TILERA®

TILE-Gx Instruction Set Architecture

Rel. 1.2
Doc. No. UG401
February 2013
Tilera Corporation

Publishing Information:

Document Number	UG401
Document Release	1.2
Date	26 February 2013

Contact Information:

Tilera Corporation
Information info@tilera.com Web Site http://www.tilera.com

Contents

CHAPTER 1 PROCESSOR ENGINE ARCHITECTURE

This chapter describes the processor engine in detail. The processor engine is the primary computational resource inside a tile. The processor engine is an asymmetric very long instruction word (VLIW) processor.

1.1 VLIW Nature of the Processor Engine

The processor engine contains three computational pipelines.

Each instruction bundle is 64-bits wide and can encode either two or three instructions. Some instructions can be encoded in either two-wide or three-wide bundles, and some can be encoded in two-wide bundles only.The most common instructions and those with short immediates can be encoded in a three instruction format.

1.2 Atomicity of Bundles

The TILE-Gx Processor architecture has a well defined, precise interrupt model with well defined instruction ordering. A bundle of instructions executes atomically. Thus either all of the instructions in the bundle are executed or none of the instructions in a bundle are executed. Inside of a single bundle, the different instructions can be dependent on many resources. In order for a bundle to execute, all of the resources upon which a bundle is dependent on must be available and ready. If one instruction in a bundle causes an exception, none of the instructions in that bundle commit state changes. Register access within a bundle is an all-or-nothing endeavor. This distinction is important for register reads as well as register writes, as register reads/writes can both modify network state when accessing network mapped registers. Memory operations are likewise atomic with respect to an instruction bundle completing.

Individual instructions within a bundle must comply with certain register semantics. Read-after-write (RAW) dependencies are enforced between instruction bundles. There is no ordering within a bundle, and the numbering of pipelines or instruction slots within a bundle is *only* used for convenience and does not imply any ordering. Within an instruction bundle, it is valid to encode an output operand that is the same as an input operand. Because there is explicitly no implied dependency within a bundle, the semantics for this specify that the input operands for all instructions in a bundle are read *before* any of the output operands are written. Write-after-write (WAW) semantics between two bundles are defined as: the latest write overwrites earlier writes.

Within a bundle, WAW dependencies are forbidden. If more than one instruction in a bundle writes to the *same* output operand register, unpredictable results for any destination operand within that bundle can occur. Also, implementations are free to signal this case as an illegal instruction. There is one exception to this rule—multiple instructions within a bundle may legally target the zero register. Lastly, some instructions, such as instructions that implicitly write the link register, implicitly write registers. If an instruction implicitly writes to a register that another instruction in the same bundle writes to, unpredictable results can occur for any output register used by that bundle and/or an illegal instruction interrupt can occur.

1.3 Register Set

The TILE-Gx Processor architecture contains 64 architected registers. Each register is 64-bits wide. Of the 64 registers, some are general purpose registers and others allow access to the on-chip networks.

Table 1 presents the registers available to be used in instructions. The first 55 registers are general purpose registers. The stack pointer `sp` is included in the 55 general purpose registers and is specified as a stack pointer only by software convention. Register lr can be used as a general purpose register. Control-transfer instructions that link have the effect of writing the value PC+8 into `lr`. Thus instructions bundled with `jal`, `jalp`, `jalr`, and `jalrp` must not write to `lr`. Note that the `LNK` instruction will write to `lr` only if `lr` is specified as the destination register. Registers idn0 and idn1 provide access to the two demultiplexed IDN networks. All writes to the IDN should use `idn0`; the result of writing to `idn1` is undefined. Registers `udn0`, `udn1`, `udn2`, and `udn3` allow access to the four demultiplexed ports of the UDN. All writes to the UDN should use `udn0`; the result of writing to `udn1-udn3` is undefined. The final register, `zero`, is a register that contains no state and always reads 0. Writes to register `63` (zero) have no effect on the register file; however, instructions that target this register might have other results, such as effecting data prefetches or causing exceptions.

Note: Note that register `r0` and register `zero` are distinct; register `r0` is a general purpose register.

Table 1 presents the register identifier mapping.

Table 1. Register Numbers

Register Numbers	Short Name	Purpose
0 - 53	r0-r53	General Purpose Registers
54	sp	Stack Pointer
55	lr	Link Register
56	r56	Reserved
57	idn0	IDN Port 0
58	idn1	IDN Port 1
59	udn0	UDN Port 0
60	udn1	UDN Port 1
61	udn2	UDN Port 2
62	udn3	UDN Port 3
63	zero	Always Returns Zero

In order to reduce latency for tile-to-tile communications and reduce instruction occupancy, the TILE-Gx Processor architecture provides access to the on-chip networks through register access. Any instruction executed in the processor engine can read or write to the following networks: UDN and IDN. There are no restrictions on the number of networks that can be written or read in a particular bundle. Each demultiplexing queue counts as an independent network for reads. For network writes, both networks (UDN and IDN) can be written to in a given instruction bundle. It is illegal for multiple instructions in a bundle to write to the same network, as this is a violation of WAW ordering for processor registers. The same network register can appear in multiple source fields in one instruction or inside of one bundle. When a single network (or demultiplex queue) is read multiple times in one bundle, only one value is dequeued from the network (demux queue) and every instruction inside of a bundle receives the same value. Network operations are atomic with respect to bundle execution.

Reading and writing networks can cause the processor to stall. If no data are available on a network port when an instruction tries to read from the corresponding network-mapped register, the entire bundle stalls waiting for the input to arrive. Likewise, if a bundle writes to a network and the output network is full, the bundle stalls until there is room in the output queue. Listing 1-1. contains example code for network reads and writes.

Listing 1-1. Network Reads and Writes

```
// add writes to udn0, sub reads
// idn0 and idn1 and writes to idn
{addi udn0, r5, 10; sub idn0, idn0, idn1}
// increment the data coming from
// udn0, add registers, and load
{addi udn0, udn0, 1; add r5, r6, r7; ld r8, r9}
// mask low bit of udn0 into r5 and
// mask second bit into r6.  reads only
// one value from udn.
{andi r5, udn0, 1; andi r6, udn0, 2}
```

1.4 Program Counter

Each processor engine contains a program counter that denotes the location of the instruction bundle that is being executed. Instruction bundles are 64 bits, thus the program counter must be aligned to 8 bytes. The program counter is modified in the natural course of program execution by branches and jumps. Also, the program counter is modified when an interrupt is signaled or when a return from interrupt instruction `iret` is executed. Instructions that link — `jal`, `jalr`, `jalrp`, and `lnk` — read the contents of the program counter for the current instruction bundle, add 8 (the length of an instruction), and write the result into a register. For `jal`, `jalr`, and `jalrp`, the register written with the link address is always `lr`; for `lnk`, the destination register is specified explicitly. Jumps that link are useful for sub-routine calls and the `lnk` instruction is useful for position independent code.

For more information, see “Control Instructions” on page 111.

1.5 Special Purpose Registers

The processor engine contains special purpose registers (SPRs) that are used to control many features of a tile. The processor engine can read an SPR with the `mfspr` instruction and write to an SPR with the `mtspr` instruction. Most SPRs are used by system software for tile configuration or for accessing context switching state.

Special purpose registers are a mixture of state and a generalized interface to control structures. Some of the special purpose registers simply hold state and provide a location to store data that is not in the general purpose register file or memory. Other special purpose registers hold no state but serve as a convenient word-oriented interface to control structures within a tile. Some SPRs possess a mixture of machine hardware status state and control functions. The act of reading or writing an SPR can cause side effects. SPRs are also the main access control mechanism for protected state in the TILE-Gx Processor architecture. The SPR space is designed so that groups of SPRs require different protection levels to access it.

1.6 TILE-Gx Processing Engine Pipeline

The TILE-Gx Processor Engine has three execution pipelines (P2, P1, P0) of two stages (EX0, EX1) each. Both modes of bundling instructions, namely the X mode and the Y mode, can issue instructions into any of the three of the execution pipelines (P2, P1, P0). Y-mode uses all three pipelines simultaneously. One of the pipelines remains in `IDLE` mode during X-mode issue. P0 is capable of executing all arithmetic and logical operations, bit and byte manipulation, selects, and all multiply and fused multiply instructions. P1 can execute all of the arithmetic and logical operations, SPR reads and writes, conditional branches, and jumps. P2 can service memory operations only: loads, stores, and test-and-set instructions.

The Processor Engine uses a short, in-order pipeline aimed at low branch latency and low load-to-use latency. The basic pipeline consists of five stages: `Fetch`, `RegisterFile`, `Execute0`, `Execute1`, and `WriteBack`.

1.6.1 Fetch

The Fetch pipeline stage runs the complete loop from updating the Program Counter (PC) through fetching an instruction to selecting a new PC. The PC provides an index into several structures in parallel: the icache data and tag arrays, the merged Branch Target Buffer and line prediction array, and the ITLB. The fetch address multiplexor must then predict the next PC based on any of several inputs: the next sequential instruction, line prediction or branch prediction, an incorrectly-predicted branch, or an interrupt.

1.6.2 RegisterFile (RF)

There are three instruction pipelines, one for each of the instructions in a bundle. These pipelines are designated as P0, P1 and P2. Bundles containing two instructions will always result in one instruction being issued in P0. The second instruction will be issued in either P1 or P2, depending on the type of instruction.

The RF stage produces valid source operands for the instructions. This operation involves four steps: decoding the two or three instructions contained in the bundle, as provided by the Fetch stage each cycle; accessing the source operands from the register file and/or network ports; checking instruction dependencies; and bypassing operand data from earlier instructions. A three-instruction bundle can require up to seven source register operands and three destination register operands — three source operands to support the fused `MulAdd` and conditional transfer operations, two source operands each for the other two instruction pipelines.

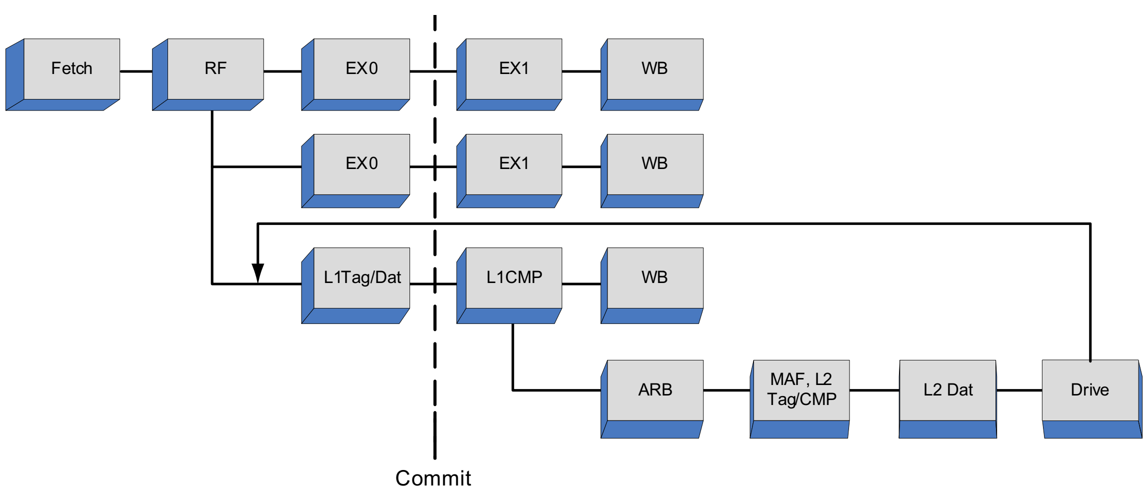

Figure 1. Processor Pipeline

1.6.3 Execute Stages (EX0, EX1)

The EX0 pipeline stage is the instruction commit point of the processor; if no exception occurs, then the architectural state can be modified. The early commit point allows the processor to transmit values computed in one tile to another tile with extremely low, register-like latencies. Single-cycle operations can bypass from the output of EX0 into the subsequent EX0. Two-cycle operations are fully pipelined and can bypass from the output of EX1 into the input of EX0.

1.6.4 WriteBack (WB)

Destination operands from P1 and P0 are written back to the Register File in the WB stage. Load data returning from memory is also written back to the Register File in the WB stage. The Register File is write-through, eliminating a bypass requirement from the output of WB into EX0.

1.6.5 Instruction/Pipeline Latencies

In a pipelined processor, multiple operations can overlap in time. In the Tile Architecture instructions that have longer latencies are fully-pipelined.

Table 2. TILE-Gx Instruction/Pipeline Latencies

Operation	Latency
Branch Mispredict	2 cycles
Load to Use - L1 hit	2 cycles
Load to Use - L1 miss, L2 hit	11 cycles
Load to Use - L1/L2 Miss, adjacent Distributed Coherent Cache (DDC™) hit	41 cycles
Load to Use - L1/L2 Miss, DDR3 page open, typical	85 cycles
Load to Use - L1/L2 Miss, DDR3 page close, typical	100 cycles
Load to Use - L1/L2 Miss, DDR3 page miss, typical	??? cycles
fsingle_{add1, sub1, mul1}	1 cycle
Other floating point, *mul*, *sad*, *adiff* instructions	2 cycles
All other instructions	1 cycle

CHAPTER 2 TILE-GX ENGINE INSTRUCTION SET

2.1 Overview

This chapter describes the Instruction Set Architecture (ISA) for the TILE-Gx Processor, Tilera's next-generation processor architecture. For a complete list of instructions, refer to "Master List of Main Processor Instructions" on page 24.

2.1 Instruction Reference

The TILE-Gx Architecture instructions can be categorized into 12 major groups:

- Arithmetic Instructions
- Bit Manipulation Instructions
- Compare Instructions
- Control Instructions
- Floating Point Instructions
- Logical Instructions
- Memory Instructions
- Memory Maintenance Instructions
- Multiply Instructions
- Nop Instructions
- SIMD Instructions
- System Instructions

2.1.1 Instruction Organization and Format

The TILE-Gx Processor architecture utilizes a 64-bit instruction bundle to specify instructions. While the bundle is a large encoding format, this encoding provides a compiler with a relatively orthogonal instruction space that aids in compilation. Likewise, the large register namespace facilitates the allocation of data into registers, but comes at the cost of extra encoding bits in an instruction word.

The TILE-Gx Processor architecture is capable of encoding up to three instructions in a bundle. In order to achieve this level of encoding density, some of the less common or large immediate operand instructions are encoded in a two instruction bundle. The bundle format is determined by the `Mode` bits 63:62. When the `Mode` bits are 00, the bundle format is a X format bundle, and when the `Mode` bits are non-zero, the bundle is a Y bundle.

Bundles that are in the Y format can encode three simultaneous operations where one is a memory operation, one is an arithmetic or jump register operation, and the last one is a arithmetic or multiplication operation. The Y bundle format contains only a simple set of instructions with 8-bit immediates. The X mode bundle is capable of encoding a superset of the instructions that can be encoded in Y mode, however only two instructions can be encoded in each bundle. X mode bundles are capable of encoding all instructions, including complex instructions such as control transfers and long immediate instructions.

Y mode instructions contain three encoding slots, Y2, Y1, and Y0. Y2 is the pipeline which executes loads and stores, Y1 is capable of executing arithmetic, logical, and jump register instructions, and Y0 is capable of executing multiply, arithmetic, and logical instructions. Figure 2-20 through Figure 2-28 present the instruction formats and encodings for the Y pipelines. X mode contains two encoding slots, X1 and X0. The X1 pipeline is capable of executing load, store, branches, arithmetic, and logical instructions by merging Y2 and Y1 pipelines. Pipeline X0 is capable of executing multiply, arithmetic, and logical instructions. Figure 2-3 through Figure 2-17 present the instruction formats and encodings for the X pipelines.

Some instruction formats, or specific instructions, contain unused fields. It is strongly recommended that these contain zeros, as future versions of the architecture may decide to assign meanings to nonzero values in these fields. Implementations are permitted, but not required, to take an Illegal Instruction interrupt when detecting a nonzero value in an unused instruction field.

Instruction formats are described in the sections that follow.

2.1.1.1 X Instruction Formats

Figure 2-1 and Figure 2-2 show the basic X format instruction encodings.

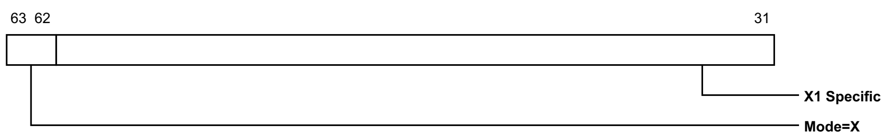

Figure 2-1: X1 Specific Format

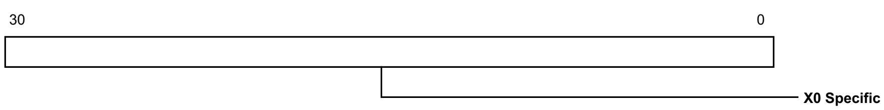

Figure 2-2: X0 Specific Format

X1 Instruction Formats

The X1 RRR format encodes an operation, which requires a destination register and two source operands. For example:

```
{add r0, r1, r2} // Add r1 and r2 placing result into r0
```

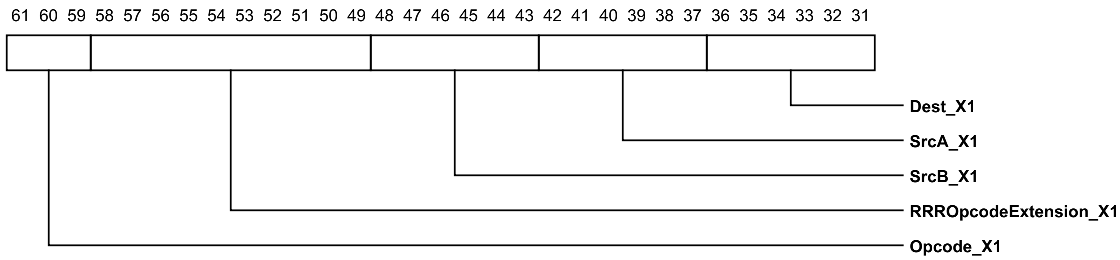

Figure 2-3: X1 RRR Format (X1_RRR)

The X1_imm8 format encodes an operation that requires a destination register, a source register, and an 8-bit signed immediate operand. For example:

```
{ addi r0, r1, -13} // Add -13 to r1 and place result in r0
```

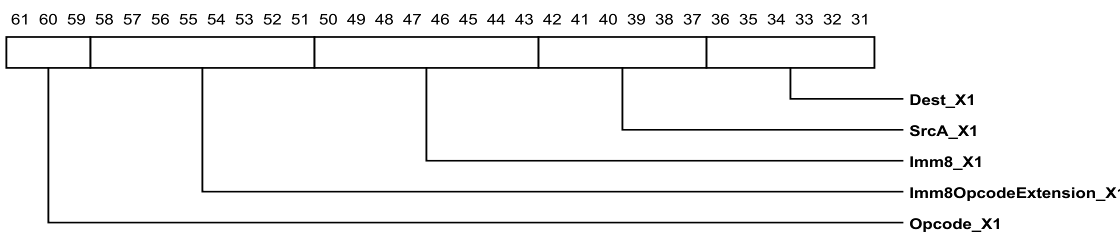

Figure 2-4: X1 Immediate Format (X1_Imm8)

The X1 Immediate MTSPR format writes an SPR with the value from a source register.For example:

```
// Move the contents of register 0 into SPR SPR_IPI_EVENT_0
{ mtspr SPR_IPI_EVENT_0, r0 }
```

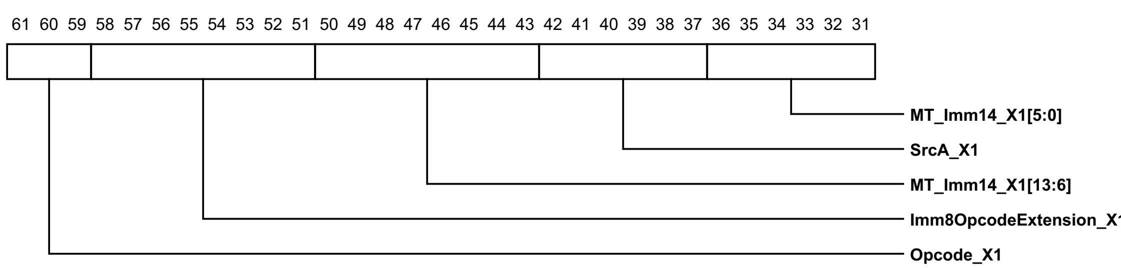

Figure 2-5: X1 Immediate MTSPR Format (X1_MT_Imm14)

The X1 Immediate MFSPR format is used to move the contents of an SPR into a destination register. For example:

```
{mfspr r0, SPR_IPI_EVENT_0}// Move the contents of the SPR SPR_IPI_EVENT_0 into
r0
```

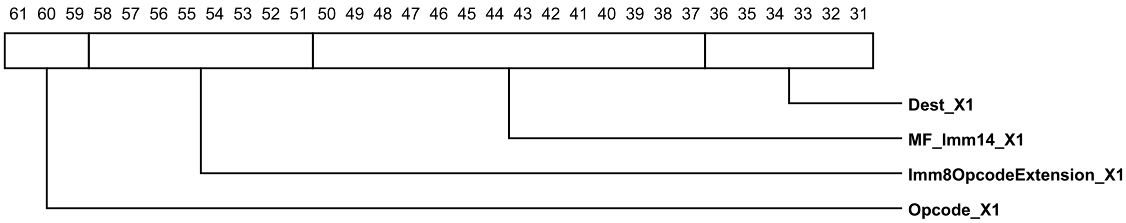

Figure 2-6: X1 Immediate MFSPR Format (X1_MF_Imm14)

The X1 Long Immediate Format is used for instructions which require a destination register, a source register and a signed 16-bit immediate operand. For example:

```
// Add 0x1234 to the contents of register 1 and place the result in register 0
{ addli r0, r1, 0x1234 }
```

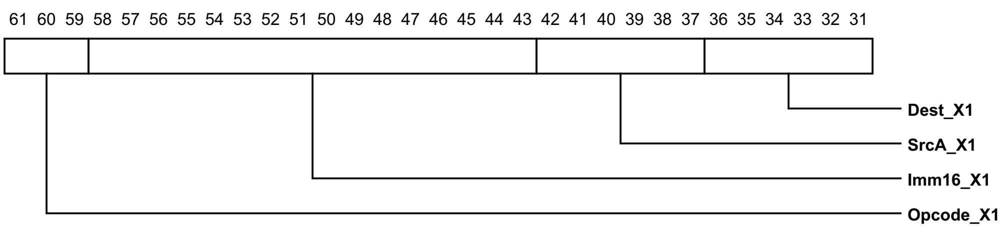

Figure 2-7: X1 Long Immediate Format (X1_Imm16)

The X1 Unary format is used for instructions which require a destination register, and a single operand register. For example:

```
{ ld r0, r1 } // Load the contents of the word addressed by r1 into r0
```

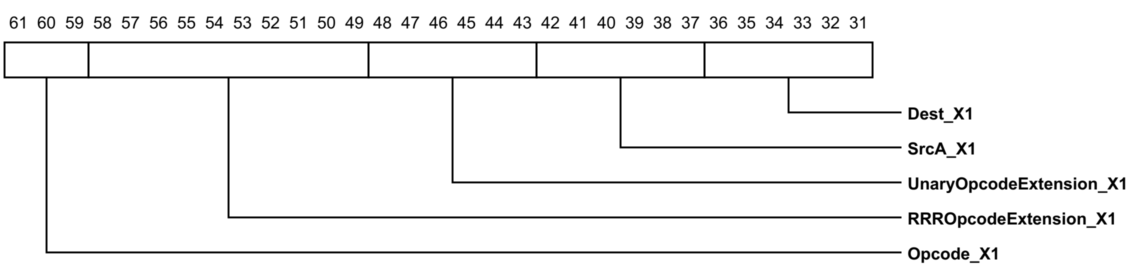

Figure 2-8: X1 Unary Format (X1_Unary)

