	`Rdd.h[2]=Rss.h[1];`
	`Rdd.h[3]=Rss.h[3];`

Type: S (slots 2,3)

Intrinsics

`Rd=vtrunehb(Rss)`	`Word32 Q6_R_vtrunehb_P(Word64 Rss)`
`Rd=vtrunohb(Rss)`	`Word32 Q6_R_vtrunohb_P(Word64 Rss)`
`Rdd=vtrunewh(Rss,Rtt)`	`Word64 Q6_P_vtrunewh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vtrunowh(Rss,Rtt)`	`Word64 Q6_P_vtrunowh_PP(Word64 Rss, Word64 Rtt)`

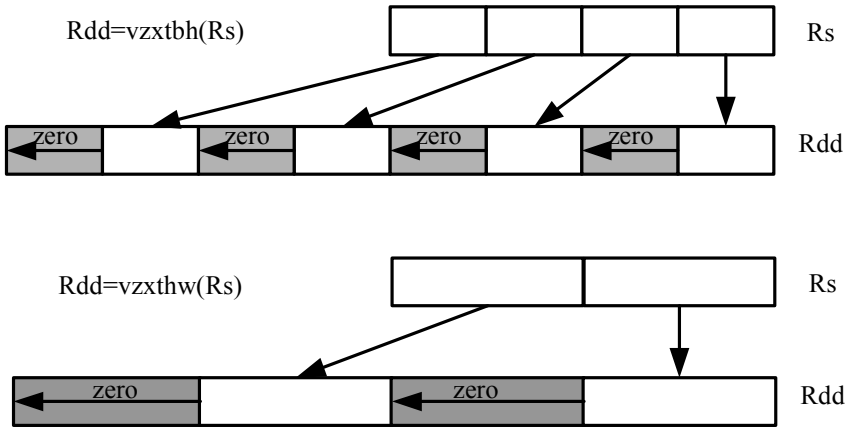
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=vtrunohb(Rss)
1	0	0	0	1	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vtrunehb(Rss)
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunewh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunowh(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

Vector zero extend

This instruction has two forms. The first zero-extends each byte of a single register source to halfwords, and place the result in the destination register pair. The second form zero-extends each halfword of a single register source to words, and places the result in the destination register pair.



Syntax	Behavior

`Rdd=vzxtbh(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.ub[i];`
	`};`

`Rdd=vzxthw(Rs)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=Rs.uh[i];`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rdd=vzxtbh(Rs)`	`Word64 Q6_P_vzxtbh_R(Word32 Rs)`
`Rdd=vzxthw(Rs)`	`Word64 Q6_P_vzxthw_R(Word32 Rs)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vzxtbh(Rs)
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vzxthw(Rs)

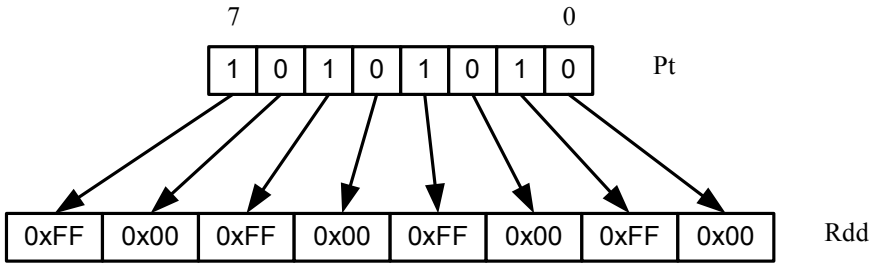
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

11.10.7 XTYPE / Predicate

The XTYPE predicate instructions perform miscellaneous operations on predicates, including mask generation, predicate transfers, and the Viterbi pack operation.

Mask generate from predicate

For each of the low 8 bits in predicate register Pt, if the bit is set then set the corresponding byte in 64-bit register pair Rdd to 0xff, otherwise, set the corresponding byte to 0x00.



Syntax	Behavior

`Rdd=mask(Pt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=(Pt.i?(0xff):(0x00));`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rdd=mask(Pt)`	`Word64 Q6_P_mask_p(Byte Pt)`

Encoding

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	
ICLASS				RegType												Parse						t2					d5					
1	0	0	0	0	1	1	0	-	-	-	-	-	-	-	-	P	P	-	-	-	-	t	t	-	-	-	d	d	d	d	d	Rdd=mask(Pt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
t2	Field to encode register t
RegType	Register Type

Predicate transfer

This instruction has two forms: one transfers a predicate to the low 8 bits of a general register (and zeros the other bits), while the other transfers the low 8 bits of a general register to a predicate.

Syntax	Behavior
`Pd=Rs`	`Pd = Rs.ub[0];`
`Rd=Ps`	`Rd = zxt`$_{8->32}$`(Ps);`

Type: S (slots 2,3)

Intrinsics

`Pd=Rs`	`Byte Q6_p_equals_R(Word32 Rs)`
`Rd=Ps`	`Word32 Q6_R_equals_p(Byte Ps)`

Encoding

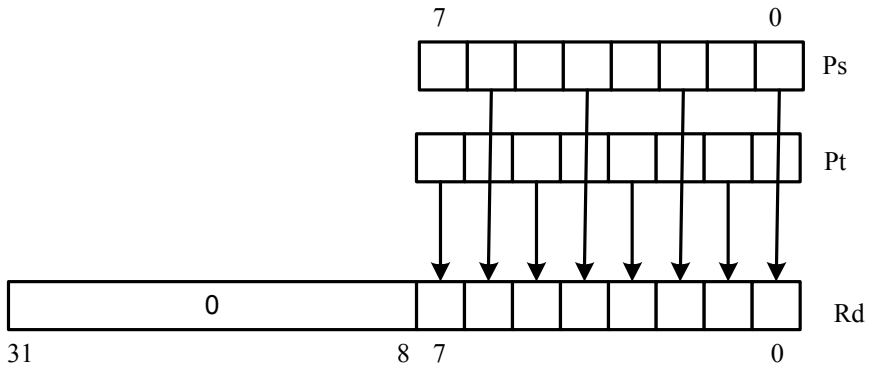
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse														d2		
1	0	0	0	0	1	0	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=Rs
ICLASS				RegType				MajOp						s2		Parse											d5					
1	0	0	0	1	0	0	1	0	1	-	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Ps

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`d5`	Field to encode register d
`s2`	Field to encode register s
`s5`	Field to encode register s
`MajOp`	Major Opcode
`RegType`	Register Type

Viterbi pack even and odd predicate bits

Pack the even and odd bits of two predicate registers into a single destination register.

This instruction is useful in Viterbi decoding.



Syntax	Behavior
`Rd=vitpack(Ps,Pt)`	`Rd = (Ps&0x55) | (Pt&0xAA);`

Type: S (slots 2,3)

Intrinsics

`Rd=vitpack(Ps,Pt)`	`Word32 Q6_R_vitpack_pp(Byte Ps, Byte Pt)`

Encoding

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	
ICLASS				RegType				MajOp						s2		Parse						t2					d5					
1	0	0	0	1	0	0	1	0	0	-	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vitpack(Ps,Pt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s2`	Field to encode register s
`t2`	Field to encode register t
`MajOp`	Major Opcode
`RegType`	Register Type

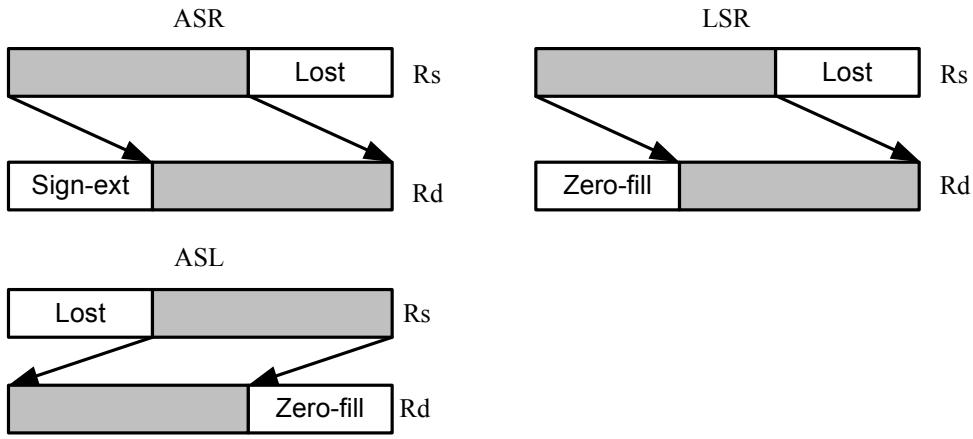
11.10.8 XTYPE / Shift

The XTYPE shift instructions perform various shift operations.

Shift by immediate

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.



Syntax	Behavior
`Rd=asl(Rs,#u5)`	`Rd = Rs << #u;`
`Rd=asr(Rs,#u5)`	`Rd = Rs >> #u;`
`Rd=lsr(Rs,#u5)`	`Rd = Rs >>> #u;`
`Rdd=asl(Rss,#u6)`	`Rdd = Rss << #u;`
`Rdd=asr(Rss,#u6)`	`Rdd = Rss >> #u;`
`Rdd=lsr(Rss,#u6)`	`Rdd = Rss >>> #u;`

Type: S (slots 2,3)

Intrinsics

`Rd=asl(Rs,#u5)`	`Word32 Q6_R_asl_RI(Word32 Rs, Word32 Iu)`

`Rd=asr(Rs,#u5)`	`Word32 Q6_R_asr_RI(Word32 Rs, Word32 Iu)`
`Rd=lsr(Rs,#u5)`	`Word32 Q6_R_lsr_RI(Word32 Rs, Word32 Iu)`
`Rdd=asl(Rss,#u6)`	`Word64 Q6_P_asl_PI(Word64 Rss, Word32 Iu)`
`Rdd=asr(Rss,#u6)`	`Word64 Q6_P_asr_PI(Word64 Rss, Word32 Iu)`
`Rdd=lsr(Rss,#u6)`	`Word64 Q6_P_lsr_PI(Word64 Rss, Word32 Iu)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=lsr(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=asl(Rss,#u6)
1	0	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=lsr(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,#u5)

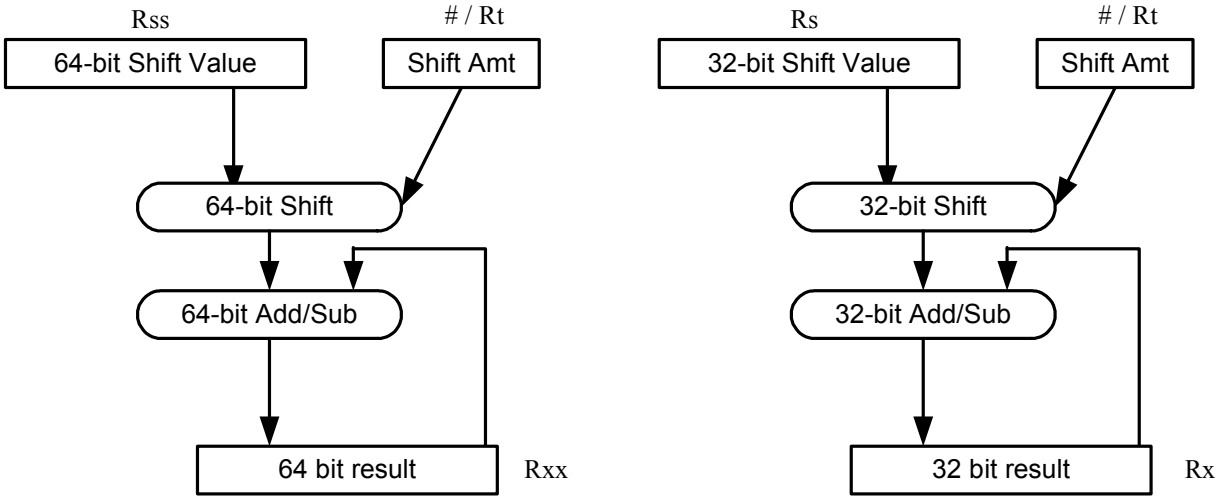
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Shift by immediate and accumulate

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.

After shifting, add or subtract the shifted value from the destination register or register pair.



Syntax	Behavior
`Rx[+-]=asl(Rs,#u5)`	`Rx = Rx [+-] Rs << #u;`
`Rx[+-]=asr(Rs,#u5)`	`Rx = Rx [+-] Rs >> #u;`
`Rx[+-]=lsr(Rs,#u5)`	`Rx = Rx [+-] Rs >>> #u;`
`Rxx[+-]=asl(Rss,#u6)`	`Rxx = Rxx [+-] Rss << #u;`
`Rxx[+-]=asr(Rss,#u6)`	`Rxx = Rxx [+-] Rss >> #u;`
`Rxx[+-]=lsr(Rss,#u6)`	`Rxx = Rxx [+-] Rss >>> #u;`

Type: S (slots 2,3)

Intrinsics

Rx+=asl(Rs,#u5)	Word32 Q6_R_aslacc_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx+=asr(Rs,#u5)	Word32 Q6_R_asracc_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx+=lsr(Rs,#u5)	Word32 Q6_R_lsracc_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx-=asl(Rs,#u5)	Word32 Q6_R_aslnac_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx-=asr(Rs,#u5)	Word32 Q6_R_asrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx-=lsr(Rs,#u5)	Word32 Q6_R_lsrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rxx+=asl(Rss,#u6)	Word64 Q6_P_aslacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu)
Rxx+=asr(Rss,#u6)	Word64 Q6_P_asracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu)
Rxx+=lsr(Rss,#u6)	Word64 Q6_P_lsracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu)
Rxx-=asl(Rss,#u6)	Word64 Q6_P_aslnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu)
Rxx-=asr(Rss,#u6)	Word64 Q6_P_asrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu)
Rxx-=lsr(Rss,#u6)	Word64 Q6_P_lsrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu)

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	x	x	x	x	x	Rxx-=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx-=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx-=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	0	x	x	x	x	x	Rxx+=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	1	x	x	x	x	x	Rxx+=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	0	x	x	x	x	x	Rxx+=asl(Rss,#u6)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	0	x	x	x	x	x	Rx-=asr(Rs,#u5)

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	
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx-=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx-=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	0	x	x	x	x	x	Rx+=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	1	x	x	x	x	x	Rx+=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	0	x	x	x	x	x	Rx+=asl(Rs,#u5)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Shift by immediate and add

Shift Rs left by 0-7 bits, add to Rt, and place the result in Rd.

This instruction is useful for calculating array pointers, where destruction of the base pointer is undesirable.

Syntax	Behavior
`Rd=addasl(Rt,Rs,#u3)`	`Rd = Rt + Rs << #u;`

Type: S (slots 2,3)

Intrinsics

`Rd=addasl(Rt,Rs,#u3)`	`Word32 Q6_R_addasl_RRI(Word32 Rt, Word32 Rs, Word32 Iu)`

Encoding

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	
ICLASS				RegType							s5					Parse			t5					Min			d5					
1	1	0	0	0	1	0	0	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	i	i	i	d	d	d	d	d	Rd=addasl(Rt,Rs,#u3)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Min	Minor Opcode
RegType	Register Type

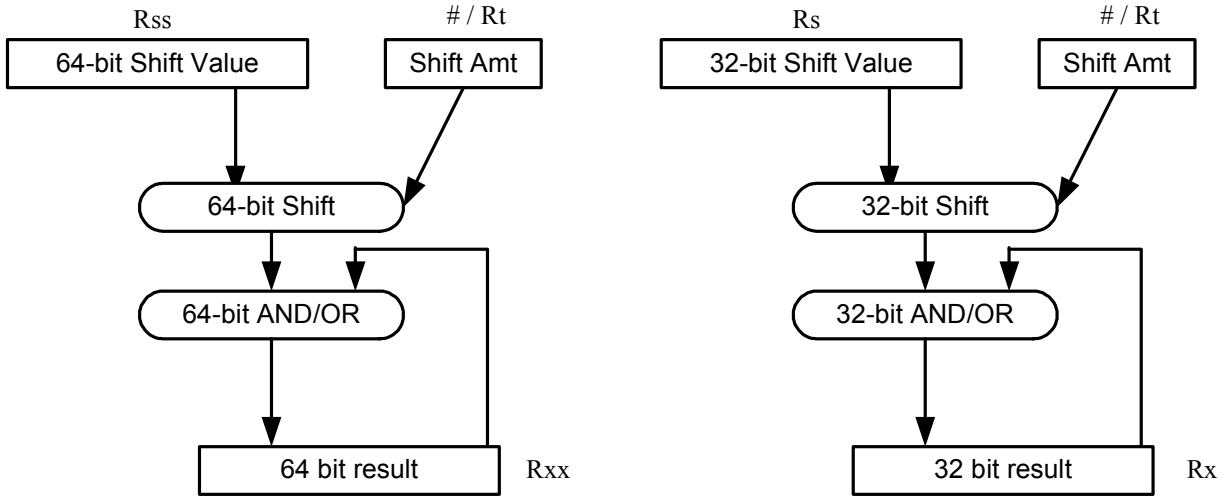
Shift by immediate and logical

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.

After shifting, take the logical AND, OR, or XOR of the shifted amount and the destination register or register pair, and place the result back in the destination register or register pair.

Saturation is not available for these instructions.



Syntax	Behavior
`Rx[&|]=asl(Rs,#u5)`	`Rx = Rx [|&] Rs << #u;`
`Rx[&|]=asr(Rs,#u5)`	`Rx = Rx [|&] Rs >> #u;`
`Rx[&|]=lsr(Rs,#u5)`	`Rx = Rx [|&] Rs >>> #u;`
`Rx^=asl(Rs,#u5)`	`Rx = Rx ^ Rs << #u;`
`Rx^=lsr(Rs,#u5)`	`Rx = Rx ^ Rs >>> #u;`

Syntax	Behavior
Rxx[&|]=asl(Rss,#u6)	Rxx = Rxx [|&] Rss << #u;
Rxx[&|]=asr(Rss,#u6)	Rxx = Rxx [|&] Rss >> #u;
Rxx[&|]=lsr(Rss,#u6)	Rxx = Rxx [|&] Rss >>> #u;
Rxx^=asl(Rss,#u6)	Rxx = Rxx ^ Rss << #u;
Rxx^=lsr(Rss,#u6)	Rxx = Rxx ^ Rss >>> #u;

Type: S (slots 2,3)

Intrinsics

Rx&=asl(Rs,#u5)	Word32 Q6_R_asland_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx&=asr(Rs,#u5)	Word32 Q6_R_asrand_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx&=lsr(Rs,#u5)	Word32 Q6_R_lsrand_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx^=asl(Rs,#u5)	Word32 Q6_R_aslxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx^=lsr(Rs,#u5)	Word32 Q6_R_lsrxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx|=asl(Rs,#u5)	Word32 Q6_R_aslor_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx|=asr(Rs,#u5)	Word32 Q6_R_asror_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rx|=lsr(Rs,#u5)	Word32 Q6_R_lsror_RI(Word32 Rx, Word32 Rs, Word32 Iu)
Rxx&=asl(Rss,#u6)	Word64 Q6_P_asland_PI(Word64 Rxx, Word64 Rss, Word32 Iu)
Rxx&=asr(Rss,#u6)	Word64 Q6_P_asrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu)
Rxx&=lsr(Rss,#u6)	Word64 Q6_P_lsrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu)

`Rxx^=asl(Rss,#u6)`	`Word64 Q6_P_aslxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu)`
`Rxx^=lsr(Rss,#u6)`	`Word64 Q6_P_lsrxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu)`
`Rxx|=asl(Rss,#u6)`	`Word64 Q6_P_aslor_PI(Word64 Rxx, Word64 Rss, Word32 Iu)`
`Rxx|=asr(Rss,#u6)`	`Word64 Q6_P_asror_PI(Word64 Rxx, Word64 Rss, Word32 Iu)`
`Rxx|=lsr(Rss,#u6)`	`Word64 Q6_P_lsror_PI(Word64 Rxx, Word64 Rss, Word32 Iu)`

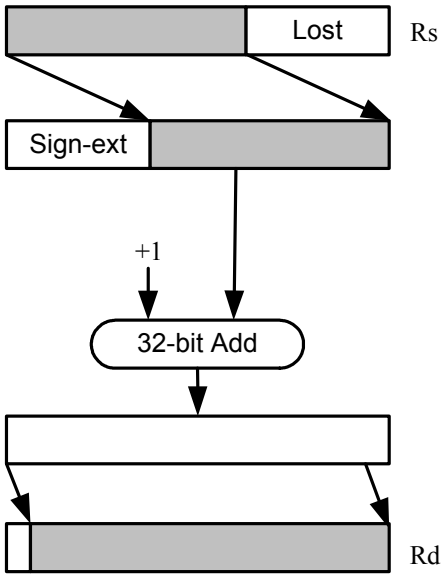
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	x	x	x	x	x	Rxx&=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx&=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx&=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	0	x	x	x	x	x	Rxx|=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	1	x	x	x	x	x	Rxx|=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	0	x	x	x	x	x	Rxx|=asl(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx^=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx^=asl(Rss,#u6)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	0	x	x	x	x	x	Rx&=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx&=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx&=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	0	x	x	x	x	x	Rx|=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	1	x	x	x	x	x	Rx|=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	0	x	x	x	x	x	Rx|=asl(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx^=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx^=asl(Rs,#u5)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Shift right by immediate with rounding

Perform an arithmetic right shift by an immediate amount, and then round the result. This instruction works by first shifting right, then adding the value +1 to the result, and finally shifting right again by one bit. The right shifts always insert the sign-bit in the vacated position.



Syntax	Behavior
`Rd=asr(Rs,#u5):rnd`	`Rd = ((Rs >> #u)+1) >> 1;`
`Rd=asrrnd(Rs,#u5)`	`if ("#u5==0") { Assembler maps to: "Rd=Rs"; } else { Assembler maps to: "Rd=asr(Rs,#u5-1):rnd"; };`

Type: S (slots 2,3)

Intrinsics

`Rd=asr(Rs,#u5):rnd`	`Word32 Q6_R_asr_RI_rnd(Word32 Rs, Word32 Iu)`
`Rd=asrrnd(Rs,#u5)`	`Word32 Q6_R_asrrnd_RI(Word32 Rs, Word32 Iu)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	1	0	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	0	d	d	d	d	d	Rd=asr(Rs,#u5):rnd

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Shift left by immediate with saturation

Perform a left shift of the 32-bit source register value by an immediate amount and saturate. Saturation works by first sign-extending the 32-bit Rs register to 64 bits. It is then left-shifted by the immediate amount. If this 64-bit value cannot fit in a signed 32-bit number (the upper word is not the sign-extension of bit 31), then saturation is performed based on the sign of the original value. Saturation clamps the 32-bit result to the range 0x80000000 to 0x7fffffff.

Syntax	Behavior
`Rd=asl(Rs,#u5):sat`	`Rd = sat_32(sxt`$_{32->64}$`(Rs) << #u);`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=asl(Rs,#u5):sat`	`Word32 Q6_R_asl_RI_sat(Word32 Rs, Word32 Iu)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	1	0	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	d	d	d	d	d	Rd=asl(Rs,#u5):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

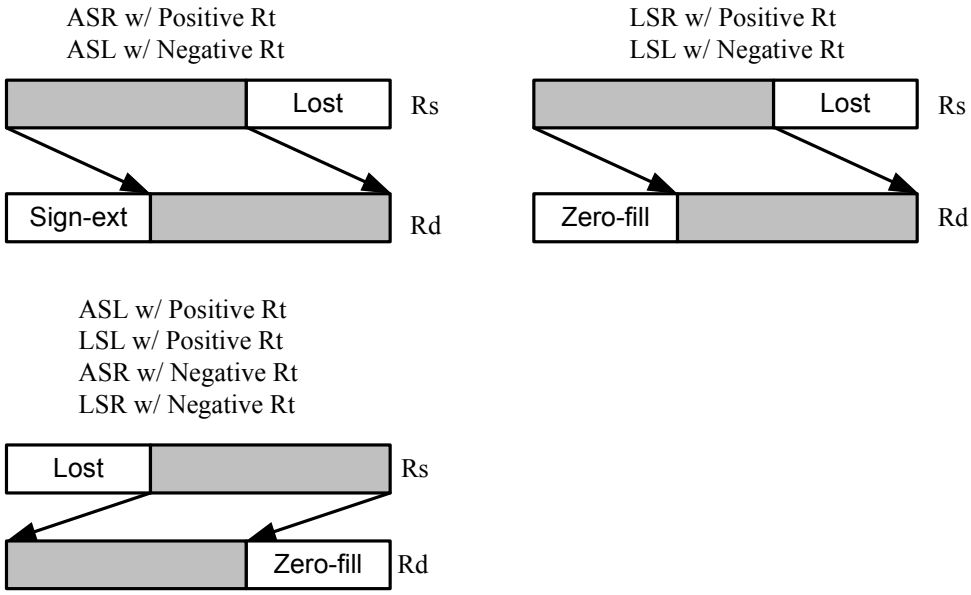
Shift by register

Perform shift by register value.

The shift amount is the least significant 7 bits of Rt, treated as a two's-complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicted in the opcode is reversed (see the figure below).

The source data to be shifted is always performed as a 64-bit shift. When the Rs source register is a 32-bit register, this register is first sign or zero-extended to 64 bits. Arithmetic shifts sign-extend the 32-bit source to 64 bits, while logical shifts zero extend.

The 64-bit source value is then right or left shifted based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
Rd=asl(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rd = (shamt>0)?(sxt$_{32->64}$(Rs)<<shamt):(sxt$_{32->64}$(Rs)>>shamt);
Rd=asr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rd = (shamt>0)?(sxt$_{32->64}$(Rs)>>shamt):(sxt$_{32->64}$(Rs)<<shamt);
Rd=lsl(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rd = (shamt>0)?(zxt$_{32->64}$(Rs)<<shamt):(zxt$_{32->64}$(Rs)>>>shamt);
Rd=lsr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rd = (shamt>0)?(zxt$_{32->64}$(Rs)>>>shamt):(zxt$_{32->64}$(Rs)<<shamt);
Rdd=asl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss<<shamt):(Rss>>shamt);
Rdd=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss>>shamt):(Rss<<shamt);
Rdd=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss<<shamt):(Rss>>>shamt);
Rdd=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss>>>shamt):(Rss<<shamt);

Type: S (slots 2,3)

Intrinsics

Rd=asl(Rs,Rt)	Word32 Q6_R_asl_RR(Word32 Rs, Word32 Rt)
Rd=asr(Rs,Rt)	Word32 Q6_R_asr_RR(Word32 Rs, Word32 Rt)
Rd=lsl(Rs,Rt)	Word32 Q6_R_lsl_RR(Word32 Rs, Word32 Rt)
Rd=lsr(Rs,Rt)	Word32 Q6_R_lsr_RR(Word32 Rs, Word32 Rt)
Rdd=asl(Rss,Rt)	Word64 Q6_P_asl_PR(Word64 Rss, Word32 Rt)
Rdd=asr(Rss,Rt)	Word64 Q6_P_asr_PR(Word64 Rss, Word32 Rt)
Rdd=lsl(Rss,Rt)	Word64 Q6_P_lsl_PR(Word64 Rss, Word32 Rt)
Rdd=lsr(Rss,Rt)	Word64 Q6_P_lsr_PR(Word64 Rss, Word32 Rt)

Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=lsr(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=asl(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=lsl(Rss,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=lsr(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=lsl(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

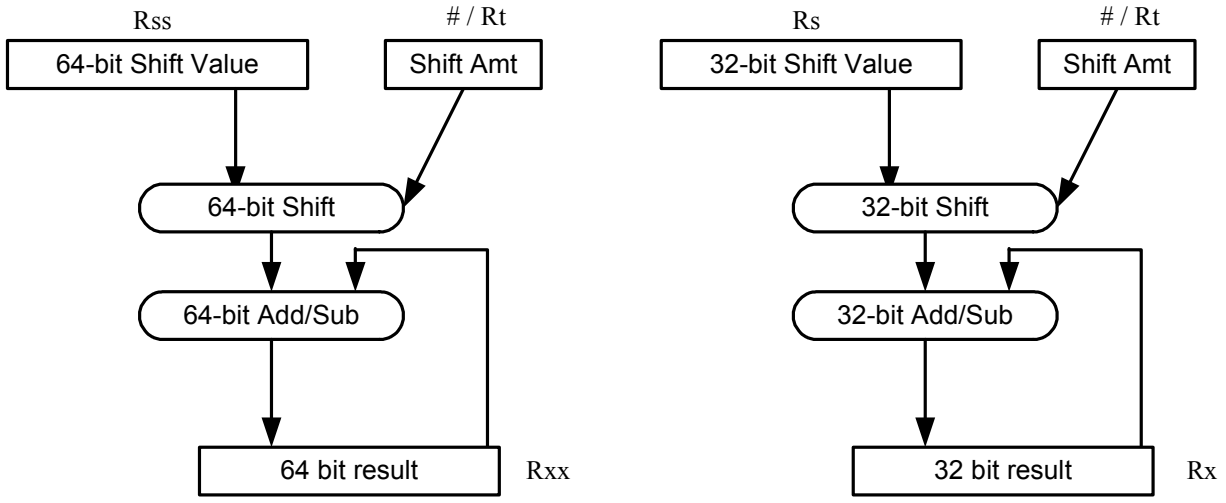
Shift by register and accumulate

The shift amount is the least significant 7 bits of Rt, treated as a two's-complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicted in the opcode is reversed.