The X1 Shift Format is used for instructions that require a destination register, a source register, and a 6-bit shift count. For example:

```
// Left shift the contents of r1 5 bits and place the result in r0.
{ shli r0, r1, 5 }
```

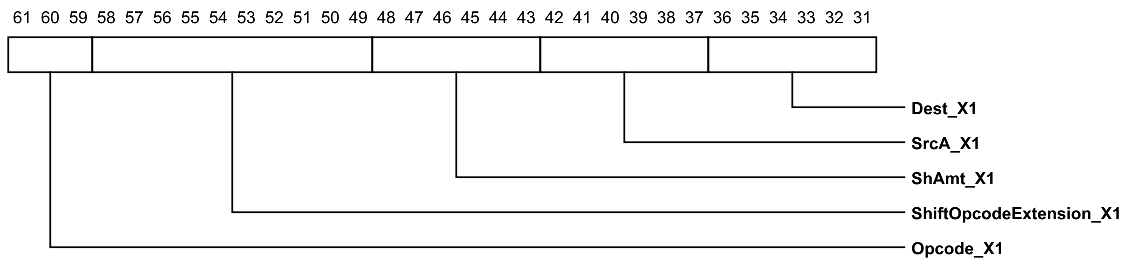

Figure 2-9: X1 Shift Format (X1_Shift)

The X1 branch format is used to encode branches. The branch offset is represented as a signed 16-bit bundle offset. For example:

```
{ bnez r0, br_target}// Branch to br_target if the contents of r0 is not zero
```

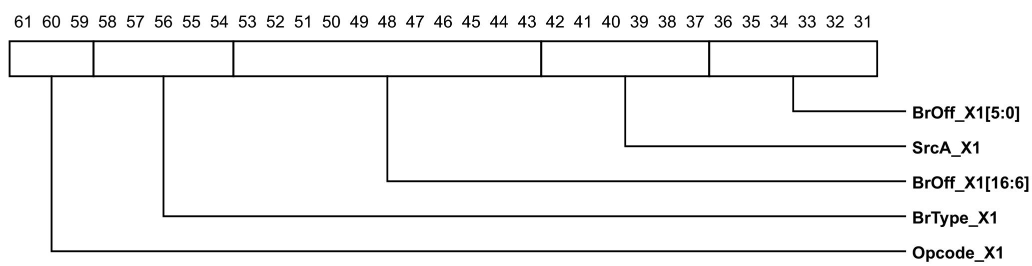

Figure 2-10: X1 Branch Format (X1_Br)

The X1 Jump format is used to encode forward or backwards jumps. The jump offset is represented as an unsigned 28-bit bundle offset. For example:

```
{ j jump_target }// Jump to jump_target
```

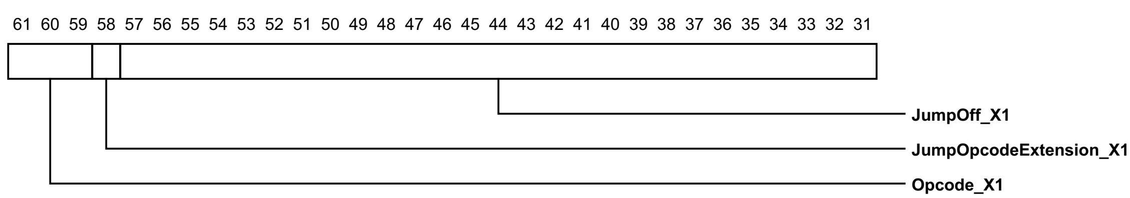

Figure 2-11: X1 Jump Format (X1_J)

X0 Instruction Formats

The X0 RRR format encodes an operation, which requires a destination register and two source operands. For example:

```
{add r0, r1, r2} // Add r1 and r2 placing result into r0
```

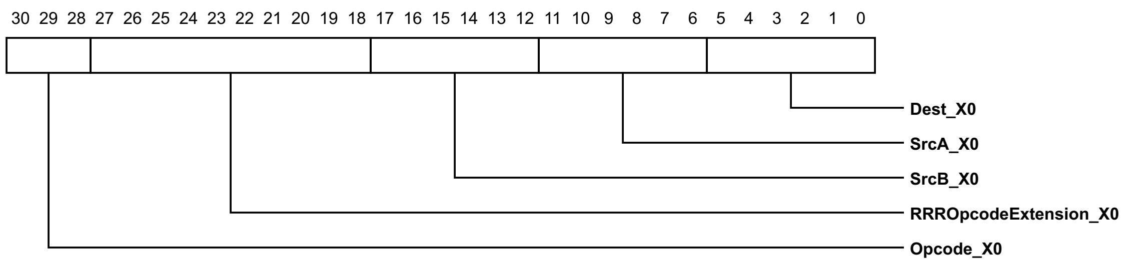

Figure 2-12: X0 RRR Format (X0_RRR)

The X0_imm8 format encodes an operation that requires a destination register, a source register, and an 8-bit signed immediate operand. For example:

```
{ addi r0, r1, -13} // Add -13 to r1 and place result in r0
```

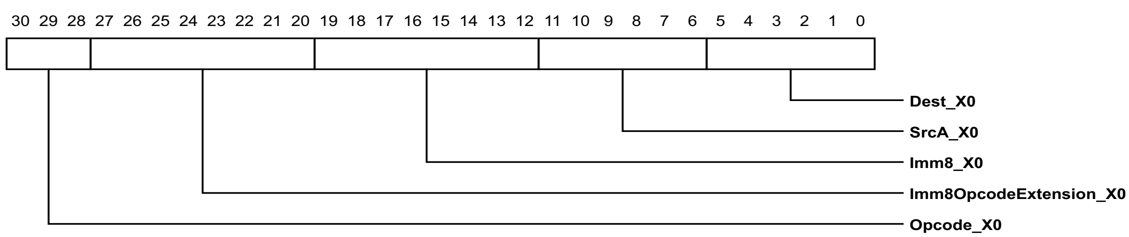

Figure 2-13: X0 Immediate Format (X0_Imm8)

The X0 Long Immediate Format is used for instructions that require a destination register, a source register, and a signed 16-bit immediate operand. For example:

```
// Add 0x1234 to the contents of register 1 and place the result in register 0
{ addli r0, r1, 0x1234 }
```

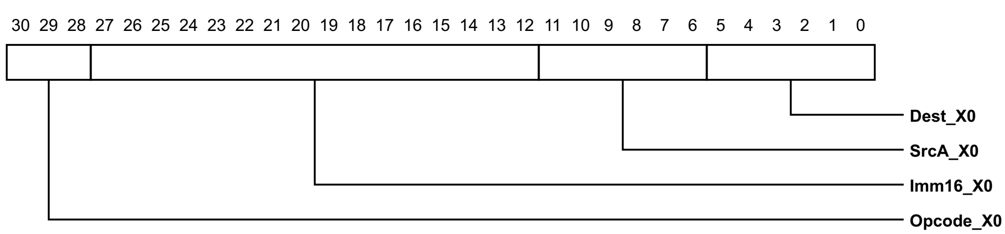

Figure 2-14: X0 Long Immediate Format (X0_Imm16)

The X0 Unary format is used for instructions that require a destination register and a single operand register. For example:

```
{ revbytes r0, r1 }// Exchange the bytes in r1 and place the result in r0
```

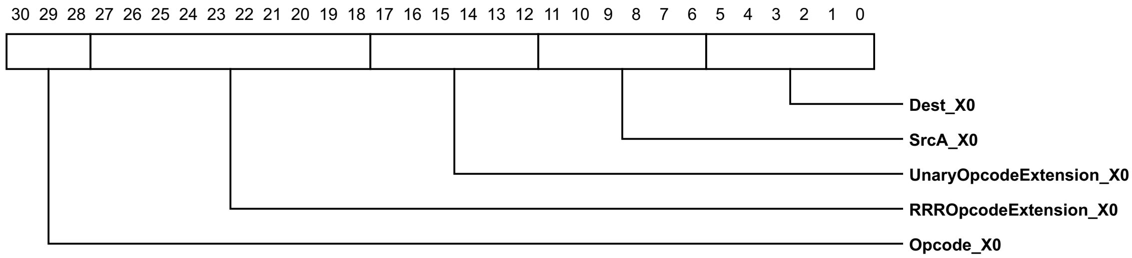

Figure 2-15: X0 Unary Format (X0_Unary)

The X0 Shift Format is used for instructions that require a destination register, a source register, and a 6-bit shift count. For example:

```
// Left shift the contents of r1 5 bits and place the result in r0.
{ shli r0, r1, 5 }
```

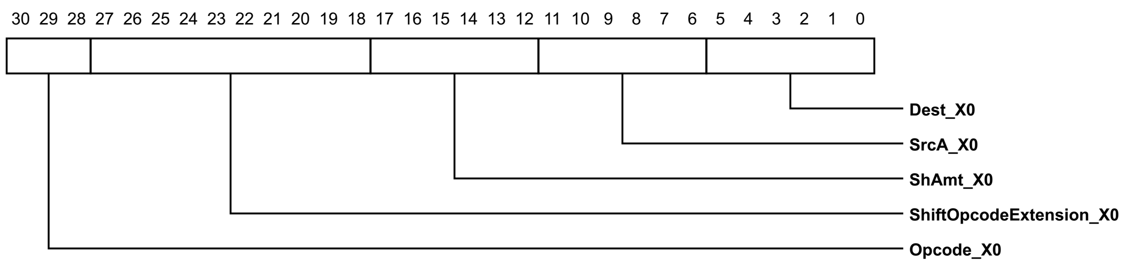

Figure 2-16: X0 Shift Format (X0_Shift)

The X0 Masked Merge format is used for the masked merge instruction. For example:

```
// Merge bits 5 through 7 of r1 into the contents of r2
//and place the result in r0
{ mm, r0, r1,  r2, 5, 7 }
```

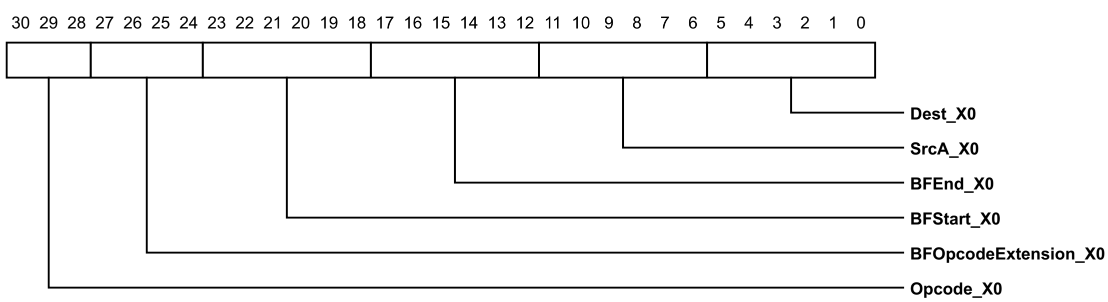

Figure 2-17: X0 Masked Merge Format (X0_MM)

2.1.1.2 Y Instruction Formats

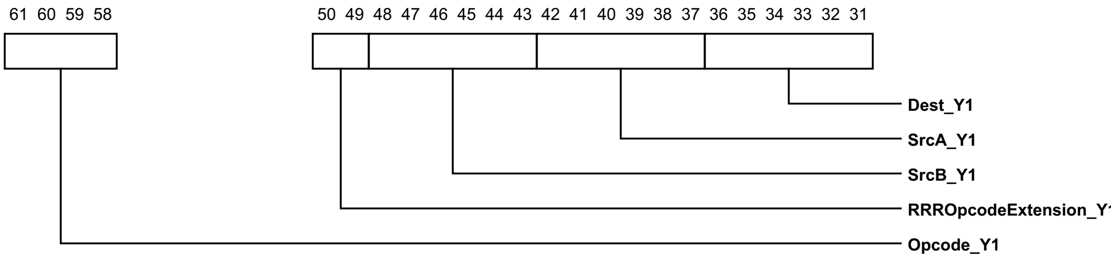

Figure 2-18: Y1 Specific Format

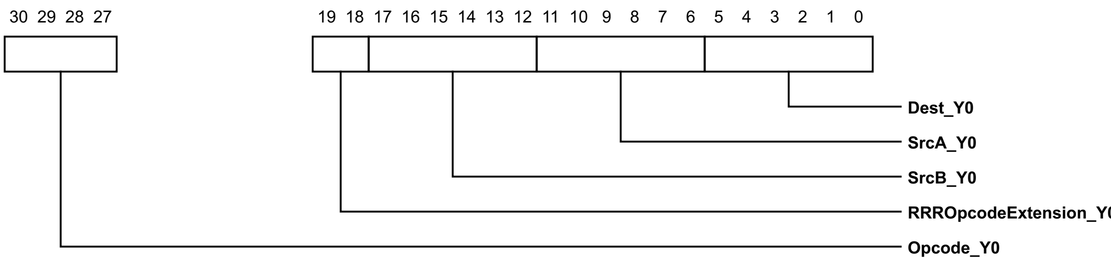

Figure 2-19: Y0 Specific Format

Y2 Instruction Formats

The Y2 Load Store Format is used to encode load or store instructions. Examples:

```
{ ld r0, r1 }// Load the contents of the word addressed by r1 into r0
{ st r0, r1}// Store the contents of register r1 into the word
// addressed by r0
```

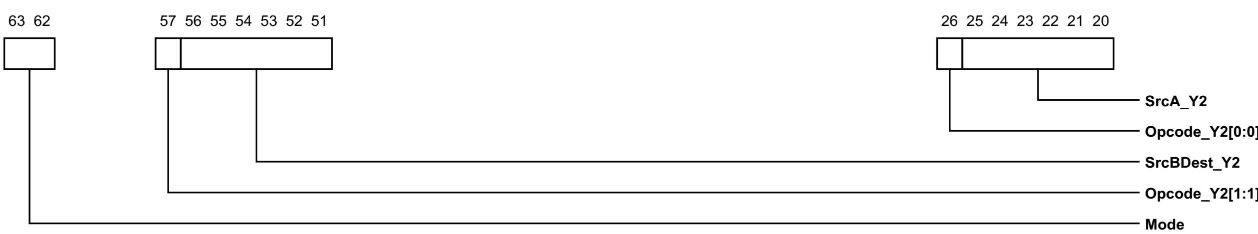

Figure 2-20: Y2 Load Store Format (Y2_LS)

Y1 Instruction Formats

The Y1 RRR format encodes an operation which requires a destination register, and two source registers. The Y1 RRR format encodes an operation, which requires a destination register and two source operands. For example:

```
{add r0, r1, r2} // Add r1 and r2 placing result into r0
```

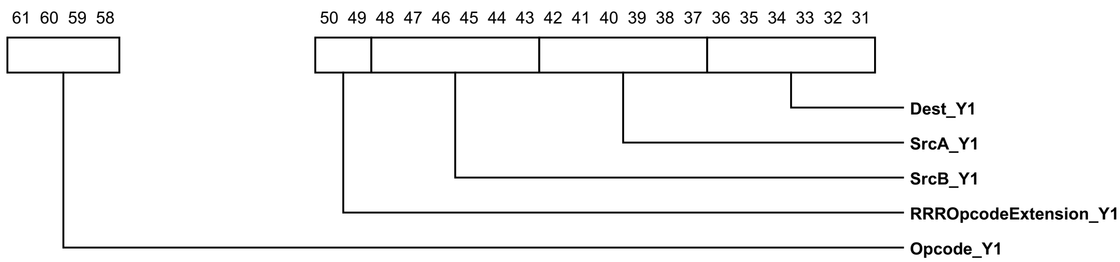

Figure 2-21: Y1 RRR Format (Y1_RRR)

The Y1_imm8 format encodes an operation that requires a destination register, a source register, and an 8-bit signed immediate operand. For example:

```
{ addi r0, r1, -13} // Add -13 to r1 and place result in r0
```

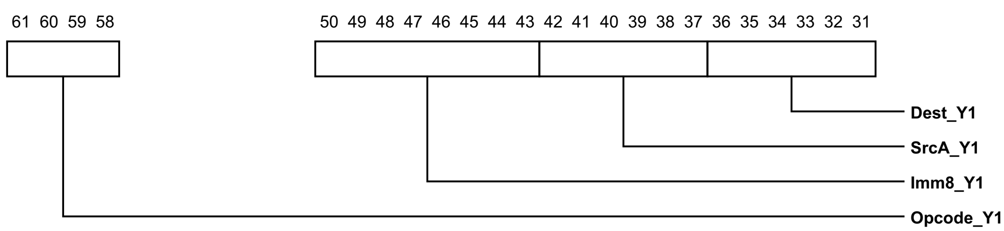

Figure 2-22: Y1 Immediate Format (Y1_Imm8)

The Y1 Unary format is used for instructions that require a destination register, and a single operand register. For example:

```
{lnk r1} // Load address of the next PC into R1
```

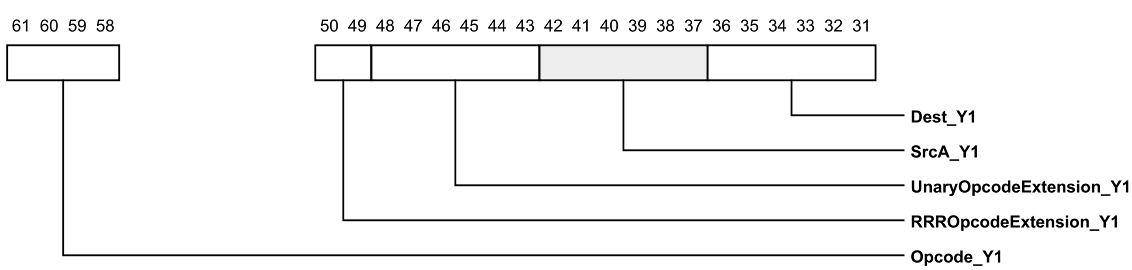

Figure 2-23: Y1 Unary Format (Y1_Unary)

The Y1 Shift Format is used for instructions that require a destination register, a source register, and a 6-bit shift count. For example:

```
// Left shift the contents of r1 5 bits and place the result in r0.
{ shli r0, r1, 5 }
```

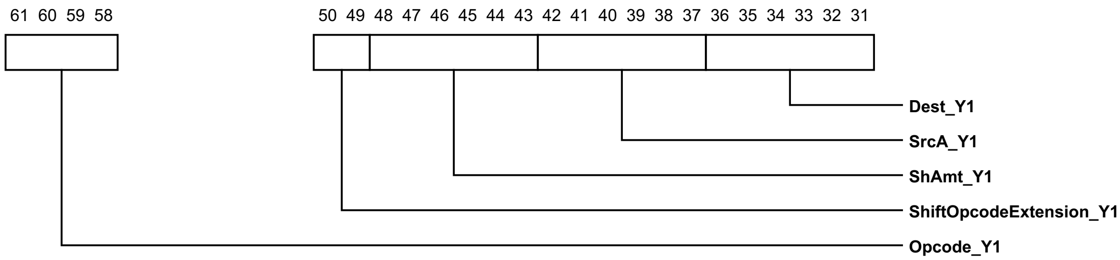

Figure 2-24: Y1 Shift Format (Y1_Shift)

Y0 Instruction Formats

The Y0 RRR format encodes an operation, which requires a destination register and two source operands. For example:

```
{add r0, r1, r2} // Add r1 and r2 placing result into r0
```

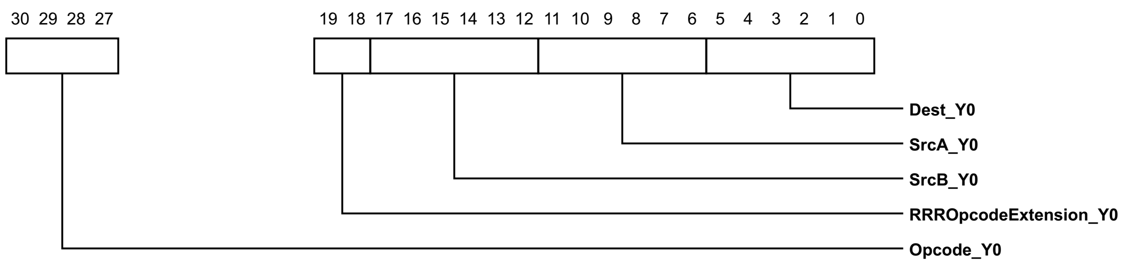

Figure 2-25: Y0 RRR Format (Y0_RRR)

The Y0_imm8 format encodes an operation that requires a destination register, a source register, and an 8-bit signed immediate operand. For example:

```
{ addi r0, r1, -13} // Add -13 to r1 and place result in r0
```

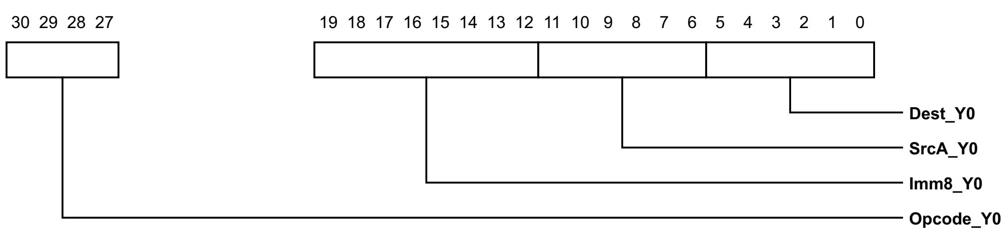

Figure 2-26: Y0 Immediate Format (Y0_Imm8)

The Y0 Unary format is used for instructions that require a destination register and a single operand register. For example:

```
{ revbytes r0, r1 }// Exchange the bytes in r1 and place the result in r0
```

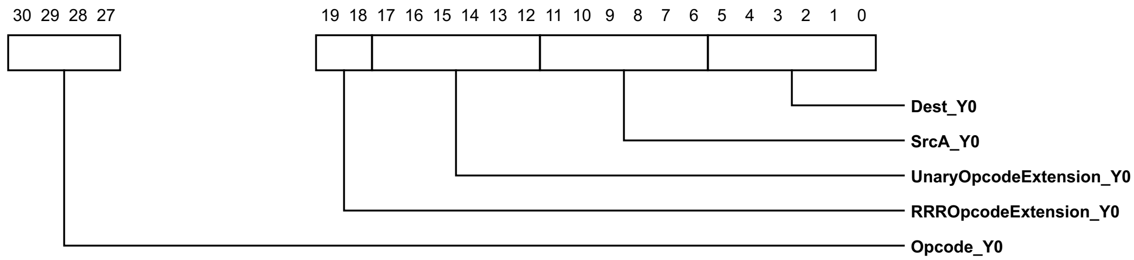

Figure 2-27: Y0 Unary Format (Y0_Unary)

The Y0 Shift Format is used for instructions that require a destination register, a source register, and a 6-bit shift count. For example:

```
// Left shift the contents of r1 5 bits and place the result in r0.
{ shli r0, r1, 5 }
```

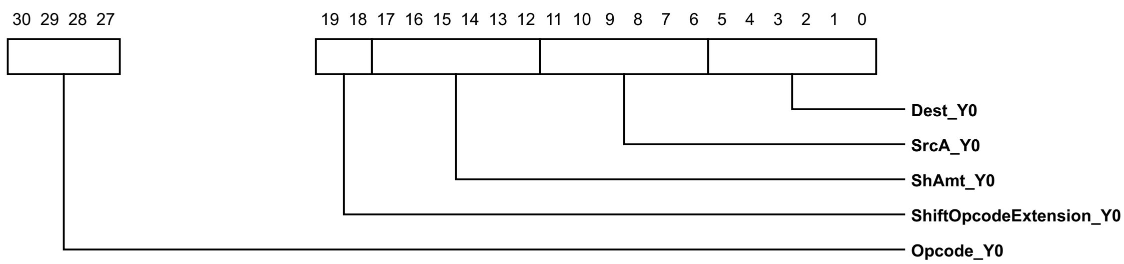

Figure 2-28: Y0 Shift Format (Y0_Shift)

2.1.2 Definitions and Semantics

Throughout the main processor's instruction reference, several function calls, types, and constants are utilized to define the function of a particular instruction. This section describes the functionality and values of each of these functions, types, and constants. Unless otherwise stated, operators and precedence in the instruction reference follow the same rules as ANSI C.