Shift the source register value right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.

The shift operation is always performed as a 64-bit shift. When the Rs source register is a 32-bit register, this register is first sign- or zero-extended to 64 bits. Arithmetic shifts sign-extend the 32-bit source to 64 bits, while logical shifts zero-extend.

After shifting, add or subtract the 64-bit shifted amount from the destination register or register pair.



Syntax	Behavior
Rx[+-]=asl(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [+-] (shamt>0)?(sxt$_{32->64}$(Rs)<<shamt):(sxt$_{32->64}$(Rs)>>shamt);
Rx[+-]=asr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [+-] (shamt>0)?(sxt$_{32->64}$(Rs)>>shamt):(sxt$_{32->64}$(Rs)<<shamt);
Rx[+-]=lsl(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [+-] (shamt>0)?(zxt$_{32->64}$(Rs)<<shamt):(zxt$_{32->64}$(Rs)>>>shamt);
Rx[+-]=lsr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [+-] (shamt>0)?(zxt$_{32->64}$(Rs)>>>shamt):(zxt$_{32->64}$(Rs)<<shamt);
Rxx[+-]=asl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss<<shamt):(Rss>>shamt);
Rxx[+-]=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss>>shamt):(Rss<<shamt);
Rxx[+-]=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss<<shamt):(Rss>>>shamt);
Rxx[+-]=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss>>>shamt):(Rss<<shamt);

Type: S (slots 2,3)

Intrinsics

Rx+=asl(Rs,Rt)	Word32 Q6_R_aslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=asr(Rs,Rt)	Word32 Q6_R_asracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=lsl(Rs,Rt)	Word32 Q6_R_lslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=lsr(Rs,Rt)	Word32 Q6_R_lsracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=asl(Rs,Rt)	Word32 Q6_R_aslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)

`Rx-=asr(Rs,Rt)`	`Word32 Q6_R_asrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=lsl(Rs,Rt)`	`Word32 Q6_R_lslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=lsr(Rs,Rt)`	`Word32 Q6_R_lsrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rxx+=asl(Rss,Rt)`	`Word64 Q6_P_aslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx+=asr(Rss,Rt)`	`Word64 Q6_P_asracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx+=lsl(Rss,Rt)`	`Word64 Q6_P_lslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx+=lsr(Rss,Rt)`	`Word64 Q6_P_lsracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=asl(Rss,Rt)`	`Word64 Q6_P_aslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=asr(Rss,Rt)`	`Word64 Q6_P_asrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=lsl(Rss,Rt)`	`Word64 Q6_P_lslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=lsr(Rss,Rt)`	`Word64 Q6_P_lsrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`

Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx-=asr(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx-=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx-=asl(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx-=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx+=asr(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx+=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx+=asl(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx+=lsl(Rss,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx-=asr(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx-=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx-=asl(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx-=lsl(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx+=asr(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx+=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx+=asl(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx+=lsl(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Shift by register and logical

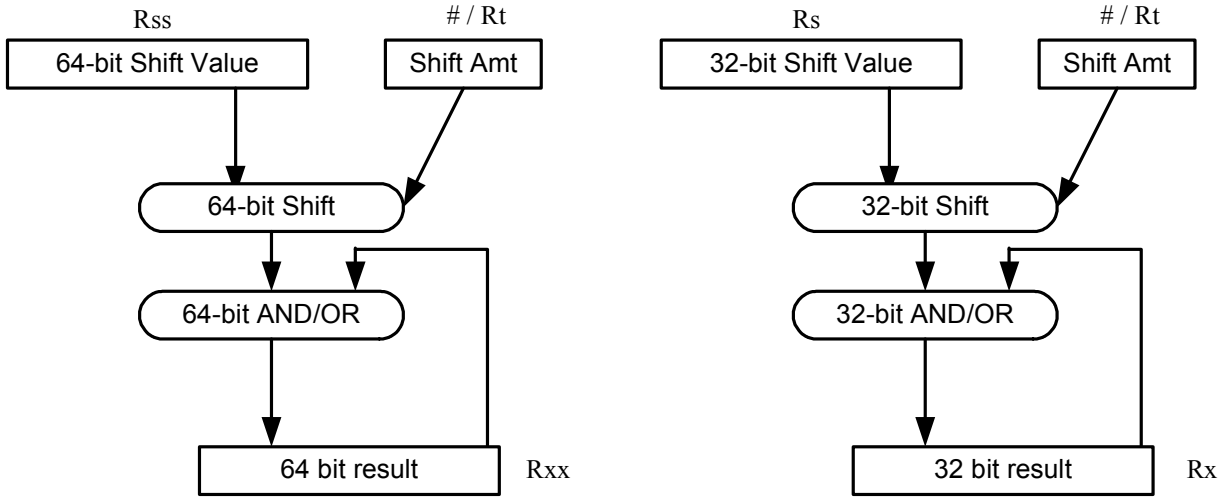
The shift amount is the least significant 7 bits of Rt, treated as a two's-complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicated in the opcode is reversed.

Shift the source register value right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.

The shift operation is always performed as a 64-bit shift. When the Rs source register is a 32-bit register, this register is first sign- or zero-extended to 64 bits. Arithmetic shifts sign-extend the 32-bit source to 64 bits, while logical shifts zero-extend.

After shifting, take the logical AND or OR of the shifted amount and the destination register or register pair, and place the result back in the destination register or register pair.

Saturation is not available for these instructions.



Syntax	Behavior
Rx[&|]=asl(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [|&] (shamt>0)?(sxt$_{32->64}$(Rs)<<shamt):(sxt$_{32->64}$(Rs)>>shamt);
Rx[&|]=asr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [|&] (shamt>0)?(sxt$_{32->64}$(Rs)>>shamt):(sxt$_{32->64}$(Rs)<<shamt);
Rx[&|]=lsl(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [|&] (shamt>0)?(zxt$_{32->64}$(Rs)<<shamt):(zxt$_{32->64}$(Rs)>>>shamt);
Rx[&|]=lsr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [|&] (shamt>0)?(zxt$_{32->64}$(Rs)>>>shamt):(zxt$_{32->64}$(Rs)<<shamt);
Rxx[&|]=asl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss<<shamt):(Rss>>shamt);
Rxx[&|]=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss>>shamt):(Rss<<shamt);
Rxx[&|]=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss<<shamt):(Rss>>>shamt);
Rxx[&|]=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss>>>shamt):(Rss<<shamt);

Type: S (slots 2,3)

Intrinsics

Rx&=asl(Rs,Rt)	Word32 Q6_R_asland_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=asr(Rs,Rt)	Word32 Q6_R_asrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=lsl(Rs,Rt)	Word32 Q6_R_lsland_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=lsr(Rs,Rt)	Word32 Q6_R_lsrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=asl(Rs,Rt)	Word32 Q6_R_aslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)

`Rx|=asr(Rs,Rt)`	`Word32 Q6_R_asror_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=lsl(Rs,Rt)`	`Word32 Q6_R_lslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=lsr(Rs,Rt)`	`Word32 Q6_R_lsror_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rxx&=asl(Rss,Rt)`	`Word64 Q6_P_asland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx&=asr(Rss,Rt)`	`Word64 Q6_P_asrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx&=lsl(Rss,Rt)`	`Word64 Q6_P_lsland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx&=lsr(Rss,Rt)`	`Word64 Q6_P_lsrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=asl(Rss,Rt)`	`Word64 Q6_P_aslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=asr(Rss,Rt)`	`Word64 Q6_P_asror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=lsl(Rss,Rt)`	`Word64 Q6_P_lslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx|=lsr(Rss,Rt)`	`Word64 Q6_P_lsror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`

Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx|=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx|=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx|=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx|=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx&=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx&=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx&=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx&=lsl(Rss,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx|=asr(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx|=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx|=asl(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx|=lsl(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx&=asr(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx&=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx&=asl(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx&=lsl(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Shift by register with saturation

The shift amount is the least significant 7 bits of Rt, treated as a two's-complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicted in the opcode is reversed.

Saturation is available for 32-bit arithmetic left shifts. This can be either an ASL instruction with positive Rt, or an ASR instruction with negative Rt. Saturation works by first sign-extending the 32-bit Rs register to 64 bits. It is then shifted by the shift amount. If this 64-bit value cannot fit in a signed 32-bit number (the upper word is not the sign-extension of bit 31), then saturation is performed based on the sign of the original value. Saturation clamps the 32-bit result to the range 0x80000000 to 0x7fffffff.

Syntax	Behavior
`Rd=asl(Rs,Rt):sat`	`shamt=sxt`$_{7->32}$`(Rt); Rd = sat_32((shamt>0)?(sxt`$_{32->64}$`(Rs)<<shamt):(sxt`$_{32->64}$`(Rs)>>shamt));`
`Rd=asr(Rs,Rt):sat`	`shamt=sxt`$_{7->32}$`(Rt); Rd = sat_32((shamt>0)?(sxt`$_{32->64}$`(Rs)>>shamt):(sxt`$_{32->64}$`(Rs)<<shamt));`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=asl(Rs,Rt):sat`	`Word32 Q6_R_asl_RR_sat(Word32 Rs, Word32 Rt)`
`Rd=asr(Rs,Rt):sat`	`Word32 Q6_R_asr_RR_sat(Word32 Rs, Word32 Rt)`

Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	1	1	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,Rt):sat
1	1	0	0	0	1	1	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,Rt):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Table index

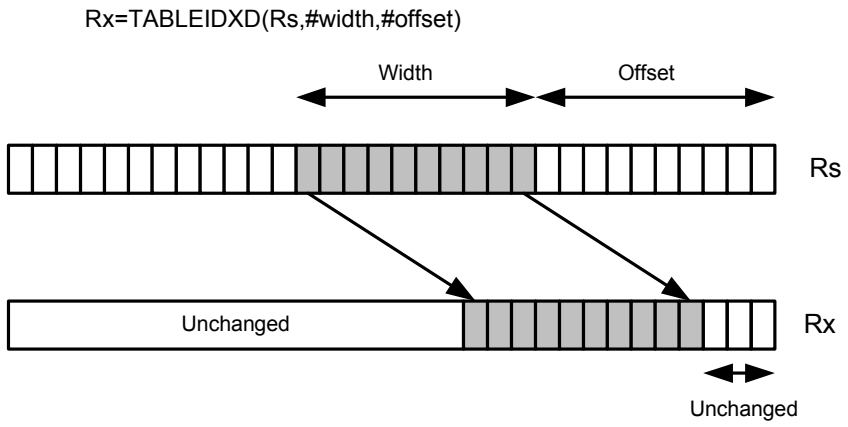
Support fast lookup tables.

Tables are defined to contain entries of bytes, halfwords, words, or doublewords. The table must be aligned to a power-of-2 size greater than or equal to the table size. For example, a 4Kbyte table should be aligned to a 4Kbyte boundary. This instruction supports a maximum of 32K entry tables.

Register Rx contains a pointer into the table. Register Rs contains a field to be extracted and used as a table index. This instruction first extracts the field from register Rs and then inserts it into register Rx. The insertion point is bit 0 for tables of bytes, bit 1 for tables of halfwords, bit 2 for tables of words, and bit 3 for tables of doublewords.

In the assembly syntax, the width and offset values represent the field in Rs to be extracted. Unsigned constants should be used to specify the width and offsets in assembly. In the encoded instruction, however, these values are adjusted by the assembler as follows:

- For `tableidxb`, no adjustment is necessary.
- For `tableidxh`, the assembler encodes offset-1 in the signed immediate field.
- For `tableidxw`, the assembler encodes offset-2 in the signed immediate field.
- For `tableidxd`, the assembler encodes offset-3 in the signed immediate field.



Syntax	Behavior

Rx=tableidxb(Rs,#u4,#S6):raw	width=#u;
	offset=#S;
	field = Rs[(width+offset):offset];
	Rx[(width+0):0]=field;

Rx=tableidxb(Rs,#u4,#U5)	Assembler maps to: "Rx=tableidxb(Rs,#u4,#U5):raw"

Rx=tableidxd(Rs,#u4,#S6):raw	width=#u;
	offset=#S+3;
	field = Rs[(width+offset):offset];
	Rx[(width+3):3]=field;

Rx=tableidxd(Rs,#u4,#U5)	Assembler maps to: "Rx=tableidxd(Rs,#u4,#U5-3):raw"

Rx=tableidxh(Rs,#u4,#S6):raw	width=#u;
	offset=#S+1;
	field = Rs[(width+offset):offset];
	Rx[(width+1):1]=field;

Rx=tableidxh(Rs,#u4,#U5)	Assembler maps to: "Rx=tableidxh(Rs,#u4,#U5-1):raw"

Rx=tableidxw(Rs,#u4,#S6):raw	width=#u;
	offset=#S+2;
	field = Rs[(width+offset):offset];
	Rx[(width+2):2]=field;

Rx=tableidxw(Rs,#u4,#U5)	Assembler maps to: "Rx=tableidxw(Rs,#u4,#U5-2):raw"

Type: S (slots 2,3)

Intrinsics

Rx=tableidxb(Rs,#u4,#U5)	Word32 Q6_R_tableidxb_RII(Word32 Rx, Word32 Rs, Word32 Iu, Word32 IU)
Rx=tableidxd(Rs,#u4,#U5)	Word32 Q6_R_tableidxd_RII(Word32 Rx, Word32 Rs, Word32 Iu, Word32 IU)
Rx=tableidxh(Rs,#u4,#U5)	Word32 Q6_R_tableidxh_RII(Word32 Rx, Word32 Rs, Word32 Iu, Word32 IU)
Rx=tableidxw(Rs,#u4,#U5)	Word32 Q6_R_tableidxw_RII(Word32 Rx, Word32 Rs, Word32 Iu, Word32 IU)

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	1	1	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx=tableidxb(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx=tableidxh(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx=tableidxw(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	1	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx=tableidxd(Rs,#u4,#S6):raw

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

11.10.9 XTYPE / Vector Byte

The XTYPE vector byte instructions perform vector operations on byte values.

Vector add unsigned bytes

Add each of the eight bytes in 64-bit vector Rss to the corresponding byte in vector Rtt. Optionally saturate each 8-bit addition to an unsigned value between 0 and 255. The eight results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vaddub(Rss,Rtt)[:sat]`	`for (i = 0; i < 8; i++) { Rdd.b[i]=[usat_8](Rss.ub[i]+Rtt.ub[i]) ; };`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddub(Rss,Rtt)`	`Word64 Q6_P_vaddub_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddub(Rss,Rtt):sat`	`Word64 Q6_P_vaddub_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	0	d	d	d	d	d	Rdd=vaddub(Rss,Rtt)
1	1	0	1	-	0	1	1	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	1	d	d	d	d	d	Rdd=vaddub(Rss,Rtt):sat

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector average unsigned bytes

Average each of the eight unsigned bytes in the 64-bit source vector Rss with the corresponding byte in Rtt. The average operation performed on each byte is the sum of the two bytes shifted right by 1 bit. If the round option is used, then 0x01 is also added to each result before shifting. This operation does not overflow. In the case that a summation (before right shift by 1) causes an overflow of 8 bits, the value shifted in is the most-significant carry out.

Syntax	Behavior

`Rdd=vavgub(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=((Rss.ub[i] + Rtt.ub[i])>>1);`
	`};`

`Rdd=vavgub(Rss,Rtt):rnd`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=((Rss.ub[i]+Rtt.ub[i]+1)>>1);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vavgub(Rss,Rtt)`	`Word64 Q6_P_vavgub_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgub(Rss,Rtt):rnd`	`Word64 Q6_P_vavgub_PP_rnd(Word64 Rss, Word64 Rtt)`

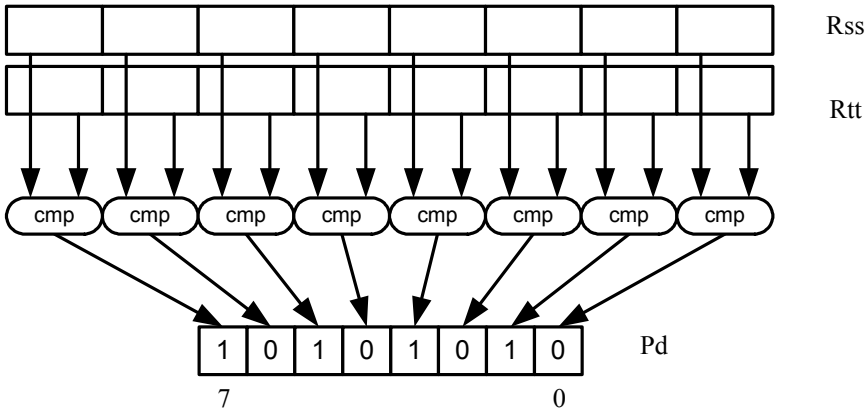
Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgub(Rss,Rtt)
1	1	0	1	-	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavgub(Rss,Rtt):rnd

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector compare unsigned bytes

Compare each of eight bytes in two 64-bit vectors and set the corresponding bit in a predicate destination to 1 if true, 0 if false. Byte comparisons can be for equal or for unsigned greater than. In the following example, every other comparison is true.



Syntax	Behavior

`Pd=vcmpb.eq(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.b[i] == Rtt.b[i]);`
	`};`

`Pd=vcmpb.gtu(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.ub[i] > Rtt.ub[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Pd=vcmpb.eq(Rss,Rtt)`	`Byte Q6_p_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmpb.gtu(Rss,Rtt)`	`Byte Q6_p_vcmpb_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp						d2		
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	-	-	-	**d**	**d**	Pd=vcmpb.eq(Rss,Rtt)
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	-	-	-	**d**	**d**	Pd=vcmpb.gtu(Rss,Rtt)

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector maximum unsigned bytes

Compare each of the eight unsigned bytes in the 64-bit source vector Rss to the corresponding byte in Rtt. For each comparison, select the maximum of the two bytes and place that byte in the corresponding location in Rdd.

Syntax	Behavior

`Rdd=vmaxub(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=max(Rss.ub[i],Rtt.ub[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vmaxub(Rss,Rtt)`	`Word64 Q6_P_vmaxub_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxub(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector minimum unsigned bytes

Compare each of the eight unsigned bytes in the 64-bit source vector Rss to the corresponding byte in Rtt. For each comparison, select the minimum of the two bytes and place that byte in the corresponding location in Rdd.

Syntax	Behavior

`Rdd=vminub(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=min(Rtt.ub[i],Rss.ub[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vminub(Rtt,Rss)`	`Word64 Q6_P_vminub_PP(Word64 Rtt, Word64 Rss)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5						MinOp			d5					
1	1	0	1	-	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminub(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector subtract unsigned bytes

Subtract each of the eight bytes in 64-bit vector Rss from the corresponding byte in vector Rtt. Optionally, saturate each 8-bit subtraction to an unsigned value between 0 and 255. The eight results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vsubub(Rtt,Rss)[:sat]`	`for (i = 0; i < 8; i++) { Rdd.b[i]=[usat_8](Rtt.ub[i]-Rss.ub[i]); };`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubub(Rtt,Rss)`	`Word64 Q6_P_vsubub_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubub(Rtt,Rss):sat`	`Word64 Q6_P_vsubub_PP_sat(Word64 Rtt, Word64 Rss)`

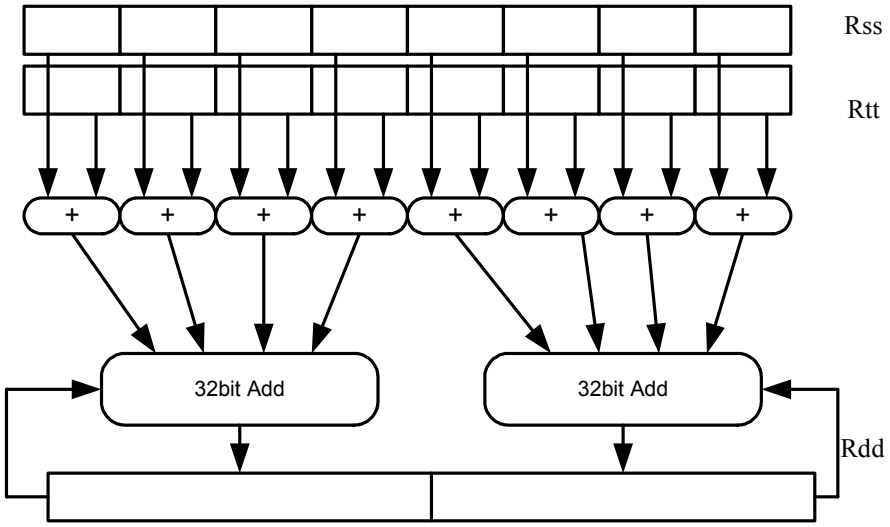
Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubub(Rtt,Rss)
1	1	0	1	-	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubub(Rtt,Rss):sat

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector reduce add unsigned bytes

For each byte in the source vector Rss, add the corresponding byte in the source vector Rtt. Add the four upper intermediate results and optionally the upper word of the destination. Add the four lower results and optionally the lower word of the destination.



Syntax	Behavior

`Rdd=vraddub(Rss,Rtt)`	`Rdd = 0;`
	`for (i=0;i<4;i++) {`
	`  Rdd.w[0]=(Rdd.w[0] +`
	`(Rss.ub[i]+Rtt.ub[i]));`
	`};`
	`for (i=4;i<8;i++) {`
	`  Rdd.w[1]=(Rdd.w[1] +`
	`(Rss.ub[i]+Rtt.ub[i]));`
	`};`

`Rxx+=vraddub(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Rxx.w[0]=(Rxx.w[0] +`
	`(Rss.ub[i]+Rtt.ub[i]));`
	`};`
	`for (i = 4; i < 8; i++) {`
	`  Rxx.w[1]=(Rxx.w[1] +`
	`(Rss.ub[i]+Rtt.ub[i]));`
	`};`

Type: M (slots 2,3)

Intrinsics

`Rdd=vraddub(Rss,Rtt)`	`Word64 Q6_P_vraddub_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vraddub(Rss,Rtt)`	`Word64 Q6_P_vraddubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

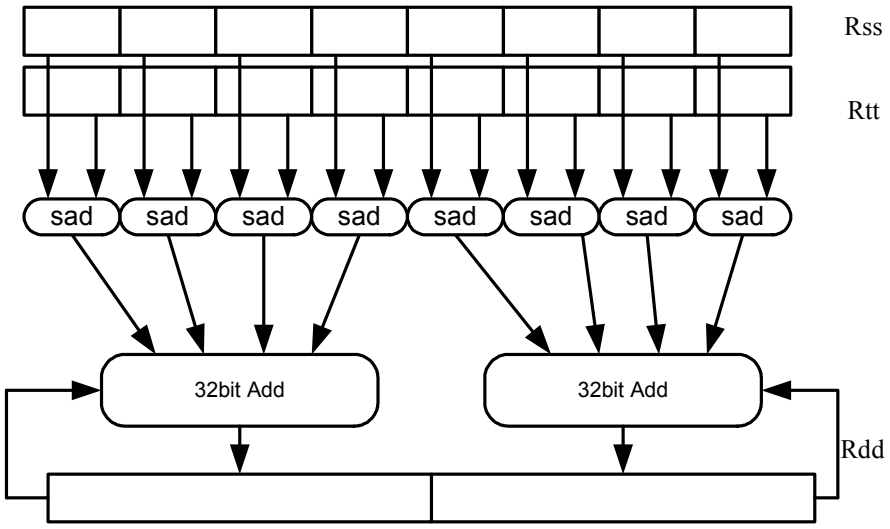
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vraddub(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vraddub(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector sum of absolute differences unsigned bytes

For each byte in the source vector Rss, subtract the corresponding byte in source vector Rtt. Take the absolute value of the intermediate results, and the upper four together and add the lower four together. Optionally, add the destination upper and lower words to these results.

This instruction is useful in determining distance between two vectors, in applications such as motion estimation.



Syntax	Behavior

`Rdd=vrsadub(Rss,Rtt)`	`Rdd = 0;`
	`for (i = 0; i < 4; i++) {`
	`  Rdd.w[0]=(Rdd.w[0] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`
	`for (i = 4; i < 8; i++) {`
	`  Rdd.w[1]=(Rdd.w[1] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`

`Rxx+=vrsadub(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Rxx.w[0]=(Rxx.w[0] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`
	`for (i = 4; i < 8; i++) {`
	`  Rxx.w[1]=(Rxx.w[1] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`

Type: M (slots 2,3)

Intrinsics

`Rdd=vrsadub(Rss,Rtt)`	`Word64 Q6_P_vrsadub_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrsadub(Rss,Rtt)`	`Word64 Q6_P_vrsadubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

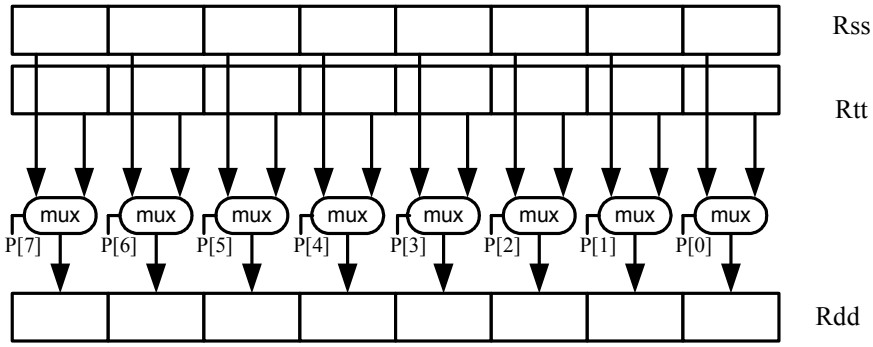
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrsadub(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrsadub(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector mux

Perform an element-wise byte selection between two vectors. For each of the low 8 bits of predicate register Pu, if the bit is set, then the corresponding byte in Rdd is set to the corresponding byte from Rss. Otherwise, set the byte in Rdd to the byte from Rtt.



Syntax	Behavior

`Rdd=vmux(Pu,Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=(Pu.i?(Rss.b[i]):(Rtt.b[i]));`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vmux(Pu,Rss,Rtt)`	`Word64 Q6_P_vmux_pPP(Byte Pu, Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType						s5					Parse			t5						u2		d5						
1	1	0	1	-	0	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=vmux(Pu,Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d

Field name	Description
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

11.10.10 XTYPE / Vector Halfword

The XTYPE vector halfword instructions perform vector operations on halfword values.

Vector absolute value halfwords

Take the absolute value of each of the four halfwords in the 64-bit source vector Rss. Place the result in Rdd. Optional saturation is available.

Syntax	Behavior

`Rdd=vabsh(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=ABS(Rss.h[i]);`
	`};`

`Rdd=vabsh(Rss):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_16(ABS(Rss.h[i]));`
	`};`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vabsh(Rss)`	`Word64 Q6_P_vabsh_P(Word64 Rss)`
`Rdd=vabsh(Rss):sat`	`Word64 Q6_P_vabsh_P_sat(Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsh(Rss)
1	0	0	0	0	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vabsh(Rss):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector add halfwords

This instruction has two forms: one supports 64-bit vectors, and the other 32-bit vectors.