2.1.2.1 Constants

`WORD_SIZE 64`	The size of a machine word in bits. The TILE-Gx Processor is a 64-bit machine.
`WORD_MASK 0xFFFFffffFFFFffff`	A mask to represent all of the bits in a word.
`WORD_ADDR_MASK 0xFFFFffffFFFFfff8`	A mask that represents the portion of an address that forms a word aligned mask.
`BYTE_SIZE 8`	The number of bits in a byte.
`BYTE_SIZE_LOG_2 3`	The logarithm base 2 of the number of bits in a byte.
`BYTE_MASK 0xFF`	A mask to represent all of the bits in a byte.
`INSTRUCTION_SIZE 64`	The length in bits of an instruction (bundle) in the TILE-Gx Processor architecture.
`INSTRUCTION_SIZE_LOG_2 6`	The logarithm base 2 of the length in bits of an instruction (bundle) in the TILE-Gx Processor.
`ALIGNED_INSTRUCTION_MASK 0xFFFFffffFFFFfff8`	A mask that selects the relevant bits for the address of an aligned instruction.
`BYTE_16_ADDR_MASK 0xFFFFffffFFFFfff0`	A mask that represents the portion of an address that forms a 16-byte aligned block
`ZERO_REGISTER 63`	The `ZERO_REGISTER` always reads as 0, and ignores all writes.
`NUMBER_OF_REGISTERS 64`	The number of architecturally visible general purpose registers in the main processor.
`LINK_REGISTER 55`	The `LINK_REGISTER` is used as an implicit destination for some control instructions.
`EX_CONTEXT_SPRF_OFFSET`	The starting SPR address of the interrupt context save blocks. The save blocks are indexed by protection level of the interrupt handler being invoked.
`EX_CONTEXT_SIZE`	The length of the interrupt context save block.
`PC_EX_CONTEXT_OFFSET`	The register offset of the saved PC in the interrupt save context block.
`PROTECTION_LEVEL_EX_CONTEXT_OFFSET`	The register offset of the saved protection level in the interrupt save context block.
`INTERRUPT_MASK_EX_CONTEXT_OFFSET`	The register offset of the saved interrupt mask in the interrupt save context block.
`MASK16 0xFFFF`	A mask of the 16 low-order bits

2.1.2.2 Types

Table 2-1. Types

Function	Description
`SignedMachineWord`	This is a signed `WORD_SIZE` type.
`UnsignedMachineWord`	This is a unsigned `WORD_SIZE` type.
`RegisterFileEntry`	This type represents a register file entry. This type can be cast to a `UnsignedMachineWord`. This type has the assignment operator overloaded for assignments of `UnsignedMachineWord`.

2.1.2.3 Functions

Functions are arranged by group:

- General Functions
- Memory Access Functions
- Instruction-Specific Functions

General Functions

These functions are used in the description of many instructions and perform the described operation.

Table 2-2. General Functions

Function	Description
`signExtend17`	Sign extends a 17-bit value up to the machine's word length `WORD_SIZE`. The type of the returned value of this function is `SignedMachineWord;`
`signExtend16`	Sign extends a 16-bit value up to the machine's word length `WORD_SIZE`. The type of the returned value of this function is `SignedMachineWord.`
`signExtend8`	Sign extends an 8-bit value up to the machine's word length `WORD_SIZE`. The type of the returned value of this function is `SignedMachineWord.`
`signExtend1`	Sign extends an 1-bit value up to the machine's word length `WORD_SIZE`. The type of the returned value of this function is `SignedMachineWord.`
`setNextPC`	Set the program counter to this function's parameter.

Table 2-2. General Functions (continued)

Function	Description
getCurrentPC	Return as an UnsignedMachineWord the current program counter.
branchHintedCorrect	Denote that a control flow event has occurred that has been hinted correctly.
branchHintedIncorrect	Denote that a control flow event has occurred what has been hinted incorrectly.
getCurrentProtectionLevel	Returns as an UnsignedMachineWord the current protection level.
setProtectionLevel	Sets the current protection level from the first parameter.
setInterruptCriticalSection	Sets the current interrupt critical section bit from the first parameter.
flushCacheLine	Flushes the cache line from a tile's local cache which contains the address passed to this function as a parameter.
invalidataCacheLine	Invalidates the cache line from a tile's local cache which contains the address passed to this function as a parameter.
flushAndInvalidataCacheLine	Flushes and invalidates the cache line from a tile's local cache which contains the address passed to this function as a parameter.
rf[]	Returns the indexed register file entry with type RegisterFileEntry. The index is an integer in the range of 0 to NUMBER_OF_REGISTERS - 1.
sprf[]	Returns the indexed special purpose register file entry. The index is an integer in the range of 0 to 2^{15} - 1.
pushReturnStack	Pushes the parameter onto the return prediction stack.
popReturnStack	Returns the top of the return prediction stack and pops the stack.
indirectBranchHintedIncorrect	Denote that an indirect branch has occurred and has been hinted incorrectly.
indirectBranchHintedCorrect	Denote that an indirect branch has occurred and has been hinted correctly.
getHighHalfWordUnsigned	Returns the high-order half word of the parameter.

Table 2-2. General Functions (continued)

Function	Description
`getLowHalfWordUnsigned`	Returns the low-order half word of the parameter.
`illegalInstruction`	Denotes that an illegal instruction has occurred.
`softwareInterrupt`	Denotes that a software interrupt has occurred. The parameter specifies which software interrupt will be generated.

Memory Access Functions

These functions are used in instructions which perform memory operations. Functions which end in "NA" do not cause alignment traps, and access the Double word which contains the byte specified by the address. Functions which end in "NonTemporal" supply a hint to the cache subsystem that this storage will not be accessed again within a short time period. The cache subsystem typically allocates storage, such that it will be reused very quickly, thus preserving capacity for other storage elements.

Table 2-3. Memory Access Functions

Function	Description

`memoryReadDoubleWord`
`memoryReadDoubleWordNA`
`memoryReadDoubleWordNonTemporal`
	Returns the 8-byte value stored in memory at the address passed to the function. The value is a UnsignedMachineWord.
	The address passed as a parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.

`memoryReadHalfWord`	Returns the 2-byte value stored in memory at the address passed to the function. The value is 0-extended to a UnsignedMachineWord. The address passed as a parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.
`memoryReadByte`	Returns the 1-byte value stored in memory at the address passed to the function. The value is 0-extended to a UnsignedMachineWord. The address passed as a parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.

Table 2-3. Memory Access Functions (continued)

Function	Description
`memoryReadWord`	Returns the 4-byte value stored in memory at the address passed to the function. The value is 0-extended to a UnsignedMachineWord. The address passed as a parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.
`memoryWriteWord` `memoryWriteWordNonTemporal`	Writes to memory 4-byte value of the second parameter into the address passed to this function as the first parameter. The address passed as the first parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.
`memoryWriteDoubleWord`	Writes to memory 8-byte value of the second parameter into the address passed to this function as the first parameter. The address passed as the first parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.
`memoryWriteHalfWord`	Writes to memory 2-byte value of the second parameter into the address passed to this function as the first parameter. The address passed as the first parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.
`memoryWriteByte`	Writes to memory 1-byte value of the second parameter into the address passed to this function as the first parameter. The address passed as the first parameter to this function is processed depending on the memory mode and contents of the TLB. The TILE-Gx Processor is a little endian machine.

Instruction-Specific Functions

These functions are used by one or two instructions. The function performed is described in the instruction definition.

Table 2-4. Instruction-Specific Functions

Function	Description
`dtlbProbe`	See "dtlbpr" on page 200.
`memoryFence`	See "mf" on page 205.
`iCoherent`	See "icoh" on page 481.
`fnop`	See "fnop" on page 300.
`nop`	See "nop" on page 302.
`drain`	See "drain" on page 480.
`nap`	See "nap" on page 486.
`fsingle_addsub1`	See "fsingle_add1" on page 144 and "fsingle_sub1" on page 150.
`fsingle_addsub2`	See "fsingle_addsub2" on page 145.
`fsingle_mul1`	See "fsingle_mul1" on page 146.
`fsingle_mul2`	See "fsingle_mul2" on page 147.
`fsingle_pack1`	See "fsingle_pack1" on page 148.
`fsingle_pack2`	See "fsingle_pack2" on page 149.
`fdouble_pack1`	See "fdouble_pack1" on page 139.
`fdouble_pack2`	See "fdouble_pack2" on page 140.
`fdouble_mul_flags`	See "fdouble_mul_flags" on page 138.
`fdouble_addsub_flags`	See "fdouble_add_flags" on page 136 and "fdouble_addsub" on page 137.
`fdouble_addsub`	See "fdouble_addsub" on page 137.
`fdouble_unpack_minmax`	See "fdouble_unpack_min" on page 143 and "fdouble_unpack_max" on page 142.

2.1.3 Master List of Main Processor Instructions

Table 3 provides a complete list of instructions in alphabetic order. Pseudo Instructions are listed on page 44. In the table, the *Slots* columns indicate Valid Execution Slots. *Type* indicates Instruction Type abbreviated as follows:

Abbreviation	Description	Abbreviation	Description
A	Arithmetic	ML	Multiply
B	Bit Manipulation	MM	Memory Maintenance
CM	Compare	N	NOP
CT	Control	PS	Pseudo Instructions
FP	Floating Point	S	System
L	Logical	SM	SIMD
M	Memory		

Note: Instructions with "Extend" in the description operate on half words.

Table 3. Master List of Main Processor Instructions

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`add`	Add Arithmetic	A	X	X	X	X		page 48
`addi`	Add Immediate	A	X	X	X	X		page 50
`addli`	Add Long Immediate	A	X	X				page 52
`addx`	Add and Extend	A	X	X	X	X		page 53
`addxi`	Add and Extend Immediate	A	X	X	X	X		page 55
`addxli`	Add and Extend Long Immediate	A	X	X				page 57
`addxsc`	Add Signed Clamped and Extend	A	X	X				page 58
`and`	And	L	X	X	X	X		page 153
`andi`	And Immediate	L	X	X	X	X		page 155
`beqz`	Branch Equal Zero	CT		X				page 112

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`beqzt`	Branch Zero Predict Taken	CT		X				page 114
`bfexts`	Bit field Extract Signed	L	X					page 157
`bfextu`	Bit field Extract Unsigned	L	X					page 158
`bfins`	Bit field Insert	L	X					page 159
`bgez`	Branch Greater Than or Equal to Zero	CT		X				page 113
`bgezt`	Branch Greater Than or Equal to Zero Pre-dict Taken	CT		X				page 114
`bgtz`	Branch Greater Than Zero	CT		X				page 116
`bgtzt`	Branch Greater Than Zero Predict Taken	CT		X				page 117
`blbc`	Branch Low Bit Clear	CT		X				page 118
`blbct`	Branch Low Bit Clear Taken	CT		X				page 119
`blbs`	Branch Low Bit Set	CT		X				page 120
`blbst`	Branch Low Bit Set Taken	CT		X				page 121
`blez`	Branch Less Than or Equal to Zero	CT		X				page 122
`blezt`	Branch Less Than or Equal to Zero Taken	CT		X				page 123
`bltz`	Branch Less Than Zero	CT		X				page 124
`bltzt`	Branch Less Than Zero Taken	CT		X				page 125
`bnez`	Branch Not Equal Zero	CT		X				page 126
`bnezt`	Branch Not Equal Zero Predict Taken	CT		X				page 127

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`bpt`	Breakpoint	PS		X				page 306
`clz`	Count Leading Zeros	B	X		X			page 78
`cmoveqz`	Conditional Move If Equal Zero	L	X		X			page 160
`cmovnez`	Conditional Move If Not Equal Zero	L	X		X			page 161
`cmpeq`	Compare Equal	CM	X	X	X	X		page 94
`cmpeqi`	Compare Equal Immediate	CM	X	X	X	X		page 96
`cmpexch`	Compare and Exchange	M		X				page 209
`cmpexch4`	Compare and Exchange Four Bytes	M		X				page 210
`cmples`	Compare Less Than or Equal Signed	CM	X	X	X	X		page 98
`cmpleu`	Compare Less Than or Equal Unsigned	CM	X	X	X	X		page 100
`cmplts`	Compare Less Than Signed	CM	X	X	X	X		page 102
`cmpltsi`	Compare Less Than Signed Immediate	CM	X	X	X	X		page 102
`cmpltu`	Compare Less Than Unsigned	CM	X	X	X	X		page 106
`cmpltui`	Compare Less Than Unsigned Immediate	CM	X	X				page 108
`cmpne`	Compare Not Equal	CM	X	X	X	X		page 109
`cmul`	Complex Multiply	ML	X					page 269
`cmula`	Complex Multiply Accumulate	ML	X					page 270
`cmulaf`	Complex Multiply Accumulate Fixed Point	ML	X					page 271

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`cmulf`	Complex Multiply Fixed Point	ML	X					page 272
`cmulfr`	Complex Multiply Fixed Point Round	ML	X					page 273
`cmulh`	Complex Multiply High Result	ML	X					page 274
`cmulhr`	Complex Multiply High Result Round	ML	X					page 275
`crc32_32`	CRC32 32-bit Step	B	X					page 80
`crc32_8`	CRC32 8-bit Step	B	X					page 81
`ctz`	Count Trailing Zeros	B	X		X			page 82
`dblalign`	Double Align	B	X					page 84
`dblalign2`	Double Align by Two Bytes	B	X	X				page 85
`dblalign4`	Double Align by Four Bytes	B	X	X				page 86
`dblalign6`	Double Align by Six Bytes	B	X	X				page 87
`drain`	Drain Instruction	S		X				page 480
`dtlbpr`	Data TLB Probe	MM		X				page 200
`exch`	Exchange	M		X				page 211
`exch4`	Exchange Four Bytes	M		X				page 212
`fdouble_add_flags`	Floating Point Double Precision Add Flags	FP	X					page 136
`fdouble_addsub`	Floating Point Double Precision Add or Subtract	FP	X					page 137
`fdouble_mul_flags`	Floating Point Double Precision Multiply Flags	FP	X					page 138

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`fdouble_pack1`	Floating Point Double Precision Pack Part 1	FP	X					page 139
`fdouble_pack2`	Floating Point Double Precision Pack Part 2	FP	X					page 140
`fdouble_sub_flags`	Floating Point Double Precision Subtract Flags	FP	X					page 141
`fdouble_unpack_max`	Floating Point Double Precision Unpack Max	FP	X					page 142
`fdouble_unpack_min`	Floating Point Double Precision Unpack Min	FP	X					page 143
`fetchadd`	Fetch and Add	M		X				page 213
`fetchadd4`	Fetch and Add Four Bytes	M		X				page 214
`fetchaddgez`	Fetch and Add if Greater or Equal Zero	M		X				page 215
`fetchaddgez4`	Fetch and Add if Greater or Equal Zero Four Bytes	M		X				page 216
`fetchand`	Fetch and And	M		X				page 217
`fetchand4`	Fetch and And Four Bytes	M		X				page 218
`fetchor`	Fetch and Or	M		X				page 219
`fetchor4`	Fetch and Or Four Bytes	M		X				page 220
`finv`	Flush and Invalidate Cache Line	MM		X				page 201
`flush`	Flush Cache Line	MM		X				page 202
`flushwb`	Flush Write Buffers	MM		X				page 203
`fnop`	Filler No Operation	N	X	X	X	X		page 300

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`fsingle_add1`	Floating Point Single Precision Add Part 1	FP	X					page 144
`fsingle_addsub2`	Floating Point Single Precision Add or Subtract Part 2	FP	X					page 145
`fsingle_mul1`	Floating Point Single Precision Multiply Part 1	FP	X					page 146
`fsingle_mul2`	Floating Point Single Precision Multiply Part 2	FP	X					page 147
`fsingle_pack1`	Floating Point Single Precision Pack Part 1	FP	X		X			page 148
`fsingle_pack2`	Floating Point Single Precision Pack Part 2	FP	X					page 149
`fsingle_sub1`	Floating Point Single Precision Subtract Part 1	FP	X					page 150
`icoh`	Instruction Stream Coherence	S		X				page 481
`ill`	Illegal Instruction	S		X		X		page 482
`info`	Informational Note	PS	X	X	X	X		page 307
`infol`	Long Informational Note	PS	X	X				page 309
`inv`	Invalidate Cache Line	MM		X				page 204
`iret`	Interrupt Return	S		X				page 483
`j`	Jump	CT		X				page 128
`jal`	Jump and Link	CT		X				page 129
`jalr`	Jump and Link Register	CT		X		X		page 130
`jalrp`	Jump and Link Register Predict	CT		X		X		page 131

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
jr	Jump Register	CT		X		X		page 132
jrp	Jump Register Predict	CT		X		X		page 133
ld	Load	M		X			X	page 221
ld_add	Load and Add	M		X				page 234
ld1s	Load One Byte Signed	M		X			X	page 222
ld1s_add	Load One Byte Signed and Add	M		X				page 223
ld1u	Load One Byte Unsigned	M		X			X	page 224
ld1u_add	Load One Byte Unsigned and Add	M		X				page 225
ld2s	Load Two Bytes Signed	M		X			X	page 226
ld2s_add	Load Two Bytes Signed and Add	M		X				page 227
ld2u	Load Two Bytes Unsigned	M		X			X	page 228
ld2u_add	Load Two Bytes Unsigned and Add	M		X				page 229
ld4s	Load Four Bytes Signed	M		X			X	page 230
ld4s_add	Load Four Bytes Signed and Add	M		X				page 231
ld4u	Load Four Bytes Unsigned	M		X			X	page 232
ld4u_add	Load Four Bytes Unsigned and Add	M		X				page 234
ld_add	Load and Add	M		X				page 234
ldna	Load No Alignment Trap	M		X				page 235

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`ldna_add`	Load No Alignment Trap and Add	M		X				page 236
`ldnt`	Load Non-Temporal	M		X				page 237
`ldnt_add`	Load Non-Temporal and Add	M		X				page 250
`ldnt1s`	Load Non-Temporal One Byte Signed	M		X				page 238
`ldnt1s_add`	Load Non-Temporal One Byte Signed and Add	M		X				page 239
`ldnt1u`	Load Non-Temporal One Byte Unsigned	M		X				page 240
`ldnt1u_add`	Load Non-Temporal One Byte Unsigned and Add	M		X				page 241
`ldnt2s`	Load Non-Temporal Two Bytes Signed	M		X				page 242
`ldnt2s_add`	Load Non-Temporal Two Bytes Signed and Add	M		X				page 243
`ldnt2u`	Load Non-Temporal Two Bytes Unsigned	M		X				page 244
`ldnt2u_add`	Load Non-Temporal Two Bytes Unsigned and Add	M		X				page 245
`ldnt4s`	Load Non-Temporal Four Bytes Signed	M		X				page 246
`ldnt4s_add`	Load Non-Temporal Four Bytes Signed and Add	M		X				page 247
`ldnt4u`	Load Non-Temporal Four Bytes Unsigned	M		X				page 248
`ldnt4u_add`	Load Non-Temporal Four Bytes Unsigned and Add	M		X				page 249
`lnk`	Link	CT		X		X		page 134

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`mf`	Memory Fence	MM		X				page 205
`mfspr`	Move from Special Purpose Register Word	S		X				page 484
`mm`	Masked Merge	L	X					page 162
`mnz`	Mask Not Zero	L	X	X	X	X		page 163
`move`	Move	PS	X	X	X	X		page 312
`movei`	Move Immediate Word	PS	X	X	X	X		page 314
`moveli`	Move Long Immediate Word	PS	X	X				page 316
`mtspr`	Move to Special Purpose Register Word	S		X				page 485
`mul_hs_hs`	Multiply High Signed High Signed	ML	X		X			page 276
`mul_hs_hu`	Multiply High Signed High Unsigned	ML	X					page 277
`mul_hs_ls`	Multiply High Signed Low Signed	ML	X					page 278
`mul_hs_lu`	Multiply High Signed Low Unsigned	ML	X					page 279
`mul_hu_hu`	Multiply High Unsigned High Unsigned	ML	X		X			page 280
`mul_hu_ls`	Multiply High Unsigned Low Signed	ML	X					page 281
`mul_hu_lu`	Multiply High Unsigned Low Unsigned	ML	X					page 282
`mul_ls_ls`	Multiply Low Signed Low Signed	ML	X		X			page 283
`mul_ls_lu`	Multiply Low Signed Low Unsigned	ML	X					page 284

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`mul_lu_lu`	Multiply Low Unsigned Low Unsigned	ML	X		X			page 285
`mula_hs_hs`	Multiply Accumulate High Signed High Signed	ML	X		X			page 286
`mula_hs_hu`	Multiply Accumulate High Signed High Unsigned	ML	X					page 287
`mula_hs_ls`	Multiply Accumulate High Signed Low Signed	ML	X					page 288
`mula_hs_lu`	Multiply Accumulate High Signed Low Unsigned	ML	X					page 289
`mula_hu_hu`	Multiply Accumulate High Unsigned High Unsigned	ML	X		X			page 290
`mula_hu_ls`	Multiply Accumulate High Unsigned Low Signed	ML	X					page 291
`mula_hu_lu`	Multiply Accumulate High Unsigned Low Unsigned	ML	X					page 292
`mula_ls_ls`	Multiply Accumulate Low Signed Low Signed	ML	X		X			page 293
`mula_ls_lu`	Multiply Accumulate Low Signed Low Unsigned	ML	X					page 294
`mula_lu_lu`	Multiply Accumulate Low Unsigned Low Unsigned	ML	X		X			page 295
`mulax`	Multiply Accumulate and Extend	ML	X		X			page 296
`mulx`	Multiply and Extend	ML	X		X			page 297
`mz`	Mask Zero	L	X	X	X	X		page 165
`nap`	Nap	S		X				page 486

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
nop	Architectural No Operation	N	X	X	X	X		page 302
nor	Nor	L	X	X	X	X		page 167
or	Or	L	X	X	X	X		page 169
ori	Or Immediate	L	X	X				page 171
pcnt	Population Count	B	X		X			page 88
prefetch	Prefetch to L1 with No Faults	PS		X			X	page 317
prefetch_add_l1	Prefetch to L1 and Add with No Faults	PS		X				page 318
prefetch_add_l1_fault	Prefetch to L1 and Add with Faults	PS		X				page 319
prefetch_add_l2	Prefetch to L2 and Add with No Faults	PS		X				page 320
prefetch_add_l2_fault	Prefetch to L2 and Add with Faults	PS		X				page 321
prefetch_add_l3	Prefetch to L3 and Add with No Faults	PS		X				page 322
prefetch_add_l3_fault	Prefetch to L3 and Add with Faults	PS		X				page 323
prefetch_l1	Prefetch to L1 with No Faults	PS		X			X	page 324
prefetch_l1_fault	Prefetch to L1 with Faults	PS		X			X	page 325
prefetch_l2	Prefetch to L2 with No Faults	PS		X			X	page 326
prefetch_l2_fault	Prefetch to L2 with Faults	PS		X			X	page 327
prefetch_l3	Prefetch to L3 with No Faults	PS		X			X	page 328
prefetch_l3_fault	Prefetch to L3 with Faults	PS		X			X	page 329
raise	Raise Signal	PS		X				page 330