In the 64-bit vector form, each of the four halfwords in 64-bit vector Rss is added to the corresponding halfword in vector Rtt. Optionally, saturate each 16-bit addition to either a signed or unsigned 16-bit value. Applying saturation to the `vaddh` instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to the `vadduh` instruction ensures that the unsigned result falls within the range 0 to 0xffff. When saturation is not needed, the `vaddh` form should be used.

For the 32-bit vector form of this instruction see the ALU32 instructions.

Syntax	Behavior

`Rdd=vaddh(Rss,Rtt)[:sat]`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=[sat_16](Rss.h[i]+Rtt.h[i]);`
	`};`

`Rdd=vadduh(Rss,Rtt):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat_16(Rss.uh[i]+Rtt.uh[i]);`
	`};`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddh(Rss,Rtt)`	`Word64 Q6_P_vaddh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddh(Rss,Rtt):sat`	`Word64 Q6_P_vaddh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vadduh(Rss,Rtt):sat`	`Word64 Q6_P_vadduh_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddh(Rss,Rtt)
1	1	0	1	-	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddh(Rss,Rtt):sat
1	1	0	1	-	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vadduh(Rss,Rtt):sat

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector average halfwords

Average each of the four halfwords in the 64-bit source vector Rss with the corresponding halfword in Rtt. The average operation performed on each halfword adds the two halfwords and shifts the result right by 1 bit. Unsigned average uses a logical right shift (shift in 0), while signed average uses an arithmetic right shift (shift in the sign bit). If the round option is used, then 0x0001 is also added to each result before shifting. This operation does not overflow. In the case that a summation (before right-shift by 1) causes an overflow of 32 bits, the value shifted in is the most-significant carry out.

The signed average and negative average halfwords is available with optional convergent rounding. In convergent rounding, if the two LSBs after the addition/subtraction are 11, then a rounding constant of 1 is added, otherwise a 0 is added. This result is then shifted right by one bit. Convergent rounding accumulates less error than arithmetic rounding.

Syntax	Behavior

`Rdd=vavgh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]+Rtt.h[i])>>1;`
	`};`

`Rdd=vavgh(Rss,Rtt):crnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=convround(Rss.h[i]+Rtt.h[i])`
	`  >>1;`
	`};`

`Rdd=vavgh(Rss,Rtt):rnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]+Rtt.h[i]+1)>>1;`
	`};`

`Rdd=vavguh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]+Rtt.uh[i])>>1;`
	`};`

`Rdd=vavguh(Rss,Rtt):rnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]+Rtt.uh[i]+1)>>1;`
	`};`

`Rdd=vnavgh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rtt.h[i]-Rss.h[i])>>1;`
	`};`

`Rdd=vnavgh(Rtt,Rss):crnd:sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_16(convround(Rtt.h[i]-`
	`  Rss.h[i])>>1);`
	`};`

`Rdd=vnavgh(Rtt,Rss):rnd:sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_16((Rtt.h[i]-`
	`  Rss.h[i]+1)>>1);`
	`};`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vavgh(Rss,Rtt)`	`Word64 Q6_P_vavgh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgh(Rss,Rtt):crnd`	`Word64 Q6_P_vavgh_PP_crnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgh(Rss,Rtt):rnd`	`Word64 Q6_P_vavgh_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavguh(Rss,Rtt)`	`Word64 Q6_P_vavguh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavguh(Rss,Rtt):rnd`	`Word64 Q6_P_vavguh_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vnavgh(Rtt,Rss)`	`Word64 Q6_P_vnavgh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgh(Rtt,Rss):crnd:sat`	`Word64 Q6_P_vnavgh_PP_crnd_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgh(Rtt,Rss):rnd:sat`	`Word64 Q6_P_vnavgh_PP_rnd_sat(Word64 Rtt, Word64 Rss)`

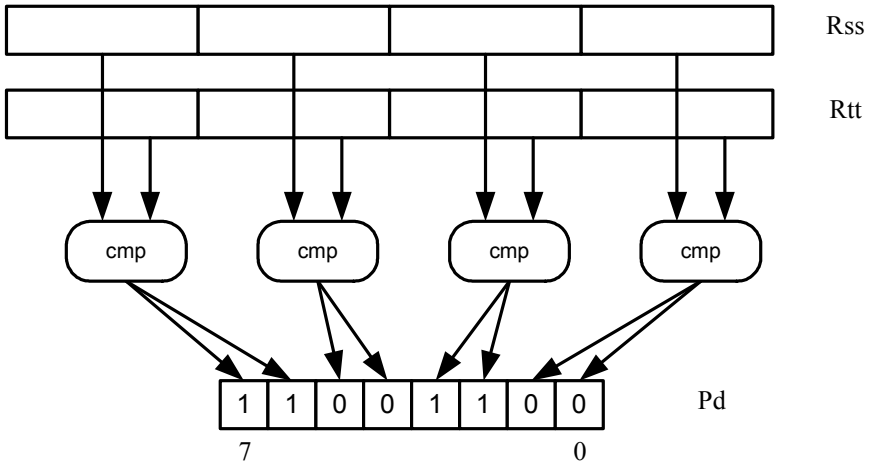
Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	0	d	d	d	d	d	Rdd=vavgh(Rss,Rtt)
1	1	0	1	-	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	1	d	d	d	d	d	Rdd=vavgh(Rss,Rtt):rnd
1	1	0	1	-	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	0	d	d	d	d	d	Rdd=vavgh(Rss,Rtt):crnd
1	1	0	1	-	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	1	d	d	d	d	d	Rdd=vavguh(Rss,Rtt)
1	1	0	1	-	0	1	1	0	1	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	d	d	d	d	d	Rdd=vavguh(Rss,Rtt):rnd
1	1	0	1	-	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	0	d	d	d	d	d	Rdd=vnavgh(Rtt,Rss)
1	1	0	1	-	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	1	d	d	d	d	d	Rdd=vnavgh(Rtt,Rss):rnd:sat
1	1	0	1	-	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	0	d	d	d	d	d	Rdd=vnavgh(Rtt,Rss):crnd:sat

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector compare halfwords

Compare each of four 16-bit halfwords in two 64-bit vectors and set the corresponding bits in a predicate destination to '11' if true, '00' if false. Halfword comparisons can be for equal, signed greater than, or unsigned greater than.



Syntax	Behavior

`Pd=vcmph.eq(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] == Rtt.h[i]);`
	`  Pd.i*2+1 = (Rss.h[i] == Rtt.h[i]);`
	`};`

`Pd=vcmph.gt(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] > Rtt.h[i]);`
	`  Pd.i*2+1 = (Rss.h[i] > Rtt.h[i]);`
	`};`

`Pd=vcmph.gtu(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.uh[i] > Rtt.uh[i]);`
	`  Pd.i*2+1 = (Rss.uh[i] > Rtt.uh[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Pd=vcmph.eq(Rss,Rtt)`	`Byte Q6_p_vcmph_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmph.gt(Rss,Rtt)`	`Byte Q6_p_vcmph_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmph.gtu(Rss,Rtt)`	`Byte Q6_p_vcmph_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp						d2		
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=vcmph.eq(Rss,Rtt)
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=vcmph.gt(Rss,Rtt)
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	-	-	-	**d**	**d**	Pd=vcmph.gtu(Rss,Rtt)

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector maximum halfwords

Compare each of the four halfwords in the 64-bit source vector Rss to the corresponding halfword in Rtt. For each comparison, select the maximum of the two halfwords and place that halfword in the corresponding location in Rdd. Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vmaxh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=max(Rss.h[i],Rtt.h[i]);`
	`};`

`Rdd=vmaxuh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=max(Rss.uh[i],Rtt.uh[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vmaxh(Rss,Rtt)`	`Word64 Q6_P_vmaxh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vmaxuh(Rss,Rtt)`	`Word64 Q6_P_vmaxuh_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxh(Rss,Rtt)
1	1	0	1	-	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxuh(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector minimum halfwords

Compare each of the four halfwords in the 64-bit source vector Rss to the corresponding halfword in Rtt. For each comparison, select the minimum of the two halfwords and place that halfword in the corresponding location in Rdd. Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vminh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=min(Rtt.h[i],Rss.h[i]);`
	`};`

`Rdd=vminuh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=min(Rtt.uh[i],Rss.uh[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vminh(Rtt,Rss)`	`Word64 Q6_P_vminh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminuh(Rtt,Rss)`	`Word64 Q6_P_vminuh_PP(Word64 Rtt, Word64 Rss)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminh(Rtt,Rss)
1	1	0	1	-	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminuh(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector subtract halfwords

This instruction has two forms: one for 64-bit vectors, the other for 32-bit vectors.

When using the 64-bit vector form, the instruction subtracts each of the four halfwords in 64-bit vector Rss from the corresponding halfword in vector Rtt. Optionally, saturate each 16-bit addition to either a signed or unsigned 16-bit value. Applying saturation to the `vsubh` instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to the `vsubuh` instruction ensures that the unsigned result falls within the range 0 to 0xffff. When saturation is not needed, the `vsubh` form should be used.

Syntax	Behavior

`Rdd=vsubh(Rtt,Rss)[:sat]`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=[sat_16](Rtt.h[i]-Rss.h[i]);`
	`};`

`Rdd=vsubuh(Rtt,Rss):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat_16(Rtt.uh[i]-Rss.uh[i]);`
	`};`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubh(Rtt,Rss)`	`Word64 Q6_P_vsubh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubh(Rtt,Rss):sat`	`Word64 Q6_P_vsubh_PP_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vsubuh(Rtt,Rss):sat`	`Word64 Q6_P_vsubuh_PP_sat(Word64 Rtt, Word64 Rss)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubh(Rtt,Rss)
1	1	0	1	-	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubh(Rtt,Rss):sat
1	1	0	1	-	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubuh(Rtt,Rss):sat

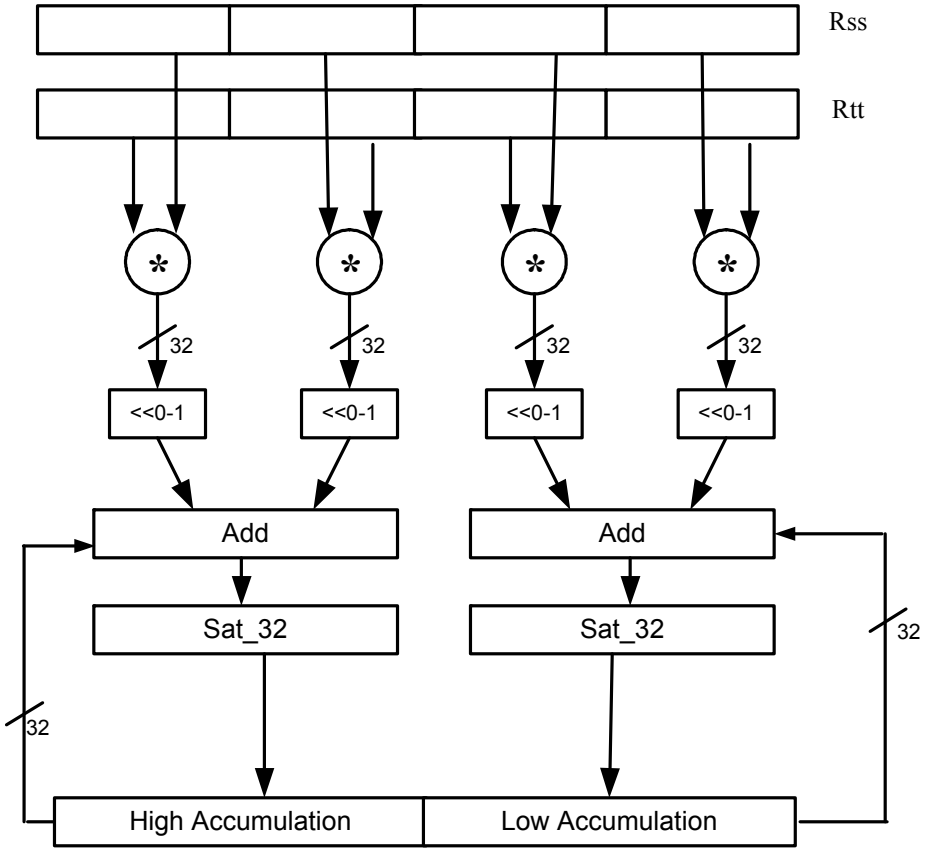
Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector dual multiply

Multiply four 16-bit halfwords in Rss by the corresponding 16-bit halfwords in Rtt. The two lower results are scaled and added. The lower word of the accumulator is optionally added. This result is saturated to 32 bits and stored in the lower word of the accumulator.

The same operation is performed on the upper two products using the upper word of the accumulator.