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`revbits`	Reverse Bits	B	X		X			page 89
`revbytes`	Reverse Bytes	B	X		X			page 90
`rotl`	Rotate Left	L	X	X	X	X		page 172
`rotli`	Rotate Left Immediate	L	X	X	X	X		page 174
`shl`	Shift Left	L	X	X	X	X		page 176
`shl16insli`	Shift left 16 Insert Long Immediate	A	X	X				page 59
`shl1add`	Shift Left One and Add	A	X	X	X	X		page 60
`shl1addx`	Shift Left One Add and Extend	A	X	X	X	X		page 62
`shl2add`	Shift Left Two Add	A	X	X	X	X		page 64
`shl2addx`	Shift Left Two Add and Extend	A	X	X	X	X		page 66
`shl3add`	Shift Left Three Add	A	X	X	X	X		page 68
`shl3addx`	Shift Left Three Add and Extend	A	X	X	X	X		page 70
`shli`	Shift Left Immediate	L	X	X	X	X		page 178
`shlx`	Shift Left and Extend	L	X	X				page 180
`shlxi`	Shift Left and Extend Immediate	L	X	X				page 181
`shrs`	Shift Right Signed	L	X	X	X	X		page 182
`shrsi`	Shift Right Signed Immediate	L	X	X	X	X		page 184
`shru`	Shift Right Unsigned	L	X	X	X	X		page 186
`shrui`	Shift Right Unsigned Immediate	L	X	X	X	X		page 188
`shrux`	Shift Right Unsigned and Extend	L	X	X				page 190

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`shruxi`	Shift Right Unsigned and Extend Immediate	L	X	X				page 191
`shufflebytes`	Shuffle Bytes	B	X					page 92
`st`	Store	M		X			X	page 251
`st_add`	Store and Add	M		X				page 258
`st1`	Store Byte	M		X			X	page 252
`st1_add`	Store Byte and Add	M		X				page 253
`st2`	Store Two bytes	M		X			X	page 254
`st2_add`	Store Two Bytes and Add	M		X				page 255
`st4`	Store Four Bytes	M		X			X	page 256
`st4_add`	Store Four Bytes and Add	M		X				page 257
`stnt`	Store Non-Temporal	M		X				page 259
`stnt_add`	Store Non-Temporal and Add	M		X				page 266
`stnt1`	Store Non-Temporal Byte	M		X				page 260
`stnt1_add`	Store Non-Temporal Byte and Add	M		X				page 261
`stnt2`	Store Non-Temporal Two bytes	M		X				page 262
`stnt2_add`	Store Non-Temporal Two Bytes and Add	M		X				page 263
`stnt4`	Store Non-Temporal Four Bytes	M		X				page 264
`stnt4_add`	Store Non-Temporal Four Bytes and Add	M		X				page 265
`sub`	Subtract	A	X	X	X	X		page 72
`subx`	Subtract and Extend	A	X	X	X	X		page 74

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`subxsc`	Subtract Signed Clamped and Extend	A	X	X				page 76
`swint0`	Software Interrupt 0	S		X				page 487
`swint1`	Software Interrupt 1	S		X				page 488
`swint2`	Software Interrupt 2	S		X				page 489
`swint3`	Software Interrupt 3	S		X				page 490
`tblidxb0`	Table Index Byte 0	L	X		X			page 192
`tblidxb1`	Table Index Byte 1	L	X		X			page 193
`tblidxb2`	Table Index Byte 2	L	X		X			page 194
`tblidxb3`	Table Index Byte 3	L	X		X			page 195
`v1add`	Vector One Byte Add	SM	X	X				page 334
`v1addi`	Vector One Byte Add Immediate	SM	X	X				page 336
`v1adduc`	Vector One Byte Add Unsigned Clamped	SM	X	X				page 337
`v1adiffu`	Vector One Byte Absolute Difference Unsigned	SM	X					page 339
`v1avgu`	Vector One Byte Average Unsigned	SM	X					page 340
`v1cmpeq`	Vector One Byte Set Equal To	SM	X	X				page 341
`v1cmpeqi`	Vector One Byte Set Equal To Immediate	SM	X	X				page 343
`v1cmples`	Vector One Byte Set Less Than or Equal	SM	X	X				page 344
`v1cmpleu`	Vector One Byte Set Less Than or Equal Unsigned	SM	X	X				page 346
`v1cmplts`	Vector One Byte Set Less Than	SM	X	X				page 348

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`v1cmpltsi`	Vector One Byte Set Less Than Immediate	SM	X	X				page 350
`v1cmpltu`	Vector One Byte Set Less Than Unsigned	SM	X	X				page 351
`v1cmpltui`	Vector One Byte Set Less Than Unsigned Immediate	SM	X	X				page 353
`v1cmpne`	Vector One Byte Set Not Equal To	SM	X	X				page 354
`v1ddotpu`	Vector One Byte Dual Dot Product Unsigned	SM	X					page 356
`v1ddotpua`	Vector One Byte Dual Dot Product Unsigned and Accumulate	SM	X					page 357
`v1ddotpus`	Vector One Byte Dual Dot Product Unsigned Signed	SM	X					page 358
`v1ddotpusa`	Vector One Byte Dual Dot Product Unsigned Signed and Accumulate	SM	X					page 359
`v1dotp`	Vector One Byte Dot Product	SM	X					page 360
`v1dotpa`	Vector One Byte Dot Product and Accumulate	SM	X					page 361
`v1dotpu`	Vector One Byte Dot Product Unsigned	SM	X					page 362
`v1dotpua`	Vector One Byte Dot Product Unsigned and Accumulate	SM	X					page 363
`v1dotpus`	Vector One Byte Dot Product Unsigned Signed	S	X					page 364

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`v1dotpusa`	Vector One Byte Dot Product Unsigned Signed and Accumulate	S	X					page 365
`v1int_h`	Vector One Byte Interleave High	SM	X	X				page 366
`v1int_l`	Vector One Byte Interleave Low	SM	X	X				page 368
`v1maxu`	Vector One Byte Maximum Unsigned	SM	X	X				page 370
`v1maxui`	Vector One Byte Maximum Unsigned Immediate	SM	X	X				page 372
`v1minu`	Vector One Byte Minimum Unsigned	SM	X	X				page 370
`v1minui`	Vector One Byte Minimum Unsigned Immediate	SM	X	X				page 374
`v1mnz`	Vector One Byte Mask Not Zero	SM	X	X				page 375
`v1multu`	Vector One Byte Multiply and Truncate Unsigned	SM	X					page 376
`v1mulu`	Vector One Byte Multiply Unsigned	SM	X					page 377
`v1mulus`	Vector One Byte Multiply Unsigned Signed	SM	X					page 378
`v1mz`	Vector One Byte Mask Zero	SM	X	X				page 379
`v1sadau`	Vector One Byte Sum of Absolute Difference Accumulate Unsigned	SM	X					page 380
`v1sadu`	Vector One Byte Sum of Absolute Difference Unsigned	SM	X					page 381
`v1shl`	Vector One Byte Shift Left	SM	X	X				page 382

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
v1shli	Vector One Byte Shift Left Immediate	SM	X	X				page 383
v1shrs	Vector One Byte Shift Right Signed	SM	X	X				page 385
v1shrsi	Vector One Byte Shift Right Signed Immediate	SM	X	X				page 387
v1shru	Vector One Byte Shift Right Unsigned	SM	X	X				page 389
v1shrui	Vector One Byte Shift Right Unsigned Immediate	SM	X	X				page 390
v1sub	Vector One Byte Subtract	SM	X	X				page 392
v1subuc	Vector One Byte Subtract Unsigned Clamped	SM	X	X				page 393
v2add	Vector Two Byte Add	SM	X	X				page 395
v2addi	Vector Two Byte Add Immediate	SM	X	X				page 396
v2addsc	Vector Two Byte Add Signed Clamped	SM	X	X				page 397
v2adiffs	Vector Two Byte Absolute Difference Signed	SM	X					page 399
v2avgs	Vector Two Byte Average Signed	SM	X					page 400
v2cmpeq	Vector Two Byte Set Equal To	SM	X	X				page 401
v2cmpeqi	Vector Two Byte Set Equal To Immediate	SM	X	X				page 403
v2cmples	Vector Two Byte Set Less Than or Equal	SM	X	X				page 404
v2cmpleu	Vector Two Byte Set Less Than or Equal Unsigned	SM	X	X				page 406

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
v2cmplts	Vector Two Byte Set Less Than	SM	X	X				page 408
v2cmpltsi	Vector Two Byte Set Less Than Immediate	SM	X	X				page 410
v2cmpltu	Vector Two Byte Set Less Than Unsigned	SM	X	X				page 411
v2cmpltui	Vector Two Byte Set Less Than Unsigned Immediate	SM	X	X				page 413
v2cmpne	Vector Two Byte Set Not Equal To	SM	X	X				page 414
v2dotp	Vector Two Byte Dot Product	SM	X					page 416
v2dotpa	Vector Two Byte Dot Product and Accumulate	SM	X					page 417
v2int_h	Vector Two Byte Interleave High	SM	X	X				page 418
v2int_l	Vector Two Byte Interleave Low	SM	X	X				page 420
v2maxs	Vector Two Byte Maximum Signed	SM	X	X				page 422
v2maxsi	Vector Two Byte Maximum Signed Immediate	SM	X	X				page 423
v2mins	Vector Two Byte Minimum Signed	SM	X	X				page 424
v2minsi	Vector Two Byte Minimum Signed Immediate	SM	X	X				page 425
v2mnz	Vector Two Byte Mask Not Zero	SM	X	X				page 426
v2mulfsc	Vector Two Byte Multiply Fixed point Signed Clamped	SM	X					page 427

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
`v2muls`	Vector Two Byte Multiply Signed	SM	X					page 428
`v2mults`	Vector Two Byte Multiply and Truncate Signed	SM	X					page 429
`v2mz`	Vector Two Byte Mask Zero	SM	X	X				page 430
`v2packh`	Vector Two Bytes Pack High Byte	SM	X	X				page 431
`v2packl`	Vector Two Byte Pack Low Byte	SM	X	X				page 433
`v2packuc`	Vector Two Byte Pack Unsigned Clamped	SM	X	X				page 435
`v2sadas`	Vector Two Byte Sum of Absolute Difference Accumulate Signed	SM	X					page 437
`v2sadau`	Vector Two Byte Sum of Absolute Difference Accumulate Unsigned	SM	X					page 438
`v2sads`	Vector Two Byte Sum of Absolute Difference Signed	SM	X					page 439
`v2sadu`	Vector Two Byte Sum of Absolute Difference Unsigned	SM	X					page 440
`v2shl`	Vector Two Byte Shift Left	SM	X	X				page 441
`v2shli`	Vector Two Byte Shift Left Immediate	SM	X	X				page 443
`v2shlsc`	Vector Two Byte Shift Left Signed Clamped	SM	X	X				page 445
`v2shrs`	Vector Two Byte Shift Right Signed	SM	X	X				page 447
`v2shrsi`	Vector Two Byte Shift Right Signed Immediate	SM	X	X				page 448

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots X0	X1	YO	Y1	Y2	Page
`v2shru`	Vector Two Byte Shift Right Unsigned	SM	X	X				page 450
`v2shrui`	Vector Two Byte Shift Right Unsigned Immediate	SM	X	X				page 452
`v2sub`	Vector Two Byte Subtract	SM	X	X				page 454
`v2subsc`	Vector Two Byte Subtract Signed Clamped	SM	X	X				page 456
`v4add`	Vector Four Byte Add	SM	X	X				page 458
`v4addsc`	Vector Four Byte Add Signed Clamped	SM	X	X				page 459
`v4int_h`	Vector Four Byte Interleave High	SM	X	X				page 461
`v4int_l`	Vector Four Byte Interleave Low	SM	X	X				page 463
`v4packsc`	Vector Four Byte Pack Signed Clamped	SM	X	X				page 465
`v4shl`	Vector Four Byte Shift Left	SM	X	X				page 467
`v4shlsc`	Vector Four Byte Shift Left Signed Clamped	SM	X	X				page 469
`v4shrs`	Vector Four Byte Shift Right Signed	SM	X	X				page 471
`v4shru`	Vector Four Byte Shift Right Unsigned	SM	X	X				page 473
`v4sub`	Vector Four Byte Subtract	SM	X	X				page 475
`v4subsc`	Vector Four Byte Subtract Signed Clamped	SM	X	X				page 477
`wh64`	Write Hint 64 Bytes	MM		X				page 206
`xor`	Exclusive Or	L	X	X	X	X		page 196

Table 3. Master List of Main Processor Instructions (continued)

Instruction	Description	Type	Slots					Page
			X0	X1	YO	Y1	Y2	
xori	Exclusive Or Immediate	L	X	X				page 198

2.1.4 Pseudo Instructions

Tilera's assembler supports several pseudo-instructions for the convenience of the programmer. Each of these instructions shares an encoding with a standard ISA instruction.

Table 2-1. Pseudo Instructions

Pseudo Instruction	Canonical Form
move dst,src	or dst, src, zero
movei dst,simm8	addi dst, zero, simm8
movelidst,simm16	addli dst, zero, simm16
prefetch[1]src	ld1u zero, src
prefetch_l1	ld1u zero, src
prefetch_add_l1	ld1u_add zero, src
prefetch_l1_fault	ld1s zero, src
prefetch_add_l1_fault	ld1s_add zero, src
prefetch_l2	ld2u zero, src
prefetch_add_l2	ld2u_add zero, src
prefetch_l2_fault	ld2s zero, src
prefetch_add_l2_fault	ld2s_add zero, src
prefetch_l3	ld4u zero, src
prefetch_add_l3	ld4u_add zero, src
prefetch_l3_fault	ld4s zero, src
prefetch_add_l3_fault	ld4s_add zero, src
bpt[2]	ill

Table 2-1. Pseudo Instructions (continued)

Pseudo Instruction	Canonical Form
`raise`[3]	`ill`
`info simm8`	`andi zero, zero,simm8`
`infol simm16`	`shl16insli zero,zero,simm16`

1 For performance reasons, loads to the zero register do not result in the register file being written. Such instructions are killed entirely if they would cause DTLB_MISS or DTLB_ACCESS interrupts. The TILE architecture does not guarantee that every prefetch instruction will cause the caches to be loaded. Thus prefetch (indeed, any load to the zero register) should be considered merely a hint to the hardware.

2 The TILE architecture does not provide an explicit breakpoint instruction. Instead, bpt is encoded as an illegal instruction with non-zero values in the implicit immediate fields. Thus bpt does not have exactly the same hardware encoding as the ill or raise instruction.

3 The TILE-Gx architecture does not provide an explicit synchronous exception instruction. Instead raise is encoded as an illegal instruction with non-zero values in the implicit immediate fields. Thus raise does not have exactly the same hardware encoding as the ill or bpt instruction.

`INFO` operations are generated by the compiler and are used to convey information about the state of the stack frame at various points in the code of a function. The backtrace library interprets these operations when performing stack unwinding.

In order to perform stack unwinding, the backtrace library requires that code conform to the stack frame conventions specified in the ABI. In the presence of compiler optimizations, however, the code may deviate from these conventions. In this case, the compiler automatically inserts `INFO` operations in the code to compensate.

Intrinsics, including the `INFO` operation, are a set of functions whose names have the format `__insn_xxxx()`, where *xxxx* is an instruction in the ISA.

CHAPTER 3 ARITHMETIC INSTRUCTIONS

3.1 Overview

The following sections provide detailed descriptions of arithmetic instructions listed alphabetically.

add	Add
addi	Add Immediate
addli	Add Long Immediate
addx	Add and Extend
addxi	Add and Extend Immediate
addxli	Add and Extend Long Immediate
addxsc	Add Signed Clamped and Extend
shl16insli	Shift Left 16 Insert Long Immediate
shl1add	Shift Left One and Add
shl1addx	Shift Left One Add and Extend
shl2add	Shift Left Two Add
shl2addx	Shift Left Two Add and Extend
shl3add	Shift Left Three Add
shl3addx	Shift Left Three Add and Extend
sub	Subtract
subx	Subtract and Extend
subxsc	Subtract Signed Clamped and Extend

3.2 Instructions

Arithmetic instructions are described in the sections that follow.

add

Add

Syntax

```
add Dest, SrcA, SrcB
```

Example

```
add r5, r6, r7
```

Description

Adds the two operands together.

Functional Description

```
rf[Dest] = rf[SrcA] + rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

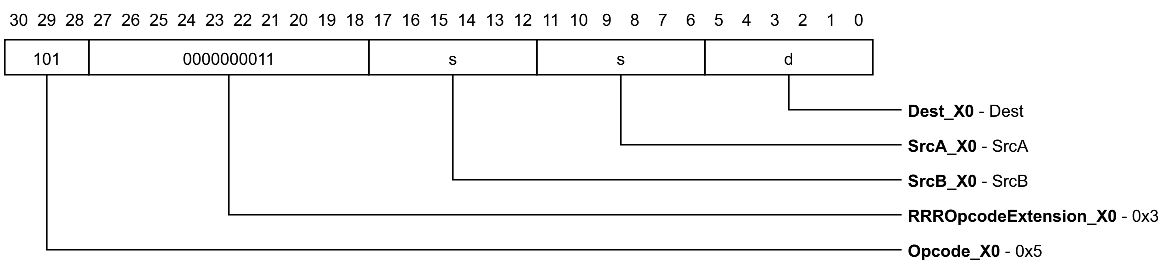

Figure 3-29: add in X0 Bits Encoding

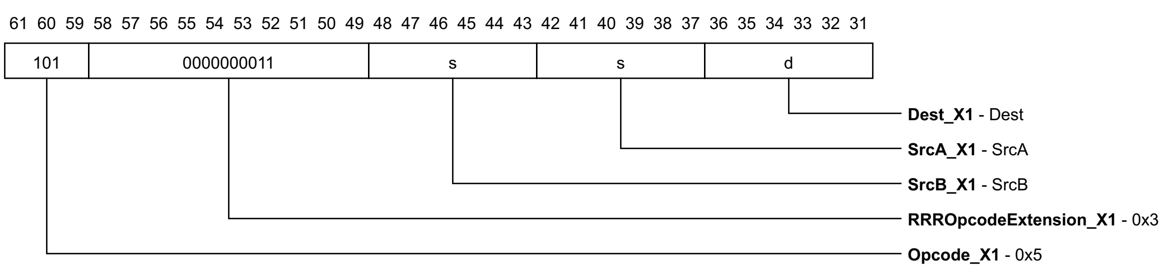

Figure 3-30: add in X1 Bits Encoding

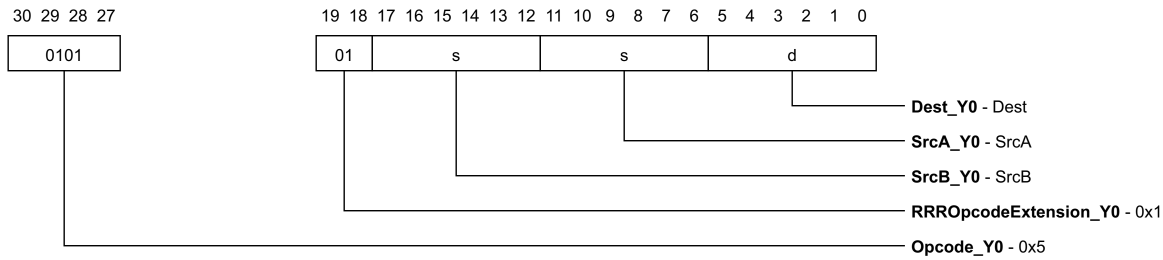

Figure 3-31: add in Y0 Bits Encoding

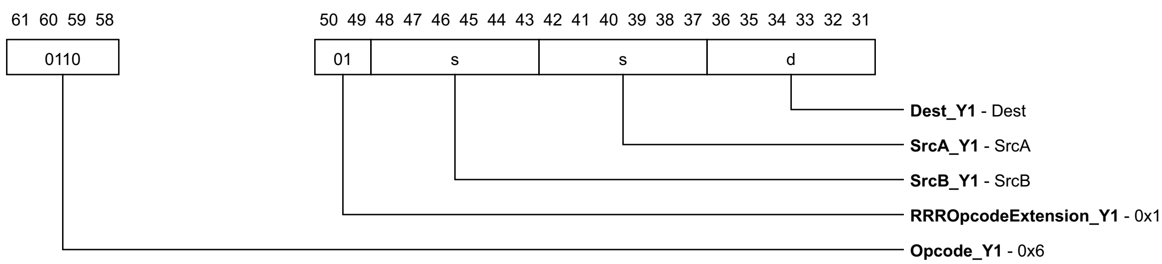

Figure 3-32: add in Y1 Bits Encoding

addi

Add Immediate

Syntax

```
addi Dest, SrcA, Imm8
```

Example

```
addi r5, r6, 5
```

Description

Adds one operand with a sign extended immediate.

Functional Description

```
rf[Dest] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

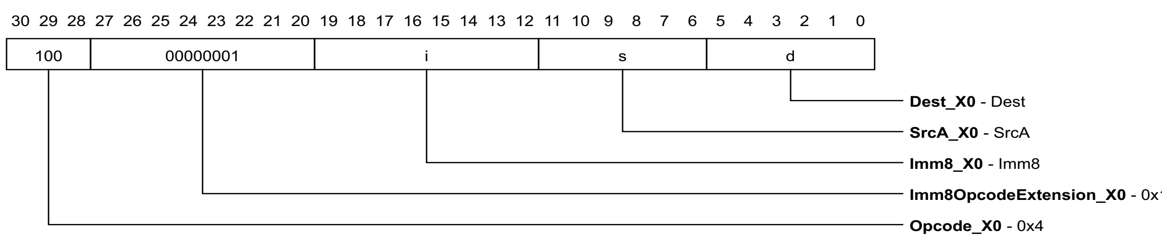

Figure 3-33: addi in X0 Bits Encoding

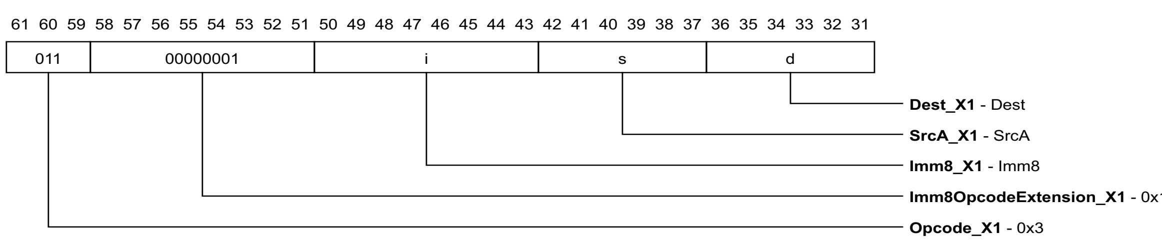

Figure 3-34: addi in X1 Bits Encoding

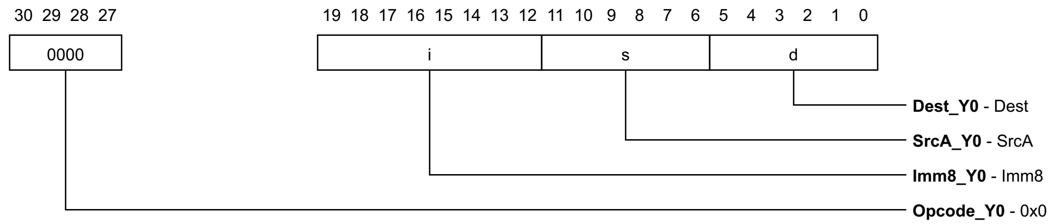

Figure 3-35: addi in Y0 Bits Encoding

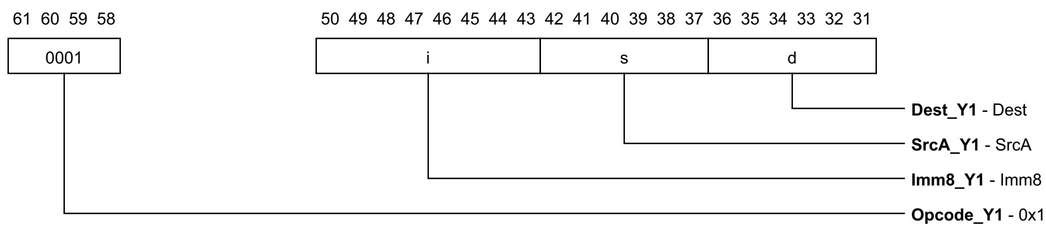

Figure 3-36: addi in Y1 Bits Encoding

addli

Add Long Immediate

Syntax

```
addli Dest, SrcA, Imm16
```

Example

```
addli r5, r6, 0x1234
```

Description

Adds one operand with a sign extended long immediate.