Rxx+=vdmpy(Rss,Rtt):sat



Syntax	Behavior
`Rdd=vdmpy(Rss,Rtt):<<1:sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<1 + (Rss.h[1] * Rtt.h[1])<<1);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<1 + (Rss.h[3] * Rtt.h[3])<<1);`
`Rdd=vdmpy(Rss,Rtt):sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<0 + (Rss.h[1] * Rtt.h[1])<<0);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<0 + (Rss.h[3] * Rtt.h[3])<<0);`
`Rxx+=vdmpy(Rss,Rtt):<<1:sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<1 + (Rss.h[1] * Rtt.h[1])<<1);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<1 + (Rss.h[3] * Rtt.h[3])<<1);`
`Rxx+=vdmpy(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<0 + (Rss.h[1] * Rtt.h[1])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<0 + (Rss.h[3] * Rtt.h[3])<<0);`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vdmpy(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vdmpy_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vdmpy(Rss,Rtt):sat`	`Word64 Q6_P_vdmpy_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpy(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vdmpyacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpy(Rss,Rtt):sat`	`Word64 Q6_P_vdmpyacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vdmpy(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vdmpy(Rss,Rtt)[:<<N]:sat

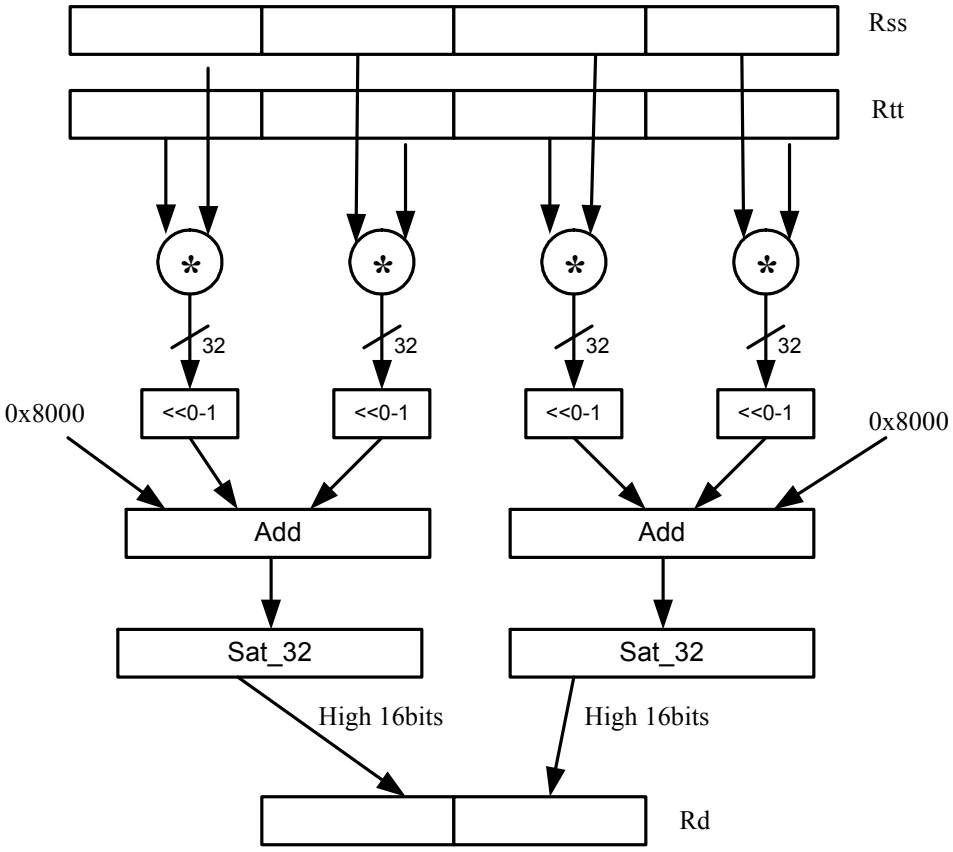
Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector dual multiply with round and pack

Multiply four 16-bit halfwords in Rss by the corresponding 16-bit halfwords in Rtt. The two lower results are scaled and added together with a rounding constant. This result is saturated to 32 bits, and the upper 16 bits of this result are stored in the lower 16 bits of the destination register.

The same operation is performed on the upper two products and the result is stored in the upper 16-bit halfword of the destination.

Rd=vdmpy(Rss,Rtt):rnd:sat



Syntax	Behavior
`Rd=vdmpy(Rss,Rtt)[:<<1]:rnd :sat`	`Rd.h[0]=(sat_32((Rss.h[0] * Rtt.h[0])[<<1] + (Rss.h[1] * Rtt.h[1])[<<1] + 0x8000)).h[1]; Rd.h[1]=(sat_32((Rss.h[2] * Rtt.h[2])[<<1] + (Rss.h[3] * Rtt.h[3])[<<1] + 0x8000)).h[1];`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vdmpy(Rss,Rtt):<<1:rnd:sat`	`Word32 Q6_R_vdmpy_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rd=vdmpy(Rss,Rtt):rnd:sat`	`Word32 Q6_R_vdmpy_PP_rnd_sat(Word64 Rss, Word64 Rtt)`

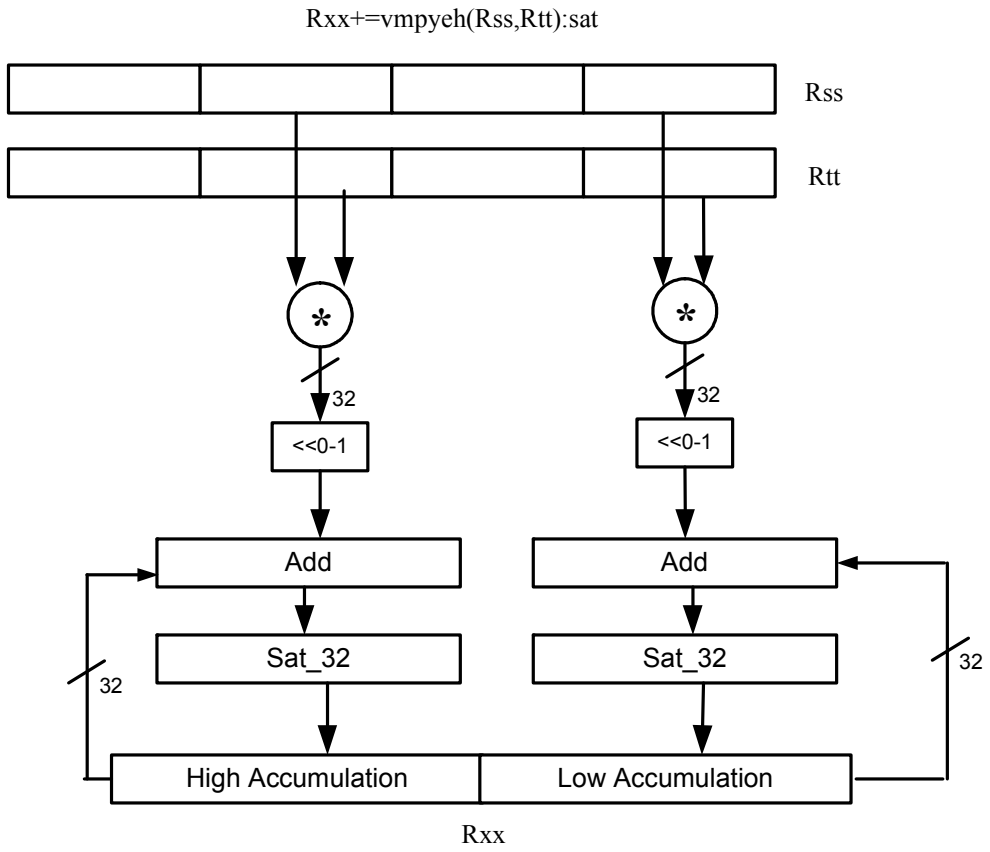
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	1	**N**	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vdmpy(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector multiply even halfwords

Multiply the even 16-bit halfwords from Rss and Rtt separately. Optionally accumulate with the low and high words of the destination register pair and optionally saturate.



Syntax	Behavior
`Rdd=vmpyeh(Rss,Rtt):<<1:sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<1);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<1);`
`Rdd=vmpyeh(Rss,Rtt):sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<0);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<0);`
`Rxx+=vmpyeh(Rss,Rtt)`	`Rxx.w[0]=Rxx.w[0] + (Rss.h[0] * Rtt.h[0]);` `Rxx.w[1]=Rxx.w[1] + (Rss.h[2] * Rtt.h[2]);`
`Rxx+=vmpyeh(Rss,Rtt):<<1:sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<1);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<1);`
`Rxx+=vmpyeh(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<0);`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyeh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyeh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyeh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyeh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt)`	`Word64 Q6_P_vmpyehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

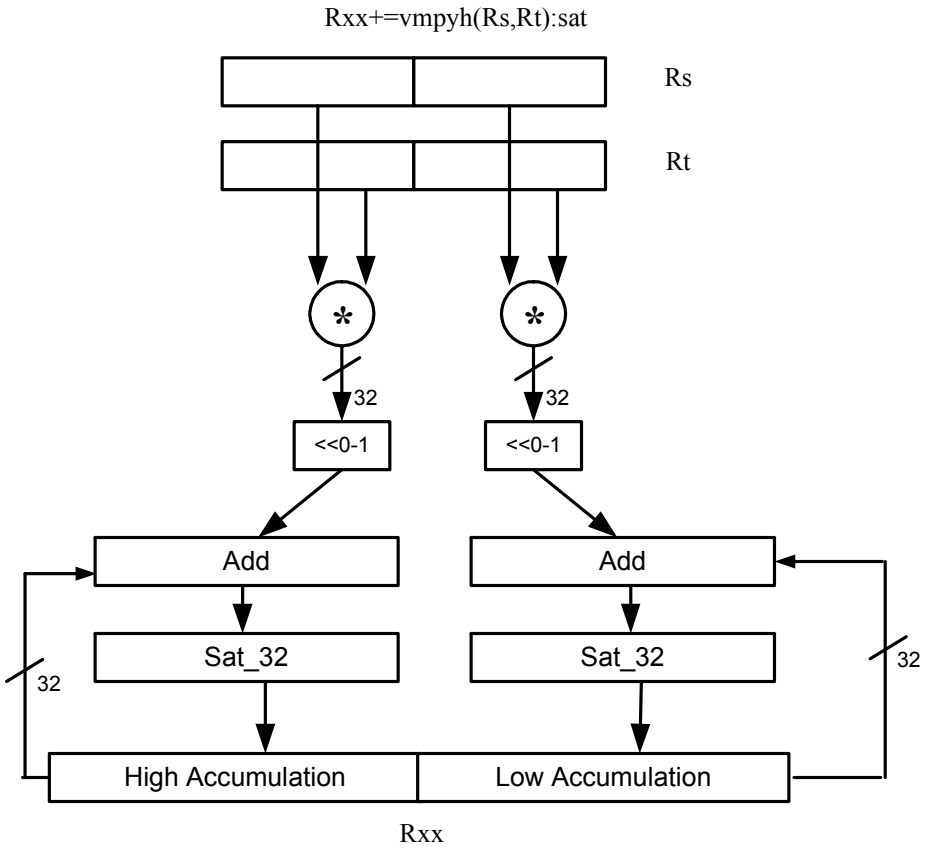
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyeh(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyeh(Rss,Rtt)
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyeh(Rss,Rtt)[:<<N]:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply halfwords

Multiply two 16-bit halfwords separately, and optionally accumulate with the low and high words of the destination. Optionally saturate, and store the results back to the destination register pair.



Syntax	Behavior
`Rdd=vmpyh(Rs,Rt)[:<<1]:sat`	`Rdd.w[0]=sat_32((Rs.h[0] * Rt.h[0])[<<1]);` `Rdd.w[1]=sat_32((Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=vmpyh(Rs,Rt)`	`Rxx.w[0]=Rxx.w[0] + (Rs.h[0] * Rt.h[0]);` `Rxx.w[1]=Rxx.w[1] + (Rs.h[1] * Rt.h[1]);`
`Rxx+=vmpyh(Rs,Rt)[:<<1]:sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1]);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rs.h[1] * Rt.h[1])[<<1]);`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyh(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyh_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=vmpyh(Rs,Rt):sat`	`Word64 Q6_P_vmpyh_RR_sat(Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt)`	`Word64 Q6_P_vmpyhacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyhacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt):sat`	`Word64 Q6_P_vmpyhacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

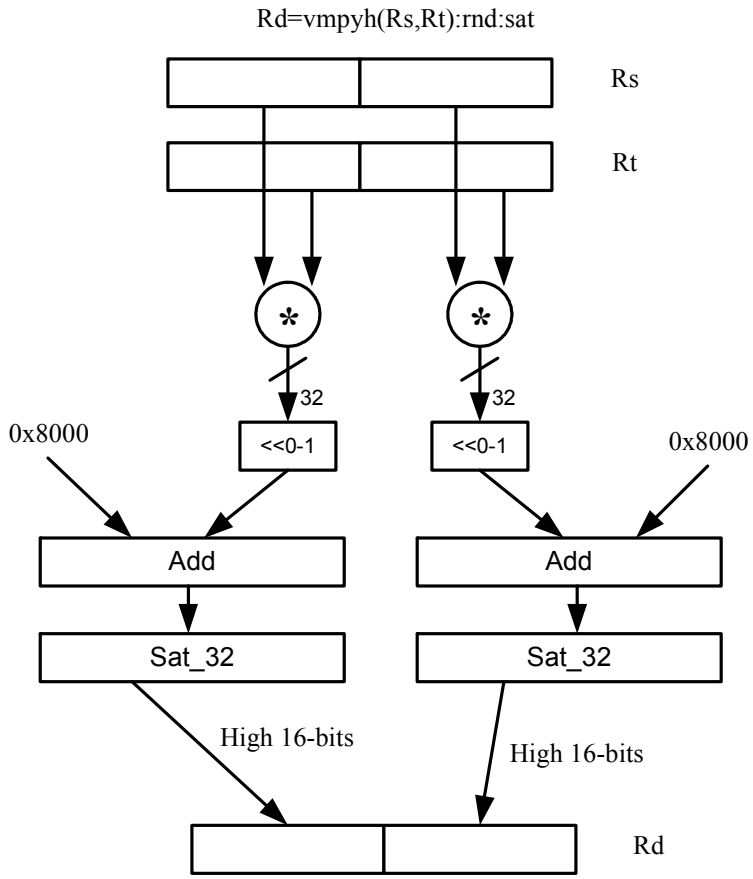
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyh(Rs,Rt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyh(Rs,Rt)
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyh(Rs,Rt)[:<<N]:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply halfwords with round and pack

Multiply two 16-bit halfwords separately. Round the results, and store the high halfwords packed in a single register destination.