Functional Description

```
rf[Dest] = rf[SrcA] + signExtend16 (Imm16);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

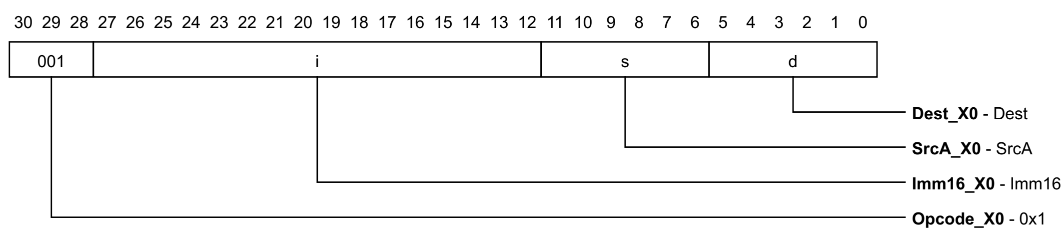

Figure 3-37: addli in X0 Bits Encoding

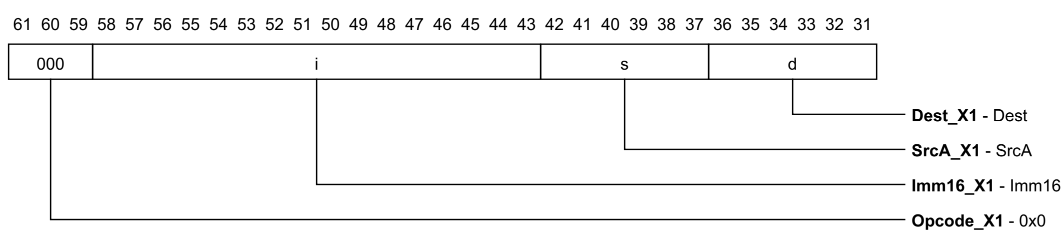

Figure 3-38: addli in X1 Bits Encoding

addx

Add and Extend

Syntax

```
addx Dest, SrcA, SrcB
```

Example

```
addx r5, r6, r7
```

Description

Adds the two 4-byte operands together and sign-extends the result.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] + rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

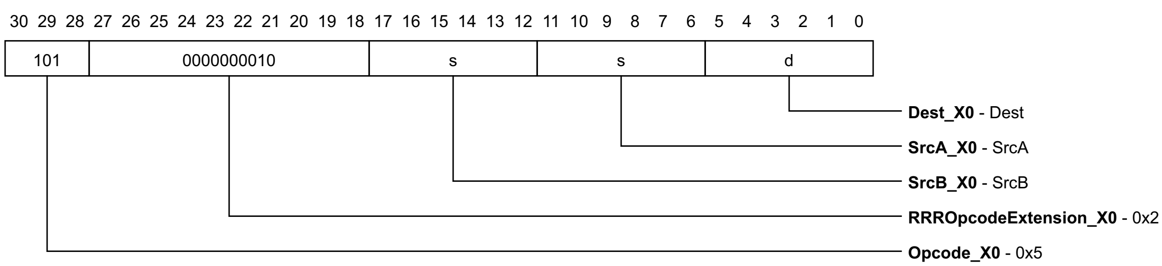

Figure 3-39: addx in X0 Bits Encoding

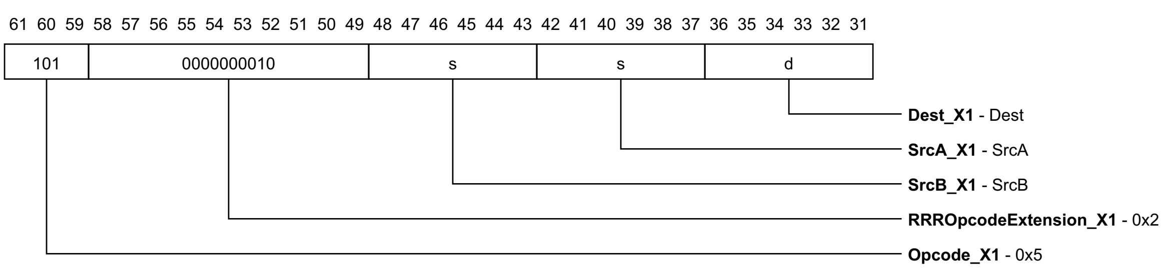

Figure 3-40: addx in X1 Bits Encoding

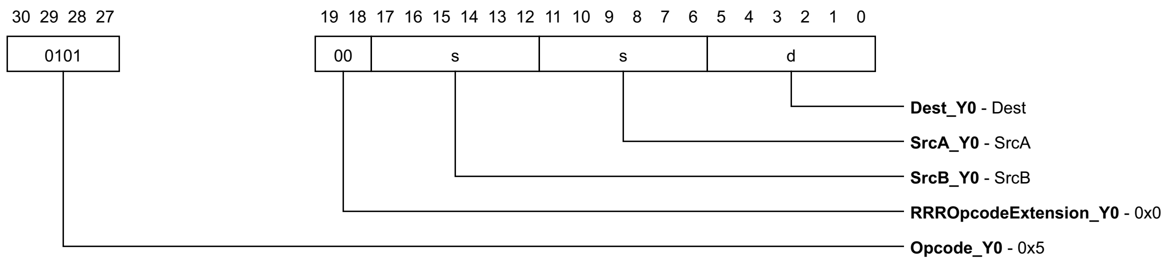

Figure 3-41: addx in Y0 Bits Encoding

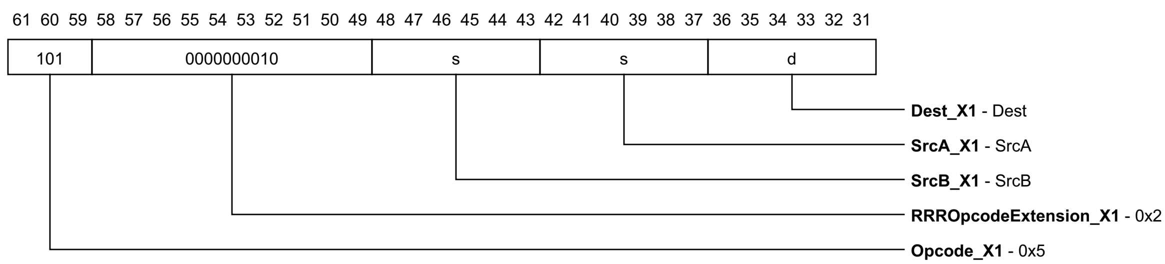

Figure 3-42: addx in Y1 Bits Encoding

addxi

Add and Extend Immediate

Syntax

```
addxi Dest, SrcA, Imm8
```

Example

```
addxi r5, r6, 5
```

Description

Adds one 4-byte operand with a sign extended immediate and sign-extends the result.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] + signExtend8 (Imm8));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

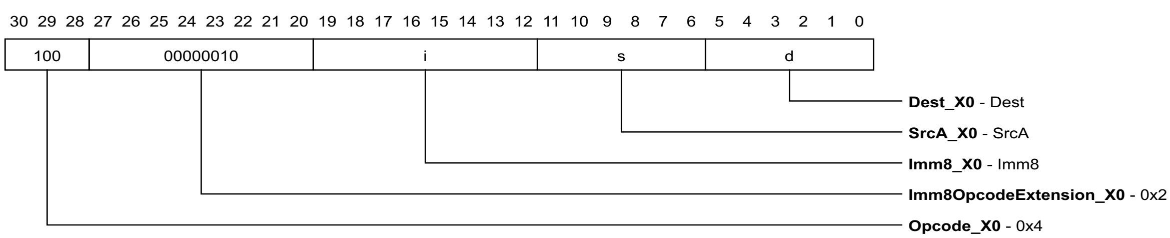

Figure 3-43: addxi in X0 Bits Encoding

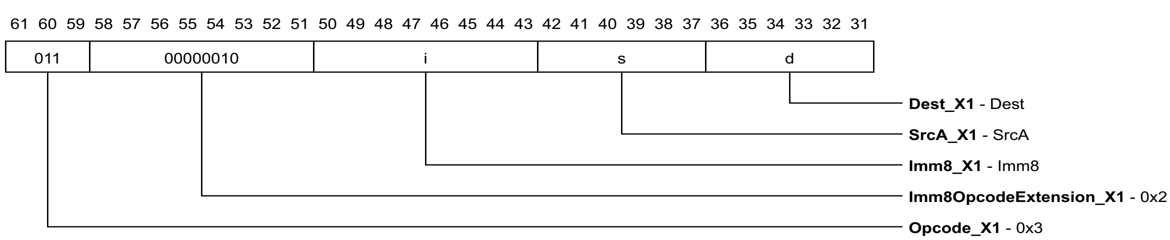

Figure 3-44: addxi in X1 Bits Encoding

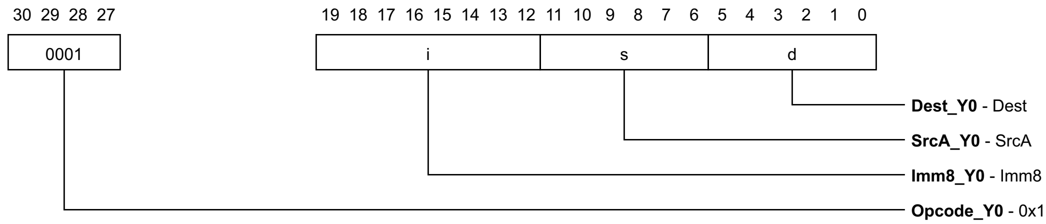

Figure 3-45: addxi in Y0 Bits Encoding

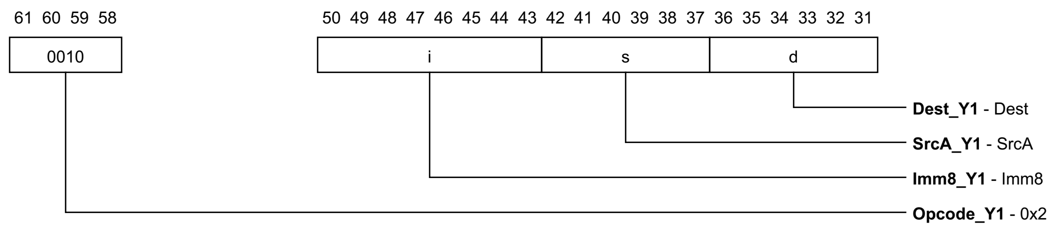

Figure 3-46: addxi in Y1 Bits Encoding

addxli

Add and Extend Long Immediate

Syntax

```
addxli Dest, SrcA, Imm16
```

Example

```
addxli r5, r6, 0x1234
```

Description

Adds one 4-byte operand with a sign extended long immediate and sign-extends the result.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] + signExtend16 (Imm16));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

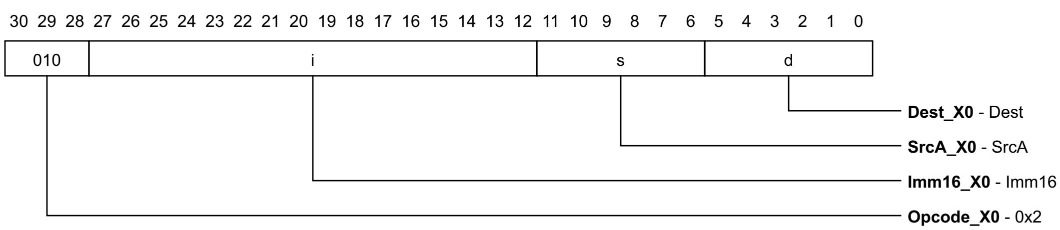

Figure 3-47: addxli in X0 Bits Encoding

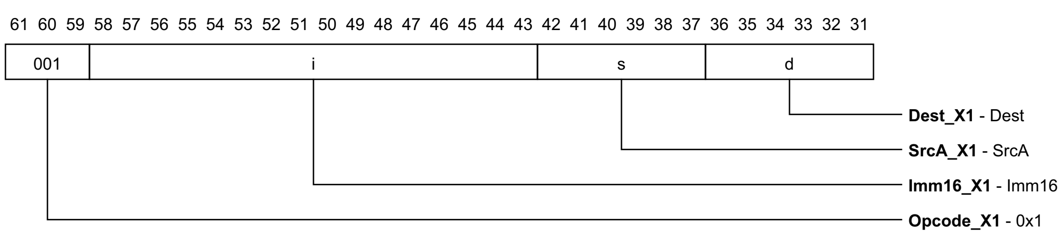

Figure 3-48: addxli in X1 Bits Encoding

addxsc

Add Signed Clamped and Extend

Syntax

```
addxsc Dest, SrcA, SrcB
```

Example

```
addxsc r5, r6, r7
```

Description

Adds two 4-byte operands together saturating the result at the minimum negative value or the maximum positive 4-byte value. The result is then sign-extended.

Functional Description

```
rf[Dest] =
signExtend32 (signed_saturate32
      (signExtend32 (rf[SrcA]) + signExtend32 (rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

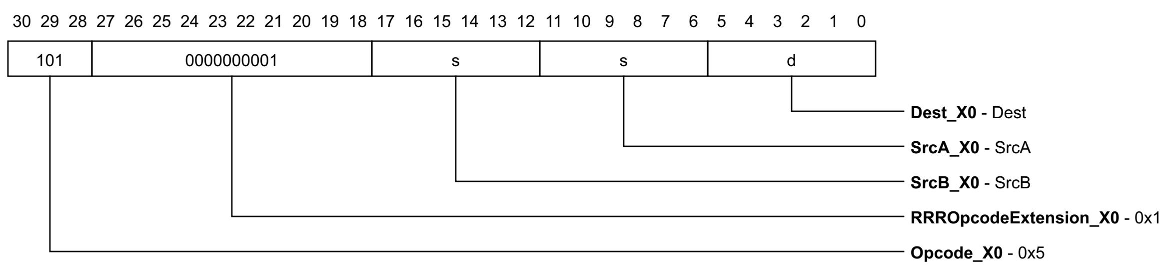

Figure 3-49: addxsc in X0 Bits Encoding

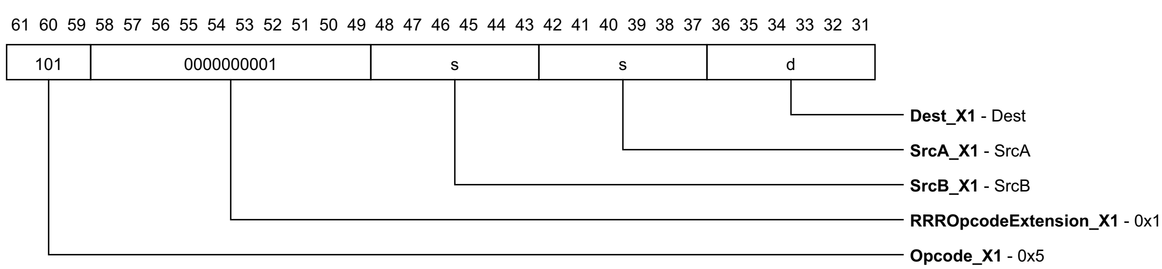

Figure 3-50: addxsc in X1 Bits Encoding

shl16insli

Shift Left 16 Insert Long Immediate

Syntax

```
shl16insli Dest, SrcA, Imm16
```

Example

```
shl16insli r5, r6, 0x1234
```

Description

Shifts the first source operand 16 bits left and inserts the 16 bit long immediate into least significant bits.

Functional Description

```
rf[Dest] = (rf[SrcA] << 16) | (Imm16 & MASK16);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

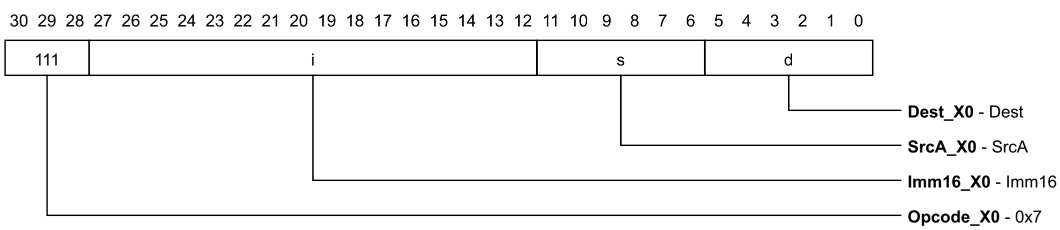

Figure 3-51: shl16insli in X0 Bits Encoding

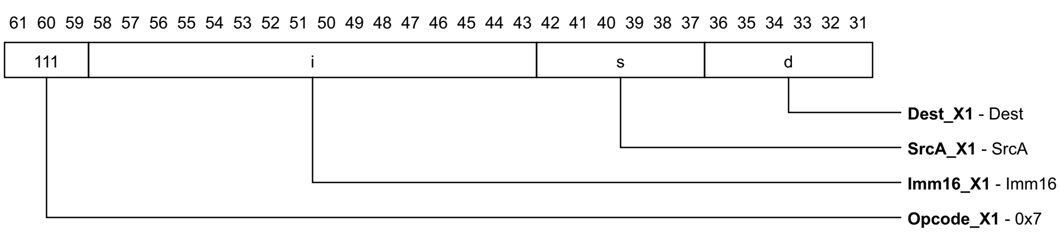

Figure 3-52: shl16insli in X1 Bits Encoding

shl1add

Shift Left One and Add

Syntax

```
shl1add Dest, SrcA, SrcB
```

Example

```
shl1add r5, r6, r7
```

Description

Shifts the first operand left by one bit and then adds the second source operand.

Functional Description

```
rf[Dest] = (rf[SrcA] << 1) + rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

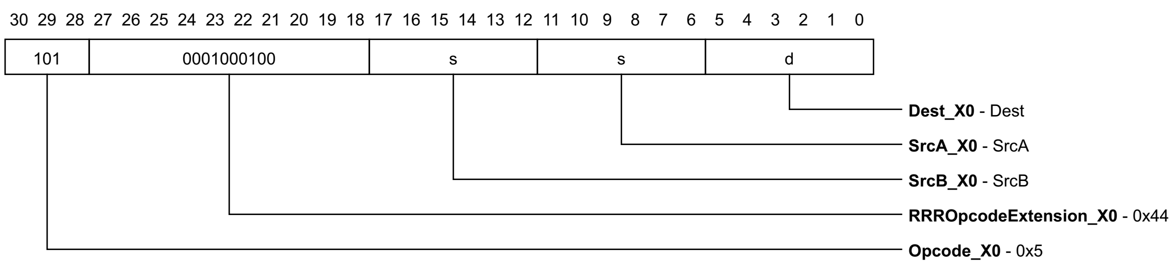

Figure 3-53: shl1add in X0 Bits Encoding

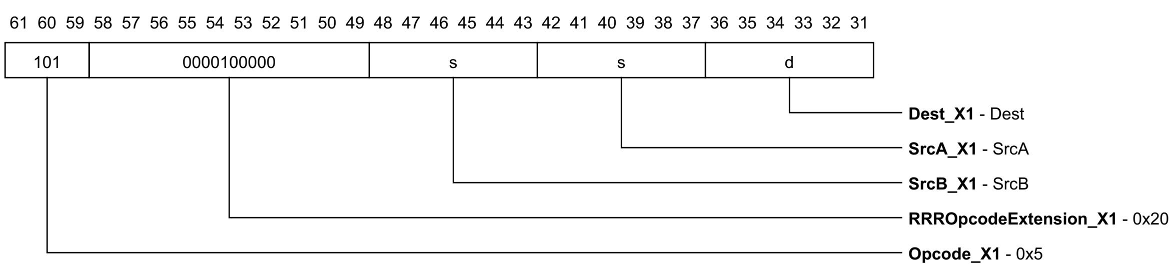

Figure 3-54: shl1add in X1 Bits Encoding

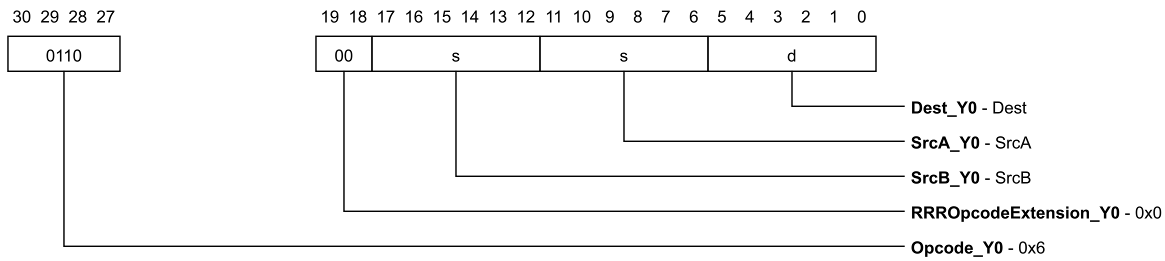

Figure 3-55: shl1add in Y0 Bits Encoding

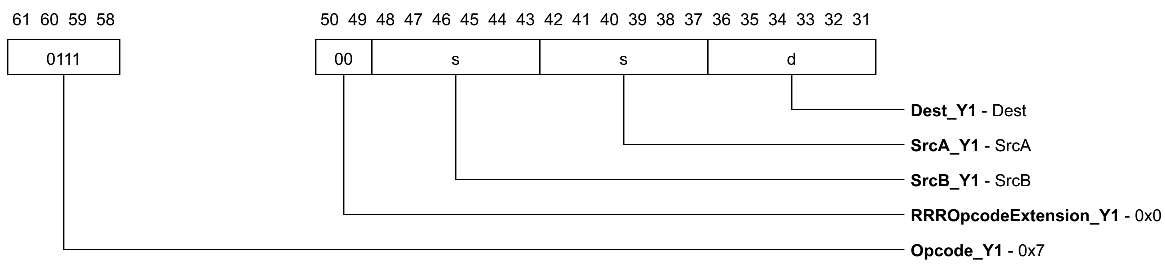

Figure 3-56: shl1add in Y1 Bits Encoding

shl1addx

Shift Left One Add and Extend

Syntax

```
shl1addx Dest, SrcA, SrcB
```

Example

```
shl1addx r5, r6, r7
```

Description

Shifts the first 4-byte operand left by one bit and then adds the second 4-byte source operand. The 4-byte result is sign-extended.

Functional Description

```
rf[Dest] = signExtend32 ((rf[SrcA] << 1) + rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

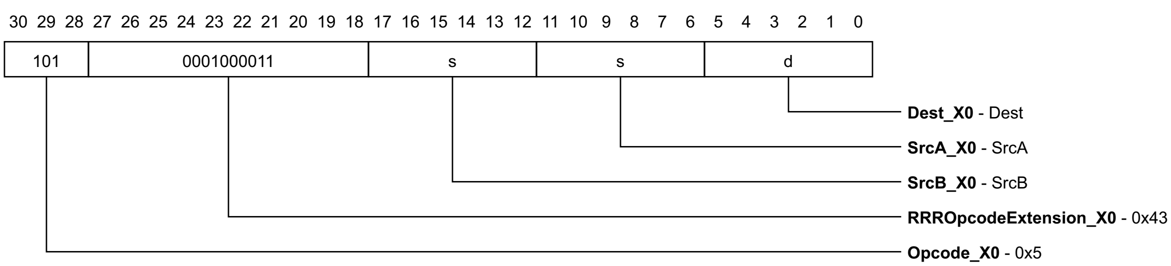

Figure 3-57: shl1addx in X0 Bits Encoding

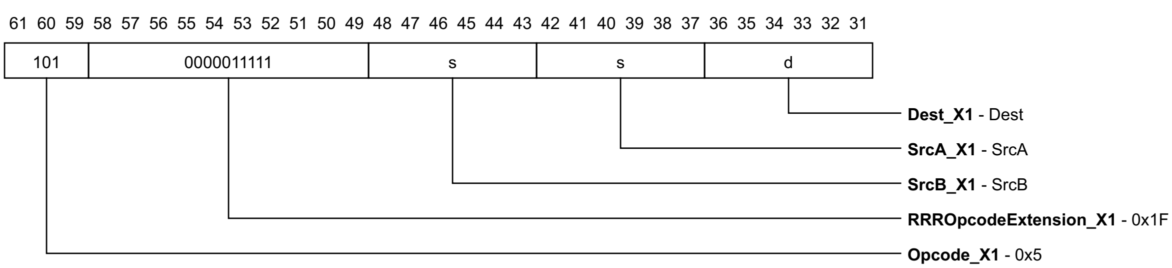

Figure 3-58: shl1addx in X1 Bits Encoding

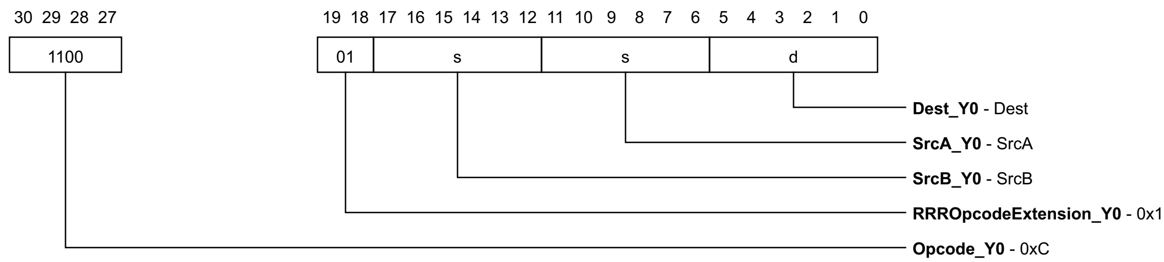

Figure 3-59: shl1addx in Y0 Bits Encoding

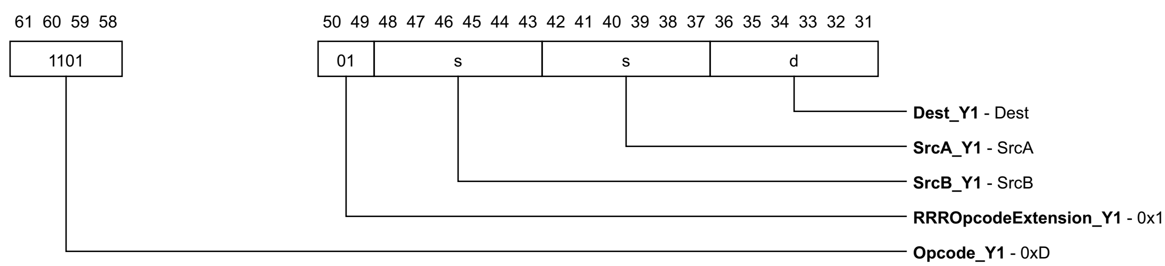

Figure 3-60: shl1addx in Y1 Bits Encoding

shl2add

Shift Left Two Add

Syntax

```
shl2add Dest, SrcA, SrcB
```

Example

```
shl2add r5, r6, r7
```

Description

Shifts the first operand left by two bits and then adds the second source operand.

Functional Description

```
rf[Dest] = (rf[SrcA] << 2) + rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

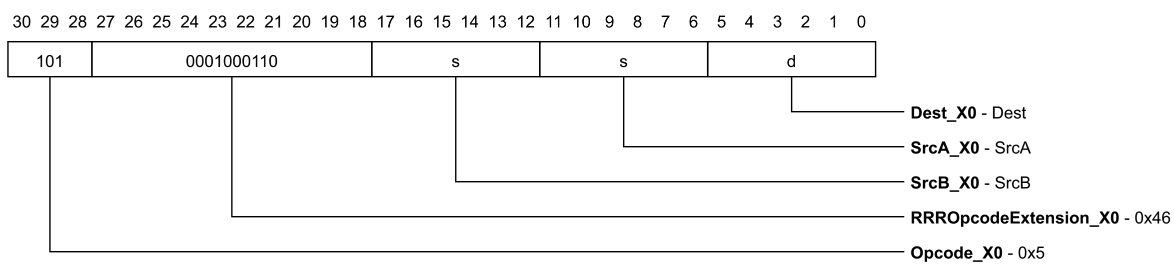

Figure 3-61: shl2add in X0 Bits Encoding

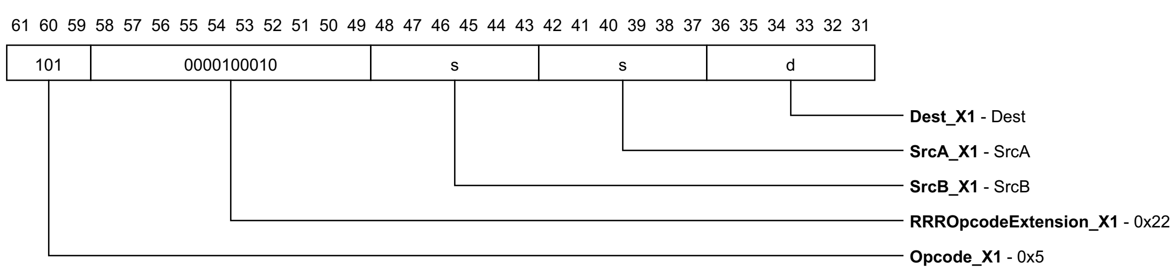

Figure 3-62: shl2add in X1 Bits Encoding

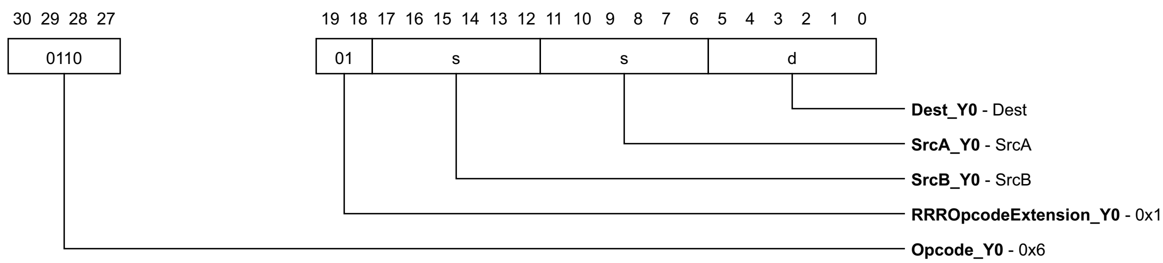

Figure 3-63: shl2add in Y0 Bits Encoding

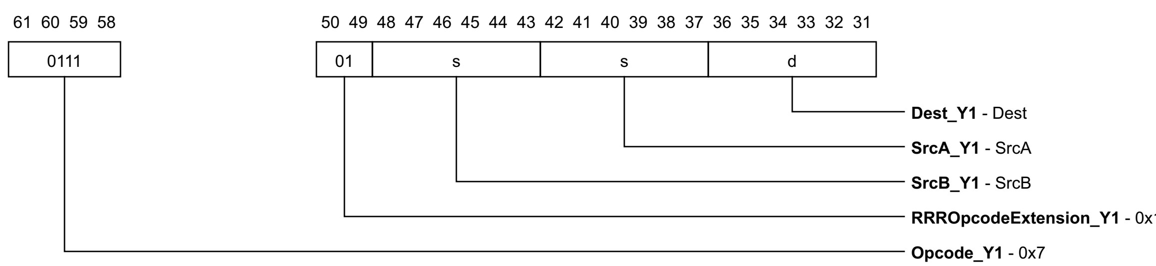

Figure 3-64: shl2add in Y1 Bits Encoding

shl2addx

Shift Left Two Add and Extend

Syntax

```
shl2addx Dest, SrcA, SrcB
```

Example

```
shl2addx r5, r6, r7
```

Description

Shifts the first 4-byte operand left by two bits and then adds the second 4-byte source operand. The 4-byte result is sign-extended.

Functional Description

```
rf[Dest] = signExtend32 ((rf[SrcA] << 2) + rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

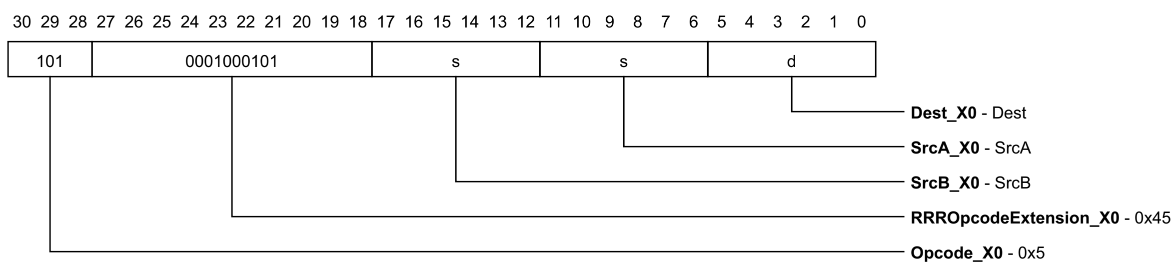

Figure 3-65: shl2addx in X0 Bits Encoding

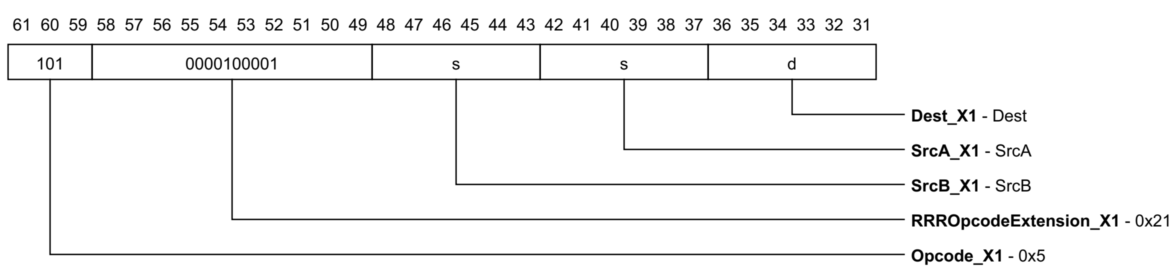

Figure 3-66: shl2addx in X1 Bits Encoding

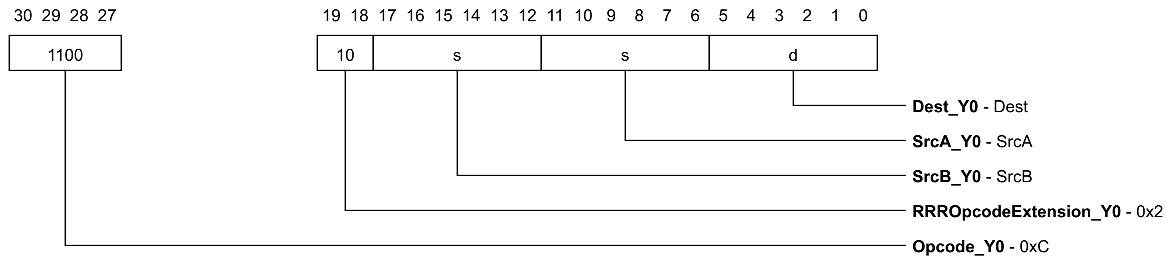

Figure 3-67: shl2addx in Y0 Bits Encoding

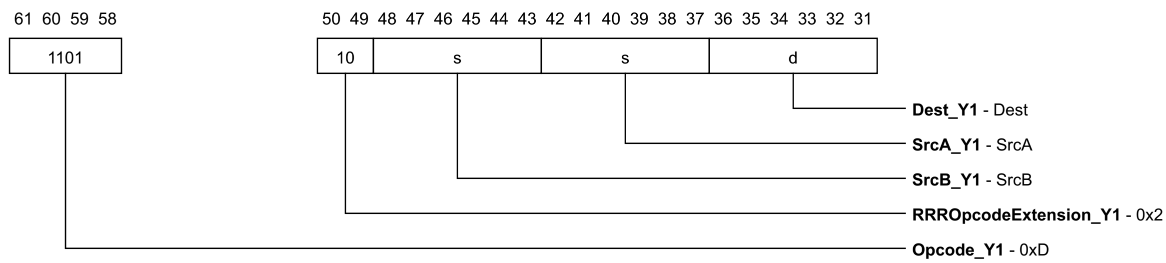

Figure 3-68: shl2addx in Y1 Bits Encoding

shl3add

Shift Left Three Add

Syntax

```
shl3add Dest, SrcA, SrcB
```

Example

```
shl3add r5, r6, r7
```

Description

Shifts the first operand left by three bits and then adds the second source operand.

Functional Description

```
rf[Dest] = (rf[SrcA] << 3) + rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

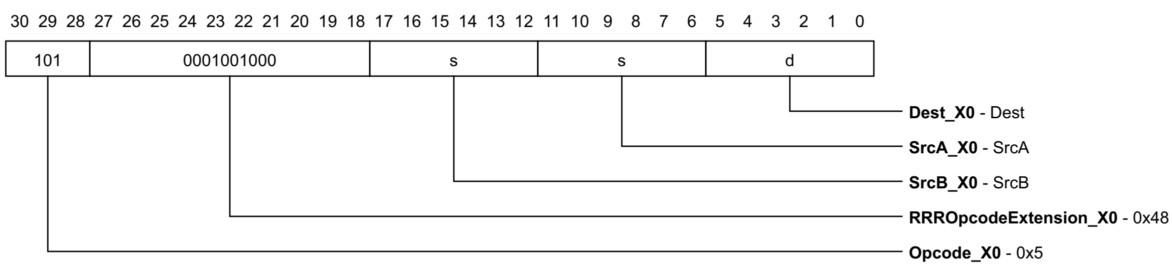

Figure 3-69: shl3add in X0 Bits Encoding

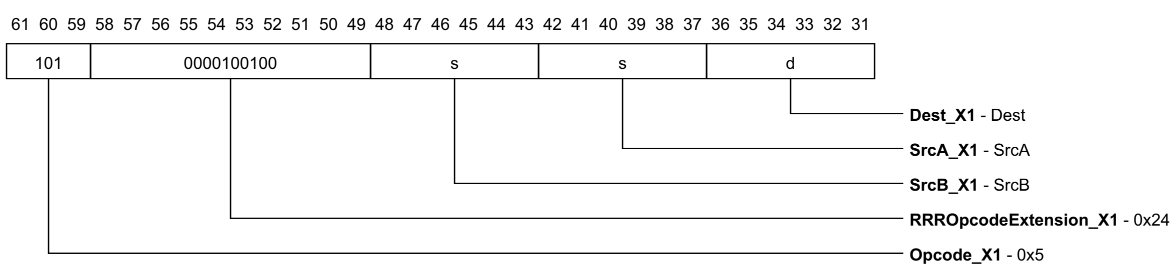

Figure 3-70: shl3add in X1 Bits Encoding

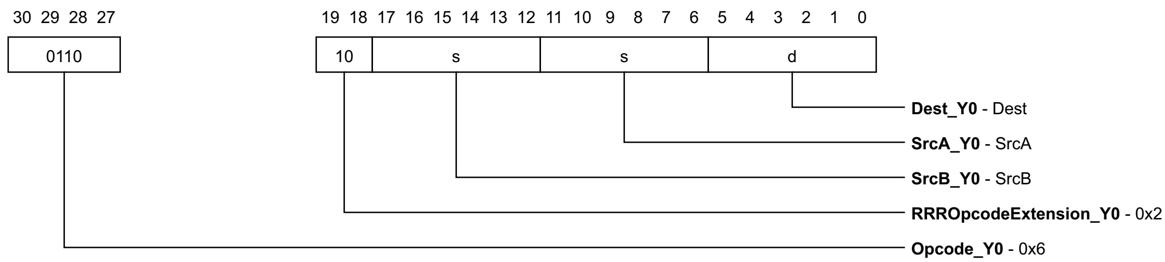

Figure 3-71: shl3add in Y0 Bits Encoding

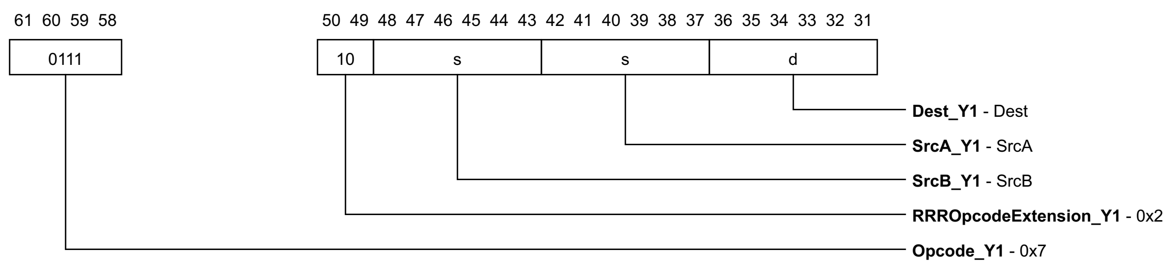

Figure 3-72: shl3add in Y1 Bits Encoding

shl3addx

Shift Left Three Add and Extend

Syntax

```
shl3addx Dest, SrcA, SrcB
```

Example

```
shl3addx r5, r6, r7
```

Description

Shifts the first 4-byte operand left by three bits and then adds the second 4-byte source operand. The 4-byte result is sign-extended.

Functional Description

```
rf[Dest] = signExtend32 ((rf[SrcA] << 3) + rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

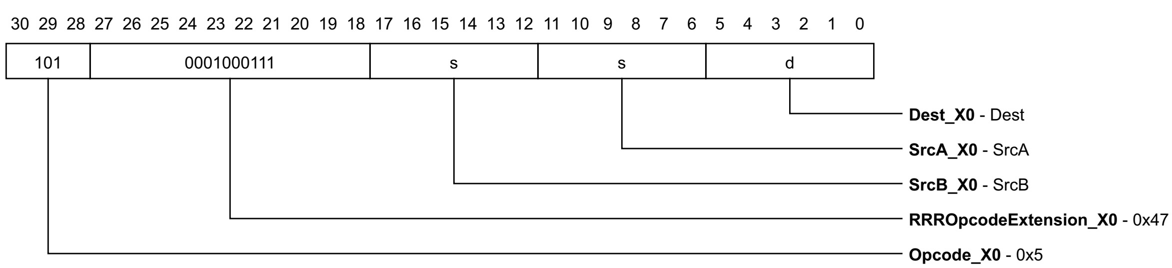

Figure 3-73: shl3addx in X0 Bits Encoding

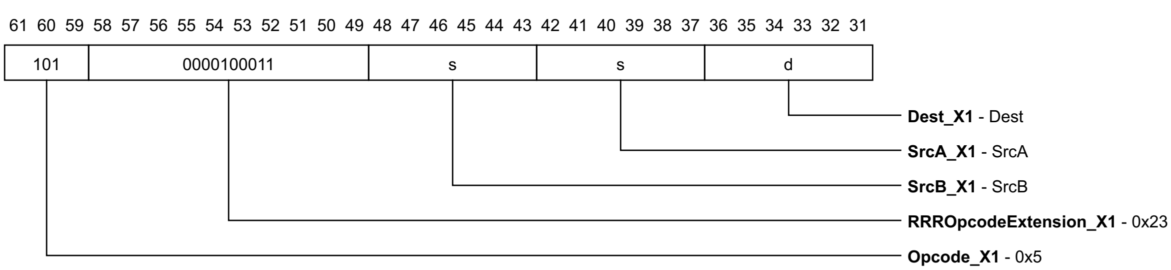

Figure 3-74: shl3addx in X1 Bits Encoding

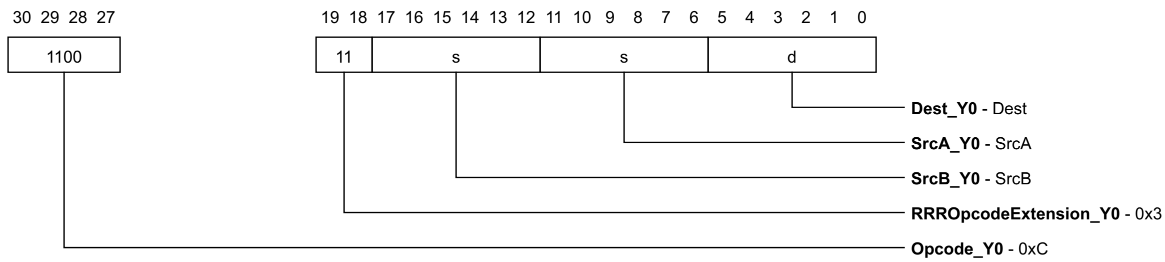

Figure 3-75: shl3addx in Y0 Bits Encoding

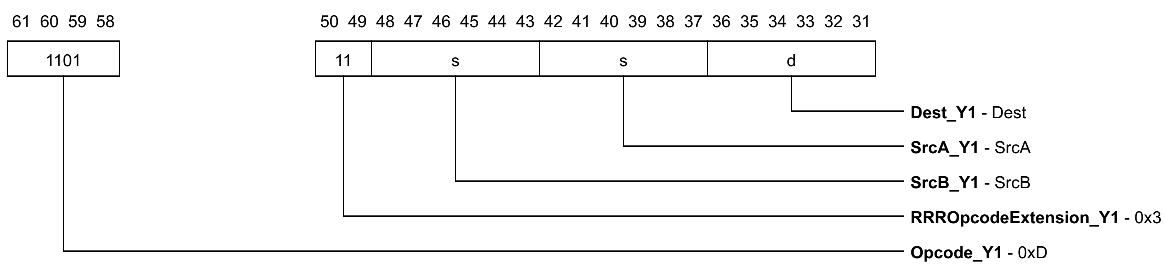

Figure 3-76: shl3addx in Y1 Bits Encoding

sub

Subtract

Syntax

```
sub Dest, SrcA, SrcB
```

Example

```
sub r5, r6, r7
```

Description

Subtracts the second operand from the first.

Functional Description