Syntax	Behavior
`Rd=vmpyh(Rs,Rt)[:<<1]:rnd:sat`	`Rd.h[1]=(sat_32((Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];` `Rd.h[0]=(sat_32((Rs.h[0] * Rt.h[0])[<<1] + 0x8000)).h[1];`

Type: M (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vmpyh(Rs,Rt):<<1:rnd:sat`	`Word32 Q6_R_vmpyh_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=vmpyh(Rs,Rt):rnd:sat`	`Word32 Q6_R_vmpyh_RR_rnd_sat(Word32 Rs, Word32 Rt)`

Encoding

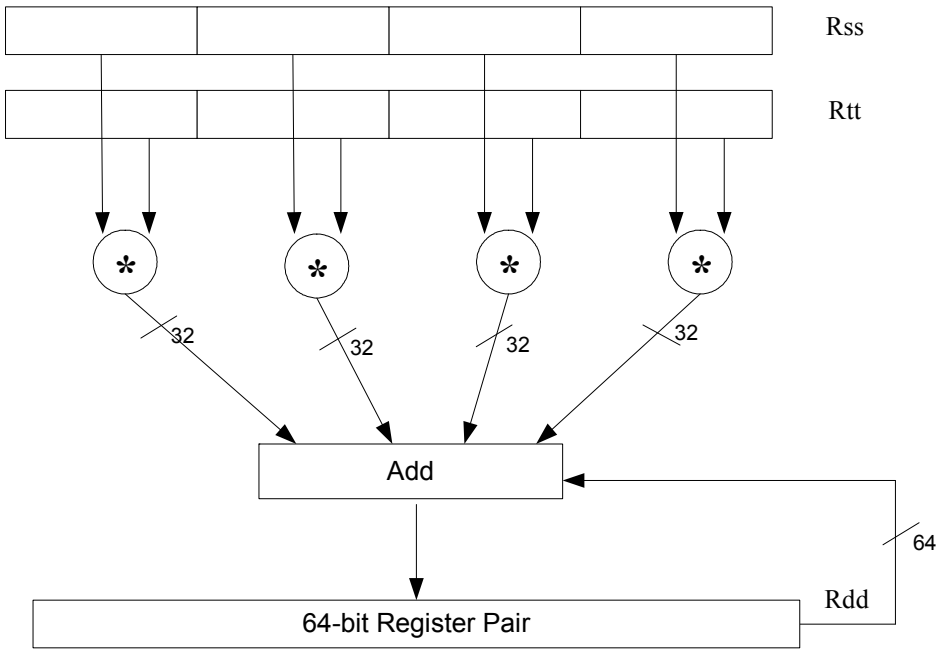
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vmpyh(Rs,Rt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector reduce multiply halfwords

Multiply each halfword of Rss by the corresponding halfword in Rtt. Add the intermediate products together and then optionally add the accumulator. The full 64-bit result is stored in the destination register pair.

This instruction is affectionately known as "big mac".



Syntax	Behavior
`Rdd=vrmpyh(Rss,Rtt)`	`Rdd = (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`
`Rxx+=vrmpyh(Rss,Rtt)`	`Rxx = Rxx + (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`

Type: M (slots 2,3)

Intrinsics

`Rdd=vrmpyh(Rss,Rtt)`	`Word64 Q6_P_vrmpyh_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpyh(Rss,Rtt)`	`Word64 Q6_P_vrmpyhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

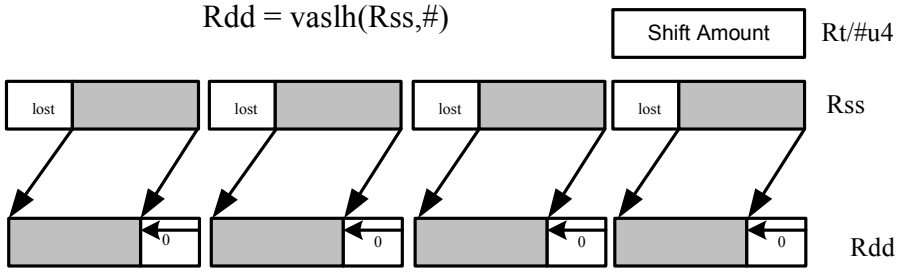
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpyh(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpyh(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector shift halfwords by immediate

Shift individual halfwords of the source vector. Arithmetic right shifts place the sign bit of the source values in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior

`Rdd=vaslh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]<<#u);`
	`};`

`Rdd=vasrh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]>>#u);`
	`};`

`Rdd=vlsrh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]>>#u);`
	`};`

Type: S (slots 2,3)

Intrinsics

`Rdd=vaslh(Rss,#u4)`	`Word64 Q6_P_vaslh_PI(Word64 Rss, Word32 Iu)`
`Rdd=vasrh(Rss,#u4)`	`Word64 Q6_P_vasrh_PI(Word64 Rss, Word32 Iu)`
`Rdd=vlsrh(Rss,#u4)`	`Word64 Q6_P_vlsrh_PI(Word64 Rss, Word32 Iu)`

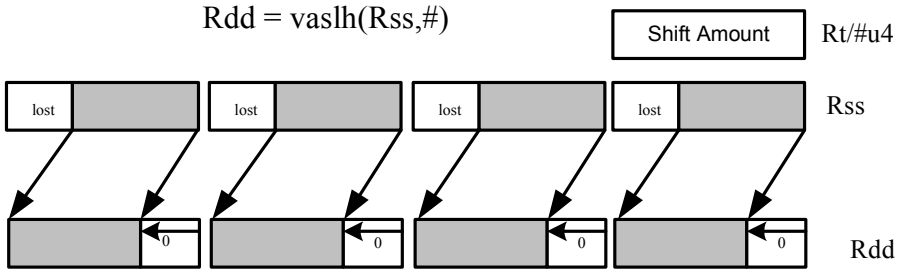
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,#u4)
1	0	0	0	0	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrh(Rss,#u4)
1	0	0	0	0	0	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaslh(Rss,#u4)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector shift halfwords by register

The shift amount is the least significant 7 bits of Rt, treated as a two's-complement value. If the shift amount is negative, the direction of the shift is reversed. Shift the source values right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rdd=vaslh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(sxt16->64(Rss.h[i])<<sxt7->32(Rt)):(sxt16->64(Rss.h[i])>>sxt7->32(Rt)); };`
`Rdd=vasrh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(sxt16->64(Rss.h[i])>>sxt7->32(Rt)):(sxt16->64(Rss.h[i])<<sxt7->32(Rt)); };`
`Rdd=vlslh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(zxt16->64(Rss.uh[i])<<sxt7->32(Rt)):(zxt16->64(Rss.uh[i])>>>sxt7->32(Rt)); };`
`Rdd=vlsrh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(zxt16->64(Rss.uh[i])>>>sxt7->32(Rt)):(zxt16->64(Rss.uh[i])<<sxt7->32(Rt)); };`

Type: S (slots 2,3)

Notes

- If the number of bits to be shifted is greater than the width of the vector element, the result is either all sign-bits (for arithmetic right shifts) or all zeros for logical and left shifts.

Intrinsics

`Rdd=vaslh(Rss,Rt)`	`Word64 Q6_P_vaslh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vasrh(Rss,Rt)`	`Word64 Q6_P_vasrh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlslh(Rss,Rt)`	`Word64 Q6_P_vlslh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlsrh(Rss,Rt)`	`Word64 Q6_P_vlsrh_PR(Word64 Rss, Word32 Rt)`

Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vaslh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlslh(Rss,Rt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

11.10.11 XTYPE / Vector Word

The XTYPE vector word instructions perform vector operations on word values.

Vector absolute value words

Take the absolute value of each of the two words in the 64-bit source vector Rss. Place the result in Rdd. Optionally saturate the result.

Syntax	Behavior

`Rdd=vabsw(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=ABS(Rss.w[i]);`
	`};`

`Rdd=vabsw(Rss):sat`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat_32(ABS(Rss.w[i]));`
	`};`

Type: S (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vabsw(Rss)`	`Word64 Q6_P_vabsw_P(Word64 Rss)`
`Rdd=vabsw(Rss):sat`	`Word64 Q6_P_vabsw_P_sat(Word64 Rss)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsw(Rss)
1	0	0	0	0	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vabsw(Rss):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector add words

Add each of the two words in 64-bit vector Rss to the corresponding word in vector Rtt. Optionally, saturate each 32-bit addition to a signed value between 0x80000000 and 0x7fffffff. The two word results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vaddw(Rss,Rtt)[:sat]`	`for (i=0;i<2;i++) { Rdd.w[i]=[sat_32](Rss.w[i]+Rtt.w[i]); };`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddw(Rss,Rtt)`	`Word64 Q6_P_vaddw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddw(Rss,Rtt):sat`	`Word64 Q6_P_vaddw_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddw(Rss,Rtt)
1	1	0	1	-	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddw(Rss,Rtt):sat

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector average words

Average each of the two words in the 64-bit source vector Rss with the corresponding word in Rtt. The average operation performed on each halfword adds the two words and shifts the result right by 1 bit. Unsigned average uses a logical right shift (shift in 0), whereas signed average uses an arithmetic right shift (shift in the sign bit). If the round option is used, then 0x1 is also added to each result before shifting.

This operation does not overflow. In the case that a summation (before right shift by 1) causes an overflow of 32 bits, the value shifted in is the most-significant carry out.

The signed average and negative average words is available with optional convergent rounding. In convergent rounding, if the two LSBs after the addition/subtraction are 11, then a rounding constant of 1 is added, otherwise a 0 is added. This result is then shifted right by one bit. Convergent rounding accumulates less error than arithmetic rounding.

Syntax	Behavior
`Rdd=vavguw(Rss,Rtt)[:rnd]`	`for (i=0;i<2;i++) { Rdd.w[i]=(zxt32->33(Rss.uw[i])+zxt32->33(Rtt.uw[i])+1)>>1; };`
`Rdd=vavgw(Rss,Rtt):crnd`	`for (i=0;i<2;i++) { Rdd.w[i]=(convround(sxt32->33(Rss.w[i])+sxt32->33(Rtt.w[i]))>>1); };`
`Rdd=vavgw(Rss,Rtt)[:rnd]`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt32->33(Rss.w[i])+sxt32->33(Rtt.w[i])+1)>>1; };`
`Rdd=vnavgw(Rtt,Rss)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i]))>>1; };`
`Rdd=vnavgw(Rtt,Rss):crnd:sat`	`for (i=0;i<2;i++) { Rdd.w[i]=sat_32(convround(sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i]))>>1); };`
`Rdd=vnavgw(Rtt,Rss):rnd:sat`	`for (i=0;i<2;i++) { Rdd.w[i]=sat_32((sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i])+1)>>1); };`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vavguw(Rss,Rtt)`	`Word64 Q6_P_vavguw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavguw(Rss,Rtt):rnd`	`Word64 Q6_P_vavguw_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt)`	`Word64 Q6_P_vavgw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt):crnd`	`Word64 Q6_P_vavgw_PP_crnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt):rnd`	`Word64 Q6_P_vavgw_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vnavgw(Rtt,Rss)`	`Word64 Q6_P_vnavgw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgw(Rtt,Rss):crnd:sat`	`Word64 Q6_P_vnavgw_PP_crnd_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgw(Rtt,Rss):rnd:sat`	`Word64 Q6_P_vnavgw_PP_rnd_sat(Word64 Rtt, Word64 Rss)`

Encoding

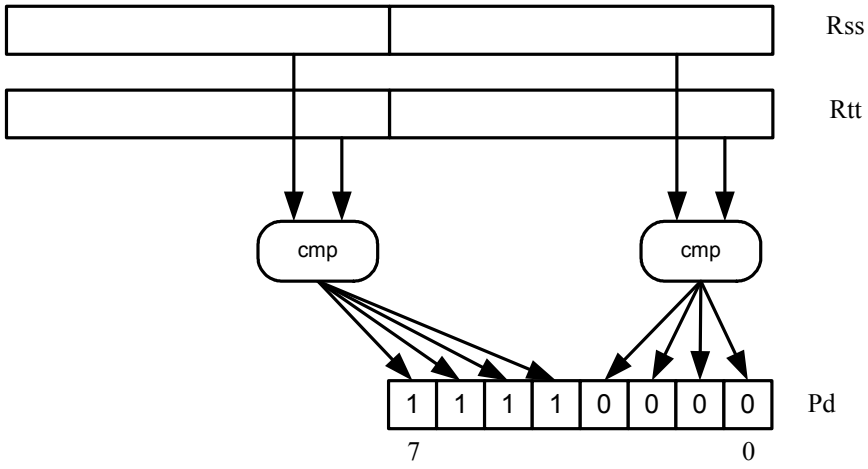
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	0	d	d	d	d	d	Rdd=vavgw(Rss,Rtt)
1	1	0	1	-	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	1	d	d	d	d	d	Rdd=vavgw(Rss,Rtt):rnd
1	1	0	1	-	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	0	d	d	d	d	d	Rdd=vavgw(Rss,Rtt):crnd
1	1	0	1	-	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	1	d	d	d	d	d	Rdd=vavguw(Rss,Rtt)
1	1	0	1	-	0	1	1	0	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	1	-	-	d	d	d	d	d	Rdd=vavguw(Rss,Rtt):rnd
1	1	0	1	-	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	1	d	d	d	d	d	Rdd=vnavgw(Rtt,Rss)
1	1	0	1	-	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	-	d	d	d	d	d	Rdd=vnavgw(Rtt,Rss):rnd:sat
1	1	0	1	-	0	1	1	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	d	d	d	d	d	Rdd=vnavgw(Rtt,Rss):crnd:sat

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector compare words

Compare each of two 32-bit words in two 64-bit vectors and set the corresponding bits in a predicate destination to '1111' if true, '0000' if false.

Compare operators supported are equal, signed greater than, and unsigned greater than.



Syntax	Behavior
`Pd=vcmpw.eq(Rss,Rtt)`	`Pd[3:0] = (Rss.w[0]==Rtt.w[0]);` `Pd[7:4] = (Rss.w[1]==Rtt.w[1]);`
`Pd=vcmpw.gt(Rss,Rtt)`	`Pd[3:0] = (Rss.w[0]>Rtt.w[0]);` `Pd[7:4] = (Rss.w[1]>Rtt.w[1]);`
`Pd=vcmpw.gtu(Rss,Rtt)`	`Pd[3:0] = (Rss.uw[0]>Rtt.uw[0]);` `Pd[7:4] = (Rss.uw[1]>Rtt.uw[1]);`

Type: ALU64 (slots 2,3)

Intrinsics

`Pd=vcmpw.eq(Rss,Rtt)`	`Byte Q6_p_vcmpw_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmpw.gt(Rss,Rtt)`	`Byte Q6_p_vcmpw_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmpw.gtu(Rss,Rtt)`	`Byte Q6_p_vcmpw_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp						d2		
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=vcmpw.eq(Rss,Rtt)
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=vcmpw.gt(Rss,Rtt)
1	1	0	1	-	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=vcmpw.gtu(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector maximum words

Compare each of the two words in the 64-bit source vector Rss to the corresponding word in Rtt. For each comparison, select the maximum of the two words and place that word in the corresponding location in Rdd.