```
rf[Dest] = rf[SrcA] - rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

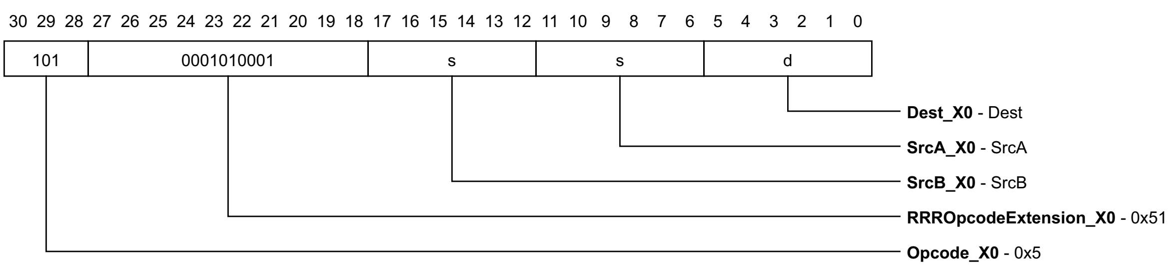

Figure 3-77: sub in X0 Bits Encoding

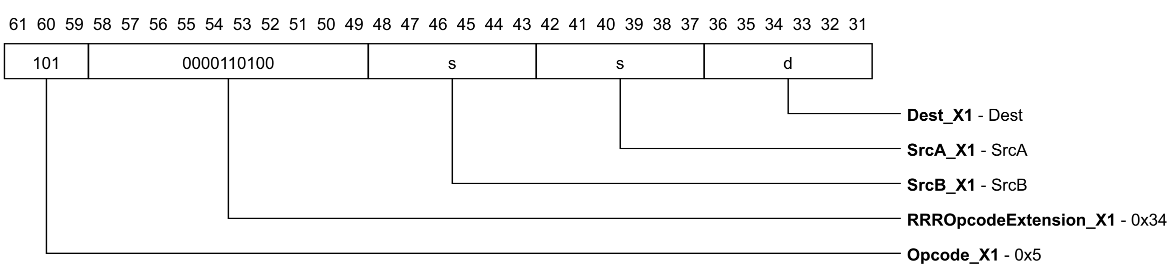

Figure 3-78: sub in X1 Bits Encoding

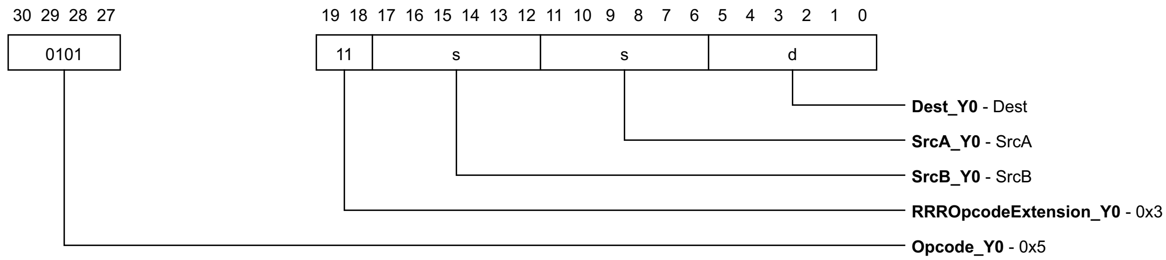

Figure 3-79: sub in Y0 Bits Encoding

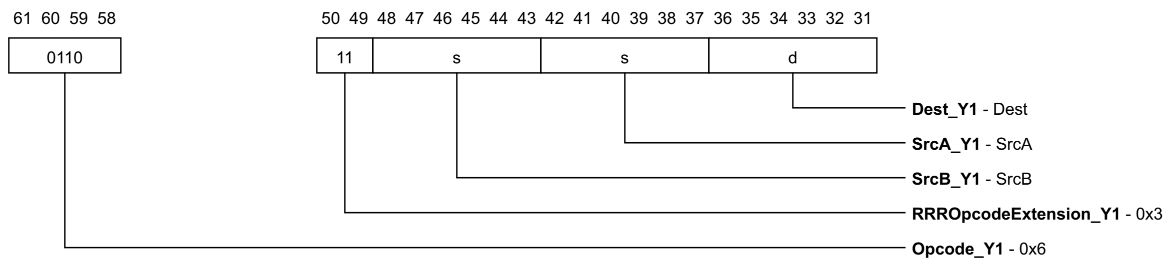

Figure 3-80: sub in Y1 Bits Encoding

subx

Subtract and Extend

Syntax

```
subx Dest, SrcA, SrcB
```

Example

```
subx r5, r6, r7
```

Description

Subtract the second 4-byte operand from the first 4-byte operand and sign-extend the result.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] - rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

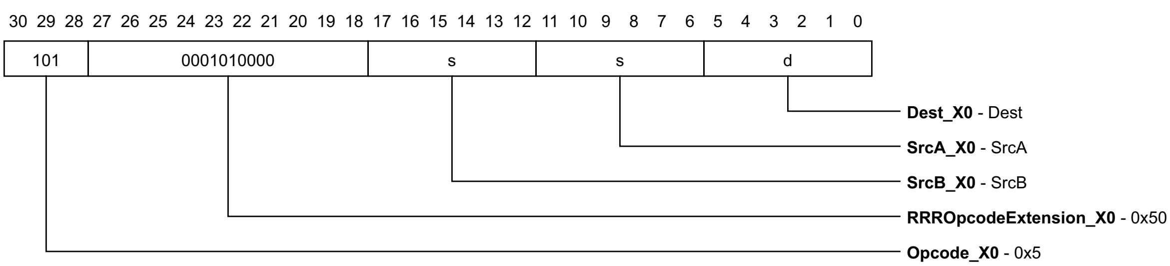

Figure 3-81: subx in X0 Bits Encoding

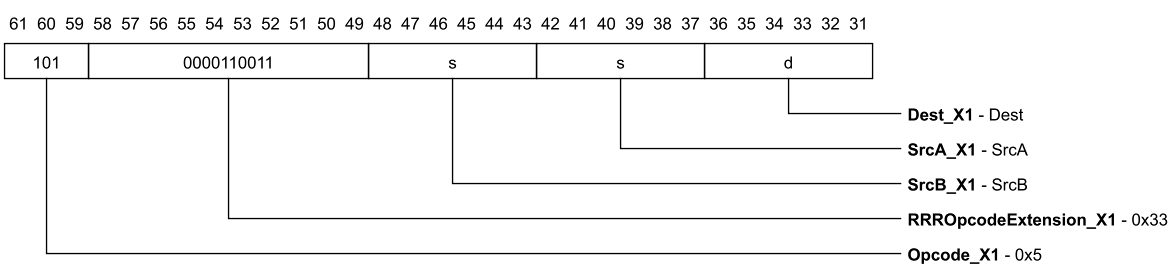

Figure 3-82: subx in X1 Bits Encoding

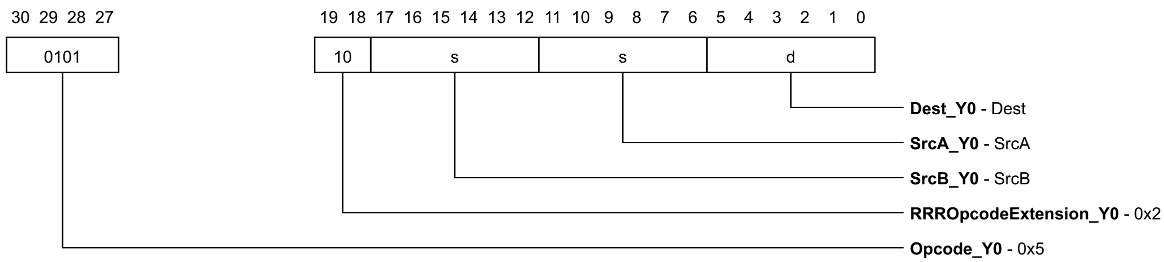

Figure 3-83: subx in Y0 Bits Encoding

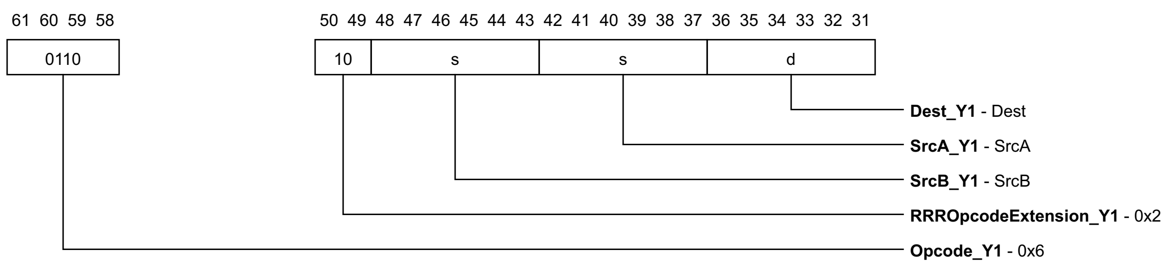

Figure 3-84: subx in Y1 Bits Encoding

subxsc

Subtract Signed Clamped and Extend

Syntax

```
subxsc Dest, SrcA, SrcB
```

Example

```
subxsc r5, r6, r7
```

Description

Subtracts the second 4-byte operand from the first 4-byte operand, saturating the result at the minimum negative value or the maximum positive 4-byte value. The result is then sign-extended.

Functional Description

```
rf[Dest] =
signExtend32 (signed_saturate32
      (signExtend32 (rf[SrcA]) - signExtend32 (rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

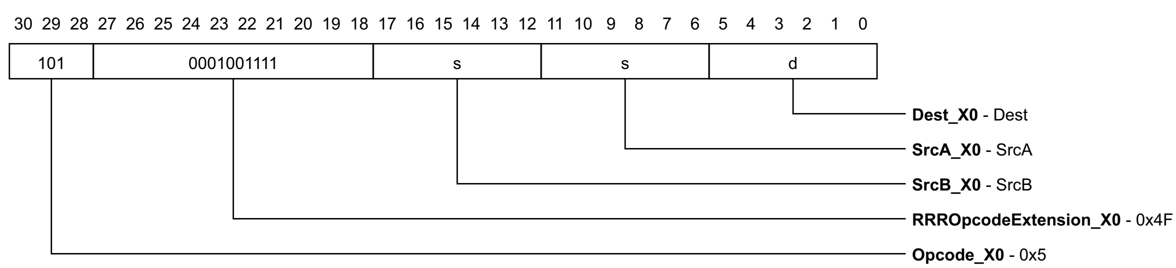

Figure 3-85: subxsc in X0 Bits Encoding

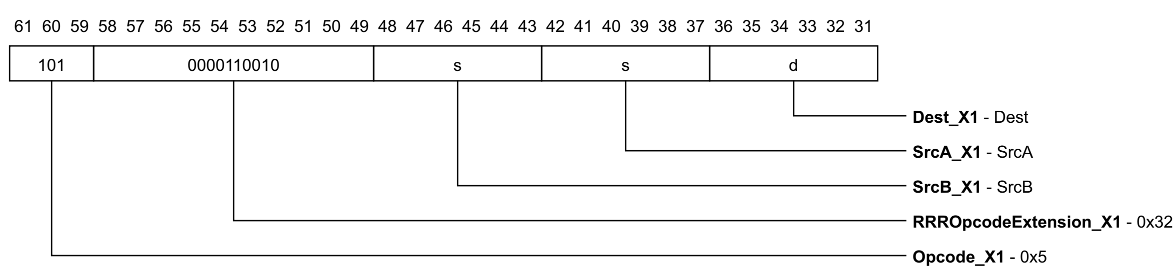

Figure 3-86: subxsc in X1 Bits Encoding

CHAPTER 4 BIT MANIPULATION INSTRUCTIONS

4.1 Overview

The following sections provide detailed descriptions of bit manipulation instructions listed alphabetically.

clz	Count Leading Zeros
crc32_32	CRC32 32-bit Step
crc32_8	CRC32 8-bit Step
ctz	Count Trailing Zeros
dblalign	Double Align
dblalign2	Double Align by Two Bytes
dblalign4	Double Align by Four Bytes
dblalign6	Double Align by Six Bytes
pcnt	Population Count
revbits	Reverse Bits
revbytes	Reverse Bytes
shufflebytes	Shuffle Bytes

4.2 Instructions

Bit manipulation instructions are described in the sections that follow.

clz

Count Leading Zeros

Syntax

```
clz Dest, SrcA
```

Example

```
clz r5, r6
```

Description

Returns the number leading zeros in a word before a bit is set (1). This instruction scans the input word from the most significant bit to the least significant bit. The result of this operation can range from 0 to `WORD_SIZE`.

Functional Description

```
unsigned int counter;
for (counter = 0; counter < WORD_SIZE; counter++)
  {
    if ((rf[SrcA] >> (WORD_SIZE - 1 - counter)) & 0x1)
      {
break;
      }
  }

rf[Dest] = counter;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

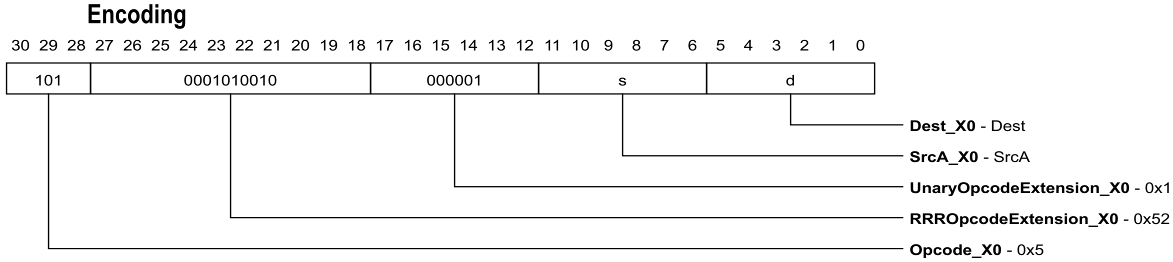

Figure 4-87: clz in X0 Bits Encoding

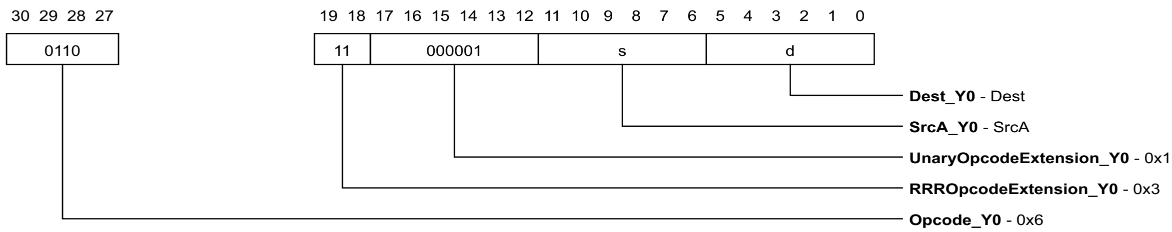

Figure 4-88: clz in Y0 Bits Encoding

crc32_32

CRC32 32-bit Step

Syntax

```
crc32_32 Dest, SrcA, SrcB
```

Example

```
crc32_32 r5, r6, r7
```

Description

Updates a CRC32 value in the first operand with the second operand.

Functional Description

```
uint32_t accum = rf[SrcA];
uint32_t input = rf[SrcB];
for (unsigned int Counter = 0; Counter < 32; Counter++)
  {
    accum =
      (accum >> 1) ^ (((input & 1) ^ (accum & 1)) ? 0xEDB88320 : 0x00000000);
    input = input >> 1;
  } rf[Dest] = accum;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

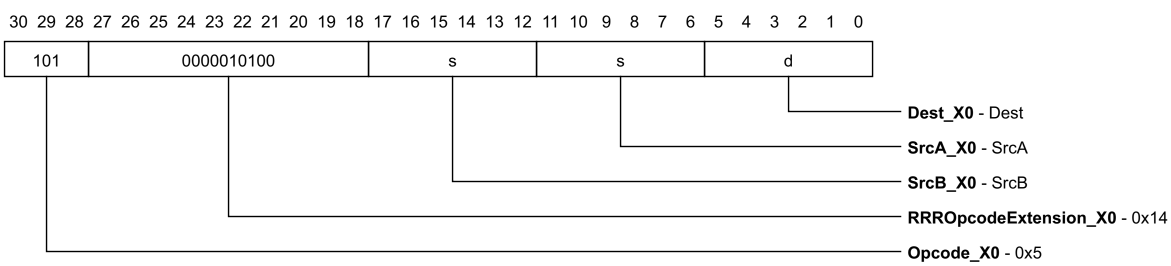

Figure 4-89: crc32_32 in X0 Bits Encoding

crc32_8

CRC32 8-bit Step

Syntax

```
crc32_8 Dest, SrcA, SrcB
```

Example

```
crc32_8 r5, r6, r7
```

Description

Updates a CRC32 value in the first operand with the low-order 8 bits of the second operand.

Functional Description

```
uint32_t accum = rf[SrcA];
uint32_t input = rf[SrcB];
for (unsigned int Counter = 0; Counter < 8; Counter++)
  {
    accum =
      (accum >> 1) ^ (((input & 1) ^ (accum & 1)) ? 0xEDB88320 : 0x00000000);
    input = input >> 1;
  } rf[Dest] = accum;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

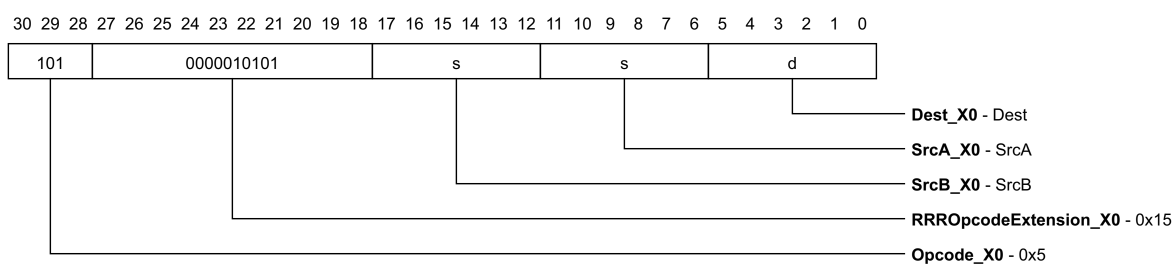

Figure 4-90: crc32_8 in X0 Bits Encoding

ctz

Count Trailing Zeros

Syntax

```
ctz Dest, SrcA
```

Example

```
ctz r5, r6
```

Description

Returns the number trailing zeros in a word before a bit is set (1). This instruction scans the input word from the least significant bit to the most significant bit. The result of this operation can range from 0 to `WORD_SIZE`.

Functional Description

```
unsigned int counter;
for (counter = 0; counter < WORD_SIZE; counter++)
  {
    if ((rf[SrcA] >> counter) & 0x1)
      {
break;
      }
  }

rf[Dest] = counter;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

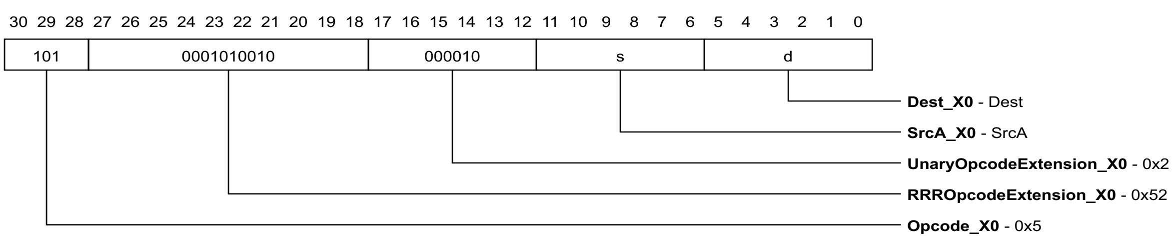

Figure 4-91: ctz in X0 Bits Encoding

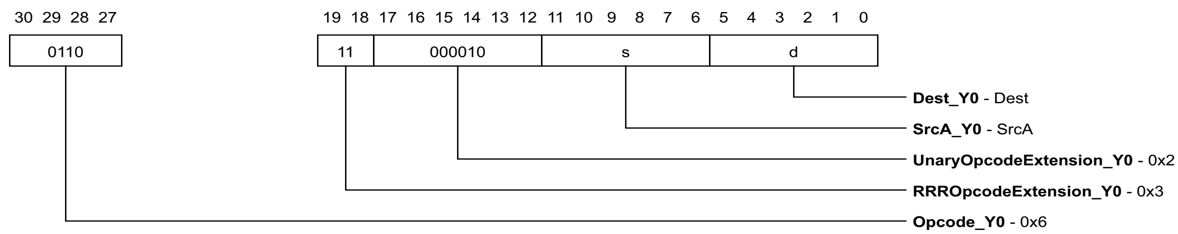

Figure 4-92: ctz in Y0 Bits Encoding

dblalign

Double Align

Syntax

```
dblalign Dest, SrcA, SrcB
```

Example

```
dblalign r5, r6, r7
```

Description

Shift a 128-bit value by the number of bytes specified by the bottom three bits of the second source operand. The shift direction is to the right when the processor is in little-endian mode, and to the left if the processor is in big-endian mode. The 128-bit source value is constructed from the concatenation of the first source operand and the destination register.

Functional Description

```
int shift = (rf[SrcB] & 7) * BYTE_SIZE;
uint64_t a = rf[SrcA];
uint64_t b = rf[Dest];
rf[Dest] = (little_endian ()? (shift == 0
                      ? b
                      : ((a << (64 - shift)) | (b >> shift)))
    : (shift == 0 ? b : ((b << shift) | (a >> (64 - shift)))));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

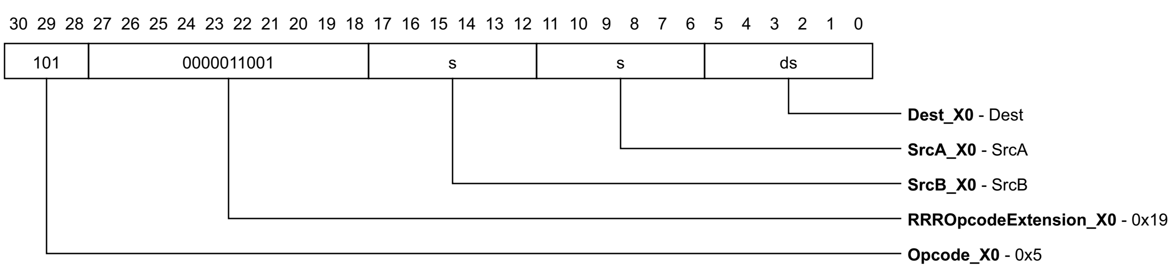

Figure 4-93: dblalign in X0 Bits Encoding

dblalign2

Double Align by Two Bytes

Syntax

```
dblalign2 Dest, SrcA, SrcB
```

Example

```
dblalign2 r5, r6, r7
```

Description

Shift a 128-bit value by two bytes. The shift direction is to the right when the processor is in little-endian mode, and to the left if the processor is in big-endian mode. The 128-bit source value is constructed from the concatenation of the first source operand and the second source register.