Compare operations are available for both signed and unsigned values.

Syntax	Behavior

`Rdd=vmaxuw(Rss,Rtt)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=max(Rss.uw[i],Rtt.uw[i]);`
	`};`

`Rdd=vmaxw(Rss,Rtt)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=max(Rss.w[i],Rtt.w[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vmaxuw(Rss,Rtt)`	`Word64 Q6_P_vmaxuw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vmaxw(Rss,Rtt)`	`Word64 Q6_P_vmaxw_PP(Word64 Rss, Word64 Rtt)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxuw(Rss,Rtt)
1	1	0	1	-	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxw(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector minimum words

Compare each of the two words in the 64-bit source vector Rss to the corresponding word in Rtt. For each comparison, select the minimum of the two words and place that word in the corresponding location in Rdd.

Compare operations are available for both signed and unsigned values.

Syntax	Behavior

`Rdd=vminuw(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=min(Rtt.uw[i],Rss.uw[i]);`
	`};`

`Rdd=vminw(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=min(Rtt.w[i],Rss.w[i]);`
	`};`

Type: ALU64 (slots 2,3)

Intrinsics

`Rdd=vminuw(Rtt,Rss)`	`Word64 Q6_P_vminuw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminw(Rtt,Rss)`	`Word64 Q6_P_vminw_PP(Word64 Rtt, Word64 Rss)`

Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminw(Rtt,Rss)
1	1	0	1	-	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminuw(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector subtract words

Subtract each of the two words in 64-bit vector Rss from the corresponding word in vector Rtt. Optionally saturate each 32-bit subtractions to a signed value between 0x80000000 and 0x7fffffff. The two word results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vsubw(Rtt,Rss)[:sat]`	`for (i=0;i<2;i++) { Rdd.w[i]=[sat_32](Rtt.w[i]-Rss.w[i]); };`

Type: ALU64 (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubw(Rtt,Rss)`	`Word64 Q6_P_vsubw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubw(Rtt,Rss):sat`	`Word64 Q6_P_vsubw_PP_sat(Word64 Rtt, Word64 Rss)`

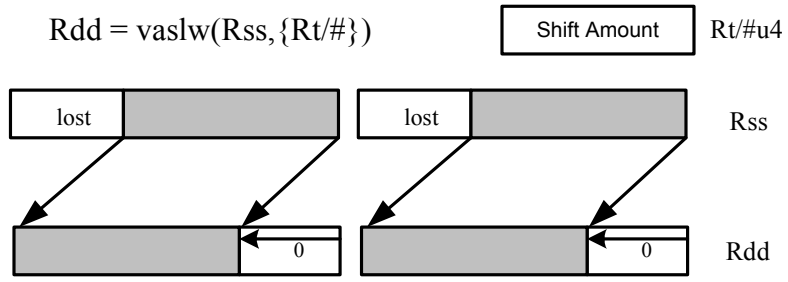
Encoding

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	
ICLASS					RegType			MajOp			s5					Parse			t5					MinOp			d5					
1	1	0	1	-	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubw(Rtt,Rss)
1	1	0	1	-	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubw(Rtt,Rss):sat

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector shift words by immediate

Shift individual words of the source vector. Arithmetic right shifts place the sign bit of the source values in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rdd=vaslw(Rss,#u5)`	`for (i=0;i<2;i++) { Rdd.w[i]=(Rss.w[i]<<#u); };`
`Rdd=vasrw(Rss,#u5)`	`for (i=0;i<2;i++) { Rdd.w[i]=(Rss.w[i]>>#u); };`
`Rdd=vlsrw(Rss,#u5)`	`for (i=0;i<2;i++) { Rdd.w[i]=(Rss.uw[i]>>#u); };`

Type: S (slots 2,3)

Intrinsics

`Rdd=vaslw(Rss,#u5)`	`Word64 Q6_P_vaslw_PI(Word64 Rss, Word32 Iu)`
`Rdd=vasrw(Rss,#u5)`	`Word64 Q6_P_vasrw_PI(Word64 Rss, Word32 Iu)`
`Rdd=vlsrw(Rss,#u5)`	`Word64 Q6_P_vlsrw_PI(Word64 Rss, Word32 Iu)`

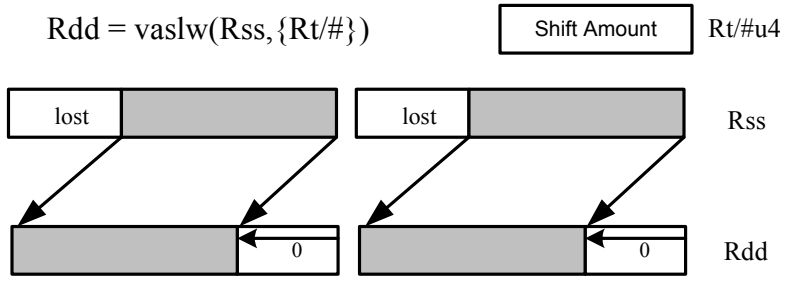
Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrw(Rss,#u5)
1	0	0	0	0	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrw(Rss,#u5)
1	0	0	0	0	0	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaslw(Rss,#u5)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector shift words by register

The shift amount is the least significant 7 bits of Rt, treated as a two's-complement value. If the shift amount is negative, the direction of the shift is reversed. Shift the source values right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rdd=vaslw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(sxt32->64(Rss.w[i])<<sxt7->32(Rt)):(sxt32->64(Rss.w[i])>>sxt7->32(Rt)); };`
`Rdd=vasrw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(sxt32->64(Rss.w[i])>>sxt7->32(Rt)):(sxt32->64(Rss.w[i])<<sxt7->32(Rt)); };`
`Rdd=vlslw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(zxt32->64(Rss.uw[i])<<sxt7->32(Rt)):(zxt32->64(Rss.uw[i])>>>sxt7->32(Rt)); };`
`Rdd=vlsrw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(zxt32->64(Rss.uw[i])>>>sxt7->32(Rt)):(zxt32->64(Rss.uw[i])<<sxt7->32(Rt)); };`

Type: S (slots 2,3)

Notes

- If the number of bits to be shifted is greater than the width of the vector element, the result is either all sign-bits (for arithmetic right shifts) or all zeros for logical and left shifts.

Intrinsics

`Rdd=vaslw(Rss,Rt)`	`Word64 Q6_P_vaslw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vasrw(Rss,Rt)`	`Word64 Q6_P_vasrw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlslw(Rss,Rt)`	`Word64 Q6_P_vlslw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlsrw(Rss,Rt)`	`Word64 Q6_P_vlsrw_PR(Word64 Rss, Word32 Rt)`

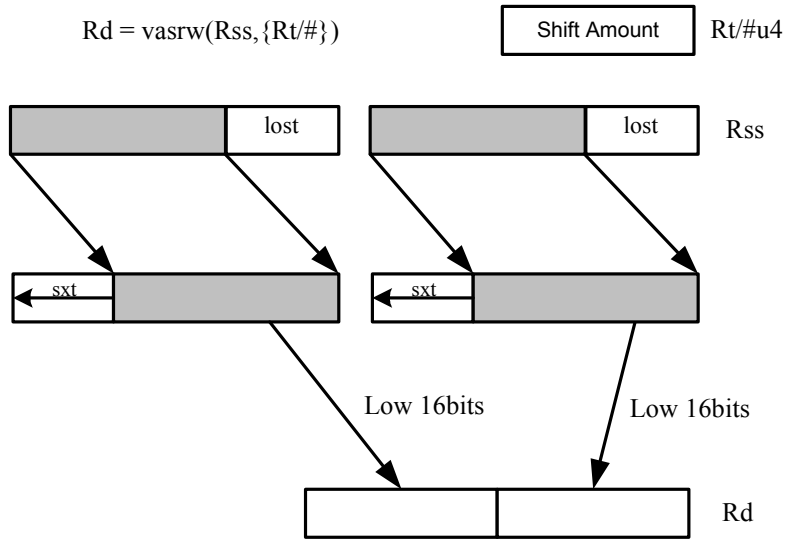
Encoding

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	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vasrw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vaslw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlslw(Rss,Rt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Vector shift words with truncate and pack

Shift individual words of the source vector Rss right by a register or immediate amount. The low 16 bits of each word are packed into destination register Rd.



Syntax	Behavior
`Rd=vasrw(Rss,#u5)`	`for (i=0;i<2;i++) { Rd.h[i]=(Rss.w[i]>>#u).h[0]; };`
`Rd=vasrw(Rss,Rt)`	`for (i=0;i<2;i++) { Rd.h[i]=(sxt7->32(Rt)>0)?(sxt32->64(Rss.w[i])>>sxt7->32(Rt)):(sxt32->64(Rss.w[i])<<sxt7->32(Rt)).h[0]; };`

Type: S (slots 2,3)

Intrinsics

`Rd=vasrw(Rss,#u5)`	`Word32 Q6_R_vasrw_PI(Word64 Rss, Word32 Iu)`
`Rd=vasrw(Rss,Rt)`	`Word32 Q6_R_vasrw_PR(Word64 Rss, Word32 Rt)`

Encoding

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	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vasrw(Rss,#u5)
ICLASS				RegType							s5					Parse			t5					Min			d5					
1	1	0	0	0	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vasrw(Rss,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

Instruction Index

C

D

E

N

O

P

S

T

V

X

Z

Intrinsics Index

A

asl

aslh

asr

deinterleave

E

extractu

I

insert

interleave

L

lfs

lsl

lsr

M

mask

max

maxu

min

minu

mpy

mpyi

mpyu

N

neg

no mnemonic

vasrw

vavgh

vavgub

vavguh

vavguw

vavgw

vcmpb.eq

vcmpb.gtu

vcmph.eq

vcmph.gt

vcmph.gtu

vcmpw.eq

vcmpw.gt

vcmpw.gtu

vcmpyi

vcmpyr

vconj

vcrotate

vdmpy

vitpack

vlslh

vlslw

vlsrh

vlsrw

vmaxh

vmaxub

vmaxuh

vmaxuw

vmaxw

Rdd=vmaxw(Rss,Rtt)	Word64 Q6_P_vmaxw_PP(Word64 Rss, Word64 Rtt)	446

vminh

Rdd=vminh(Rtt,Rss)	Word64 Q6_P_vminh_PP(Word64 Rtt, Word64 Rss)	418

vminub

Rdd=vminub(Rtt,Rss)	Word64 Q6_P_vminub_PP(Word64 Rtt, Word64 Rss)	400

vminuh

Rdd=vminuh(Rtt,Rss)	Word64 Q6_P_vminuh_PP(Word64 Rtt, Word64 Rss)	418

vminuw

Rdd=vminuw(Rtt,Rss)	Word64 Q6_P_vminuw_PP(Word64 Rtt, Word64 Rss)	447

vminw

Rdd=vminw(Rtt,Rss)	Word64 Q6_P_vminw_PP(Word64 Rtt, Word64 Rss)	447

vmpyeh

Rdd=vmpyeh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyeh_PP_s1_sat(Word64 Rss, Word64 Rtt)	427
Rdd=vmpyeh(Rss,Rtt):sat	Word64 Q6_P_vmpyeh_PP_sat(Word64 Rss, Word64 Rtt)	427
Rxx+=vmpyeh(Rss,Rtt)	Word64 Q6_P_vmpyehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	427
Rxx+=vmpyeh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	427
Rxx+=vmpyeh(Rss,Rtt):sat	Word64 Q6_P_vmpyehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	427

vmpyh

Rd=vmpyh(Rs,Rt):<<1:rnd:sat	Word32 Q6_R_vmpyh_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)	431
Rd=vmpyh(Rs,Rt):rnd:sat	Word32 Q6_R_vmpyh_RR_rnd_sat(Word32 Rs, Word32 Rt)	431
Rdd=vmpyh(Rs,Rt):<<1:sat	Word64 Q6_P_vmpyh_RR_s1_sat(Word32 Rs, Word32 Rt)	429
Rdd=vmpyh(Rs,Rt):sat	Word64 Q6_P_vmpyh_RR_sat(Word32 Rs, Word32 Rt)	429
Rxx+=vmpyh(Rs,Rt)	Word64 Q6_P_vmpyhacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	429
Rxx+=vmpyh(Rs,Rt):<<1:sat	Word64 Q6_P_vmpyhacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	429
Rxx+=vmpyh(Rs,Rt):sat	Word64 Q6_P_vmpyhacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	429

vmpyweh

Rdd=vmpyweh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)	309
Rdd=vmpyweh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweh_PP_s1_sat(Word64 Rss, Word64 Rtt)	309
Rdd=vmpyweh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweh_PP_rnd_sat(Word64 Rss, Word64 Rtt)	309
Rdd=vmpyweh(Rss,Rtt):sat	Word64 Q6_P_vmpyweh_PP_sat(Word64 Rss, Word64 Rtt)	309
Rxx+=vmpyweh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywehacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	309
Rxx+=vmpyweh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	309
Rxx+=vmpyweh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywehacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	309
Rxx+=vmpyweh(Rss,Rtt):sat	Word64 Q6_P_vmpywehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	309

vmpyweuh

Rdd=vmpyweuh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweuh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)	313
Rdd=vmpyweuh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweuh_PP_s1_sat(Word64 Rss, Word64 Rtt)	313
Rdd=vmpyweuh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweuh_PP_rnd_sat(Word64 Rss, Word64 Rtt)	313
Rdd=vmpyweuh(Rss,Rtt):sat	Word64 Q6_P_vmpyweuh_PP_sat(Word64 Rss, Word64 Rtt)	313
Rxx+=vmpyweuh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweuhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	313
Rxx+=vmpyweuh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweuhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	313
Rxx+=vmpyweuh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweuhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	313
Rxx+=vmpyweuh(Rss,Rtt):sat	Word64 Q6_P_vmpyweuhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	313

vmpywoh

vmpywouh

vmux

vnavgh

vnavgw

vraddub

vrcmpyi

vrcmpyr

vrmpyh

vrndwh

vrsadub

vsathb

vsathub

vsatwh

vsatwuh

vsplatb

vsplath

vspliceb

vsubh

vsubub

vsubuh

vsubw