Functional Description

```
uint64_t a = rf[SrcA];
uint64_t b = rf[SrcB];
rf[Dest] =
  (little_endian ()? ((a << (64 - 2 * BYTE_SIZE)) | (b >> 2 * BYTE_SIZE))
   : ((b << 2 * BYTE_SIZE) | (a >> (64 - 2 * BYTE_SIZE))));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

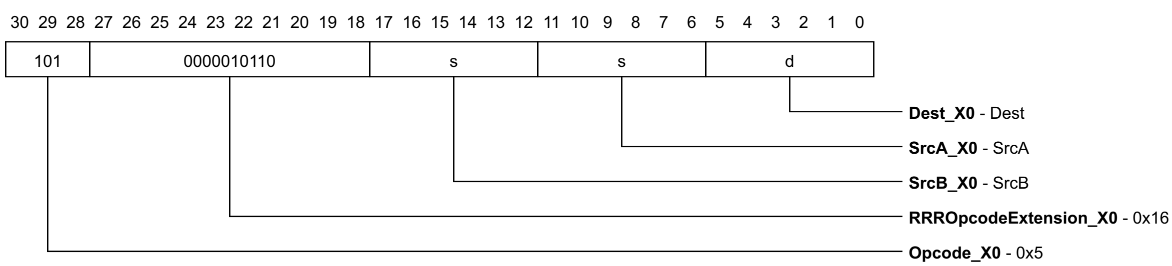

Figure 4-94: dblalign2 in X0 Bits Encoding

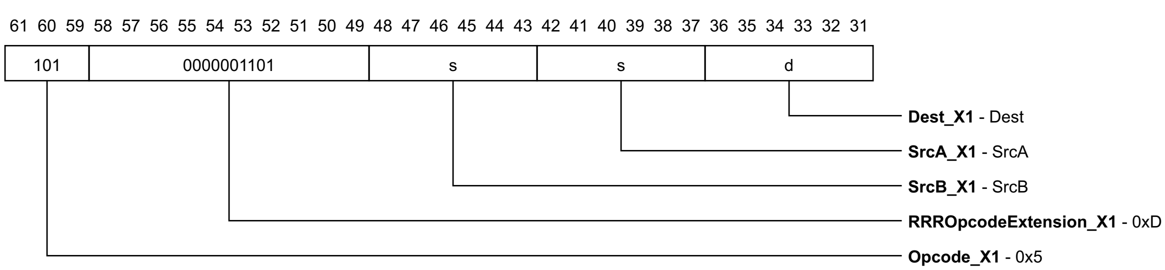

Figure 4-95: dblalign2 in X1 Bits Encoding

dblalign4

Double Align by Four Bytes

Syntax

```
dblalign4 Dest, SrcA, SrcB
```

Example

```
dblalign4 r5, r6, r7
```

Description

Shift a 128-bit value by four bytes. The shift direction is to the right when the processor is in little-endian mode, and to the left if the processor is in big-endian mode. The 128-bit source value is constructed from the concatenation of the first source operand and the second source register.

Functional Description

```
uint64_t a = rf[SrcA];
uint64_t b = rf[SrcB];
rf[Dest] =
  (little_endian ()? ((a << (64 - 4 * BYTE_SIZE)) | (b >> 4 * BYTE_SIZE))
   : ((b << 4 * BYTE_SIZE) | (a >> (64 - 4 * BYTE_SIZE))));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

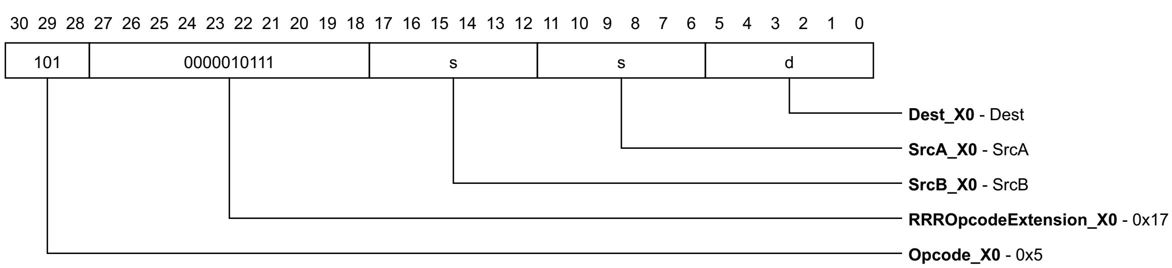

Figure 4-96: dblalign4 in X0 Bits Encoding

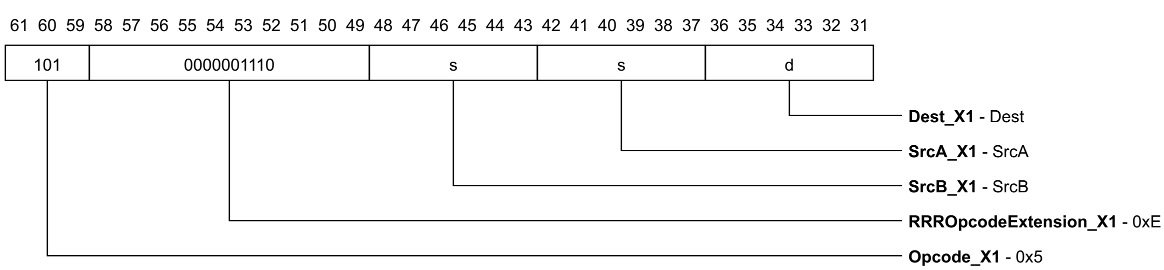

Figure 4-97: dblalign4 in X1 Bits Encoding

dblalign6

Double Align by Six Bytes

Syntax

```
dblalign6 Dest, SrcA, SrcB
```

Example

```
dblalign6 r5, r6, r7
```

Description

Shift a 128-bit value by six bytes. The shift direction is to the right when the processor is in little-endian mode, and to the left if the processor is in big-endian mode. The 128-bit source value is constructed from the concatenation of the first source operand and the second source register.

Functional Description

```
uint64_t a = rf[SrcA];
uint64_t b = rf[SrcB];
rf[Dest] =
  (little_endian ()? ((a << (64 - 6 * BYTE_SIZE)) | (b >> 6 * BYTE_SIZE))
   : ((b << 6 * BYTE_SIZE) | (a >> (64 - 6 * BYTE_SIZE))));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

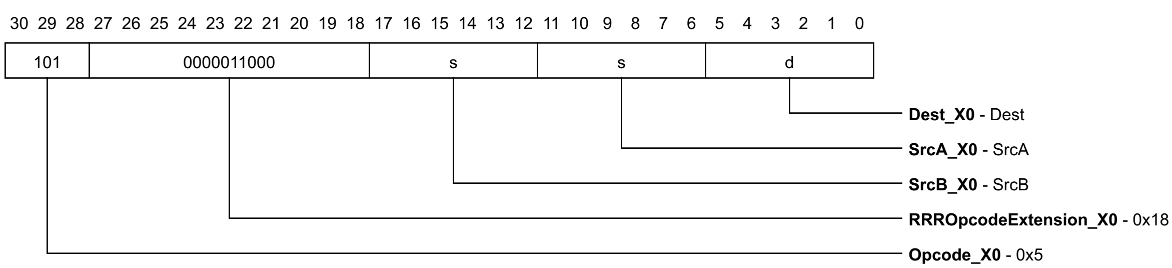

Figure 4-98: dblalign6 in X0 Bits Encoding

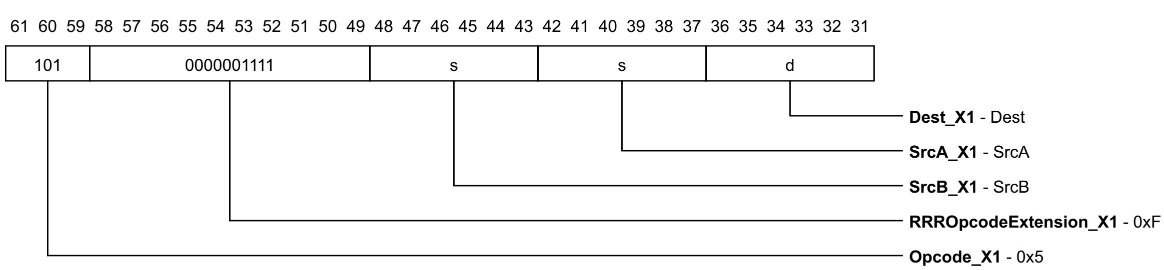

Figure 4-99: dblalign6 in X1 Bits Encoding

pcnt

Population Count

Syntax

```
pcnt Dest, SrcA
```

Example

```
pcnt r5, r6
```

Description

Returns the number of bits set (1) in the source operand. The result of this operation can range from 0 to `WORD_SIZE`.

Functional Description

```
unsigned int counter;
int numberOfOnes = 0;
for (counter = 0; counter < WORD_SIZE; counter++)
  {
    numberOfOnes += (rf[SrcA] >> counter) & 0x1;
  }

rf[Dest] = numberOfOnes;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

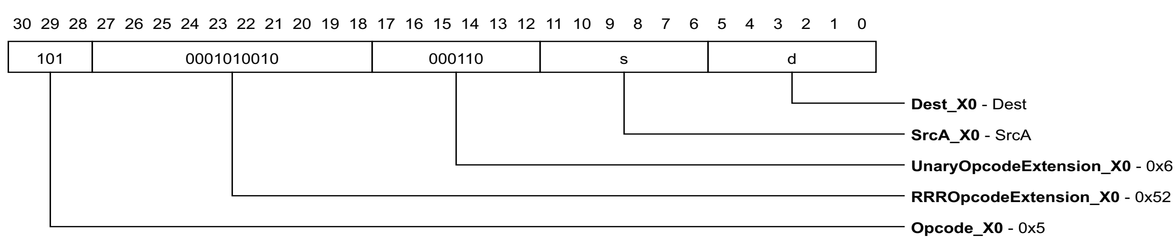

Figure 4-100: pcnt in X0 Bits Encoding

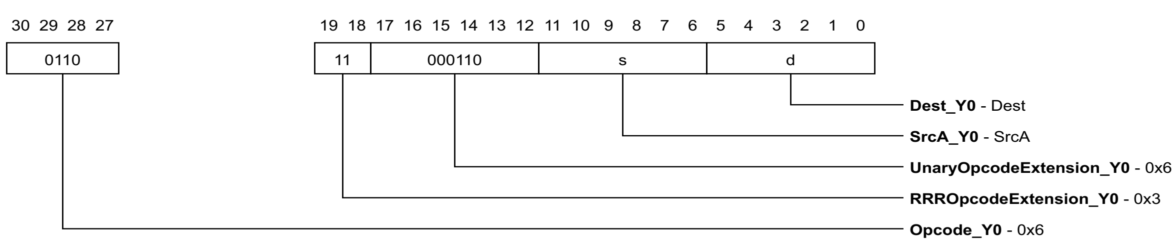

Figure 4-101: pcnt in Y0 Bits Encoding

revbits

Reverse Bits

Syntax

```
revbits Dest, SrcA
```

Example

```
revbits r5, r6
```

Description

Reorders a word such that the most significant bit becomes the least significant bit in the output, the second most significant bit becomes the second least significant bit in the output, and the n'th most significant bit becomes n'th least significant bit in the output.

Functional Description

```
uint64_t output = 0;
unsigned int counter;
for (counter = 0; counter < (WORD_SIZE); counter++)
  {
    output |=
      (((rf[SrcA] >> (counter)) & 0x1) << ((WORD_SIZE - 1) - counter));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

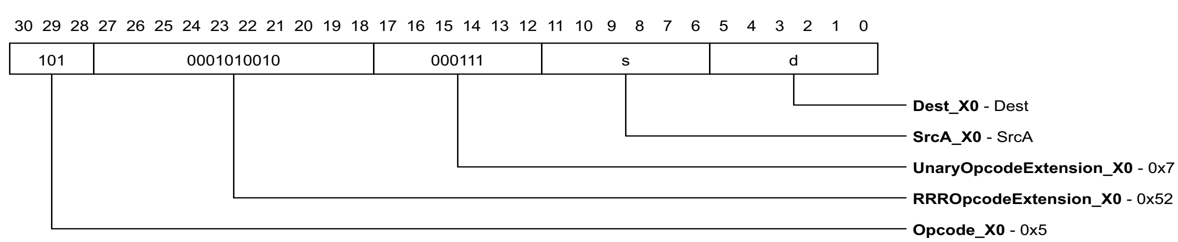

Figure 4-102: revbits in X0 Bits Encoding

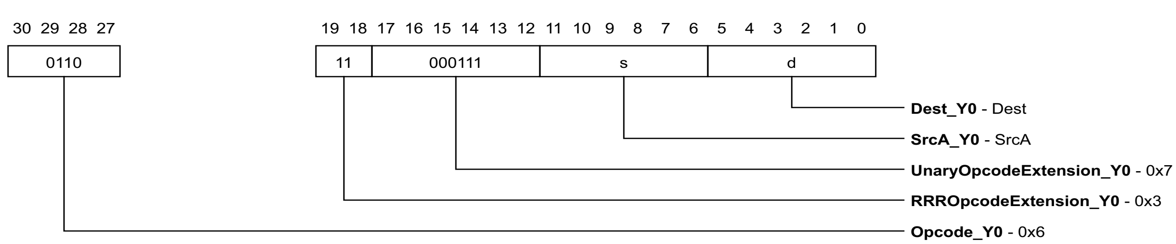

Figure 4-103: revbits in Y0 Bits Encoding

revbytes

Reverse Bytes

Syntax

```
revbytes Dest, SrcA
```

Example

```
revbytes r5, r6
```

Description

Reorders a word such that the most significant byte becomes the least significant byte in the output, the second most significant byte becomes the second least significant byte in the output, and the n'th most significant byte becomes n'th least significant byte in the output.

Functional Description

```
uint64_t output = 0;
unsigned int counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output |=
      (((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK) <<
       ((((WORD_SIZE / BYTE_SIZE) - 1) - counter) * BYTE_SIZE));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

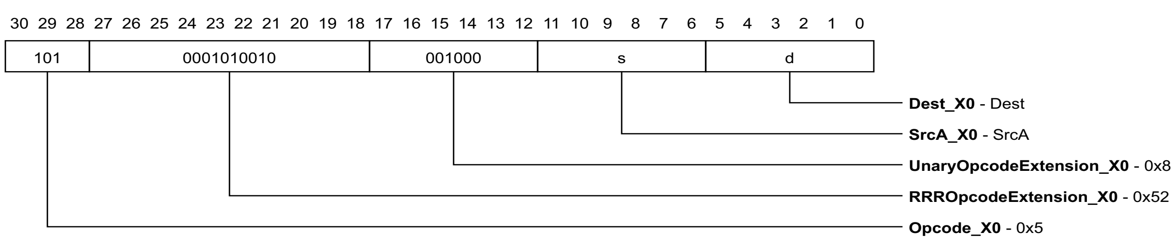

Figure 4-104: revbytes in X0 Bits Encoding

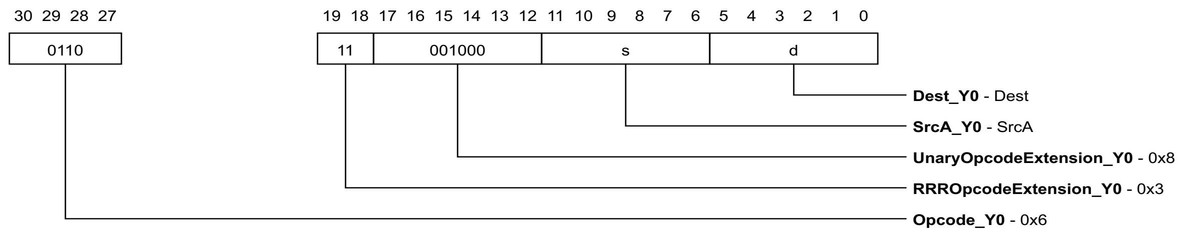

Figure 4-105: revbytes in Y0 Bits Encoding

shufflebytes

Shuffle Bytes

Syntax

```
shufflebytes Dest, SrcA, SrcB
```

Example

```
shufflebytes r5, r6, r7
```

Description

Set each byte in the destination to a byte extracted from the destination operand or the first source operand. The selected byte is specified by a byte number in the corresponding byte of the second operand. Byte specification between 0 and 7 correspond to selecting bytes from the destination operand, and byte numbers between 8 and 15 correspond to selecting bytes from the first source operand.

Functional Description

```
uint64_t a = rf[SrcA];
uint64_t b = rf[SrcB];
uint64_t d = rf[Dest];
uint64_t output = 0;
unsigned int counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int sel = getByte (b, counter) & 0xf;
    uint8_t byte = (sel < 8) ? getByte (d, sel) : getByte (a, (sel - 8));
    output = setByte (output, counter, byte);
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

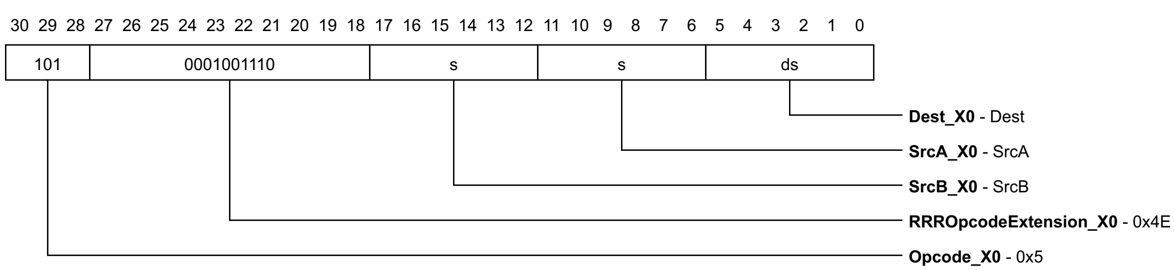

Figure 4-106: shufflebytes in X0 Bits Encoding

CHAPTER 5 COMPARE INSTRUCTIONS

5.1 Overview

The following sections provide detailed descriptions of compare instructions listed alphabetically.

cmpeq	Compare Equal
cmpeqi	Compare Equal Immediate
cmples	Compare Less Than or Equal Signed
cmpleu	Compare Less Than or Equal Unsigned
cmplts	Compare Less Than Signed
cmpltsi	Compare Less Than Signed Immediate
cmpltu	Compare Less Than Unsigned
cmpltui	Compare Less Than Unsigned Immediate
cmpne	Compare Not Equal

5.2 Instructions

Compare instructions are described in the sections that follow.

cmpeq

Compare Equal

Syntax

```
cmpeq Dest, SrcA, SrcB
```

Example

```
cmpeq r5, r6, r7
```

Description

Sets the result to 1 if the first source operand is equal to the second source operand. Otherwise the result is set to 0.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] == (uint64_t) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

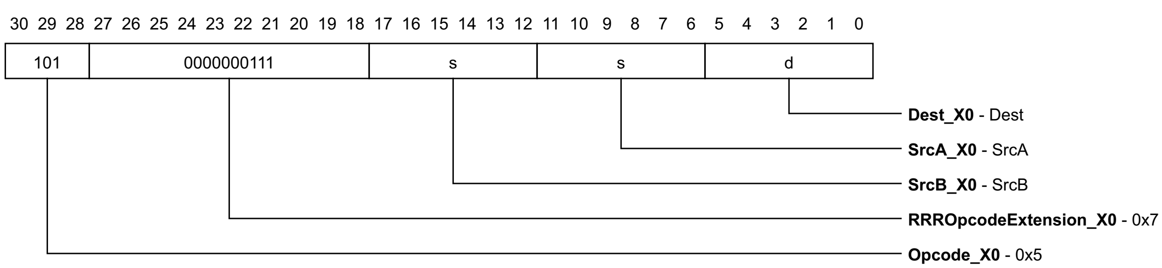

Figure 5-107: cmpeq in X0 Bits Encoding

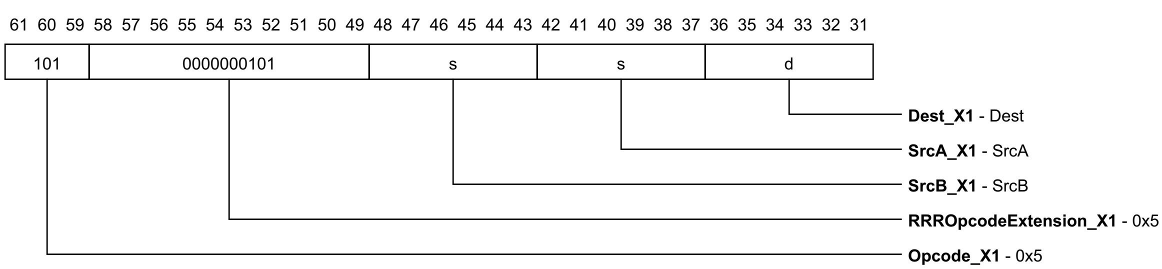

Figure 5-108: cmpeq in X1 Bits Encoding

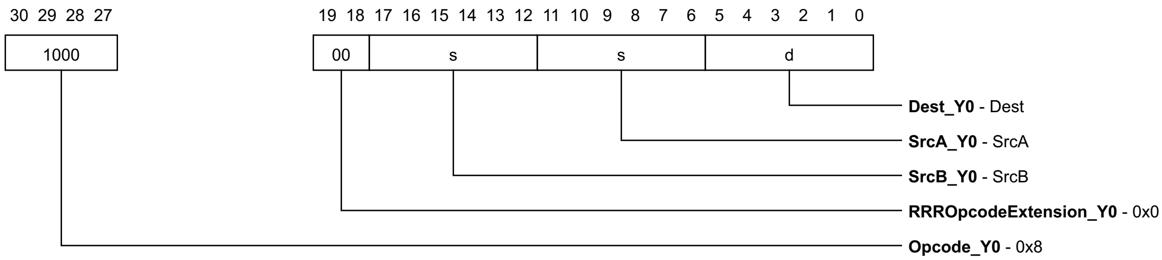

Figure 5-109: cmpeq in Y0 Bits Encoding

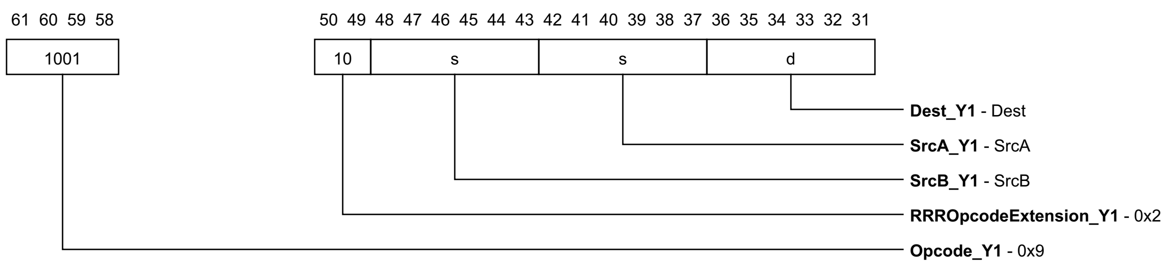

Figure 5-110: cmpeq in Y1 Bits Encoding

cmpeqi

Compare Equal Immediate

Syntax

```
cmpeqi Dest, SrcA, Imm8
```

Example

```
cmpeqi r5, r6, 5
```

Description

Sets the result to 1 if the first source operand is equal to a sign extended immediate. Otherwise the result is set to 0.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] == (uint64_t) signExtend8 (Imm8)) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

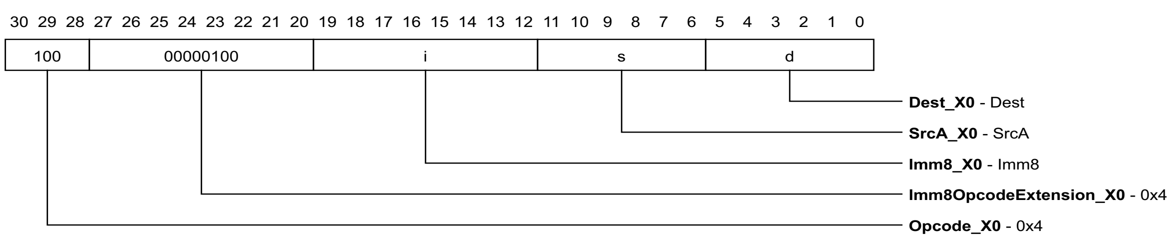

Figure 5-111: cmpeqi in X0 Bits Encoding

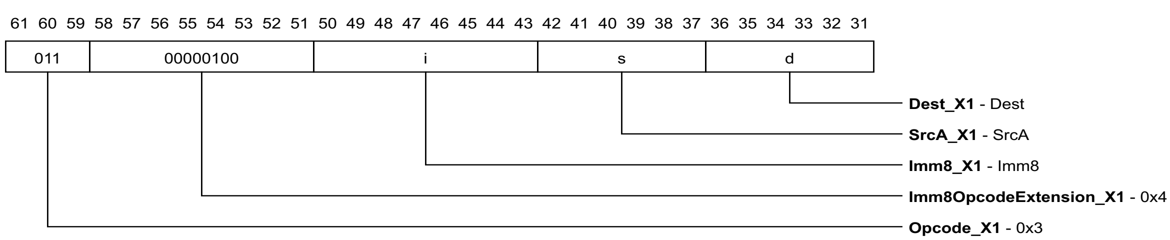

Figure 5-112: cmpeqi in X1 Bits Encoding

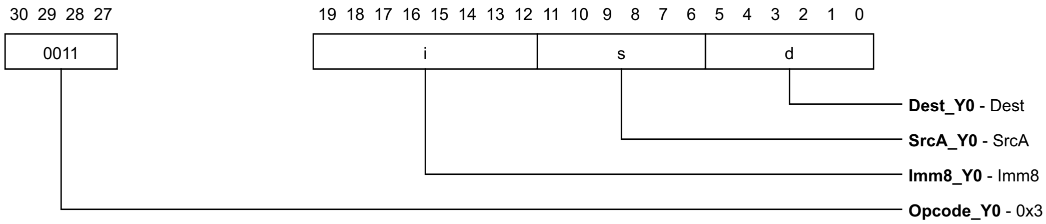

Figure 5-113: cmpeqi in Y0 Bits Encoding

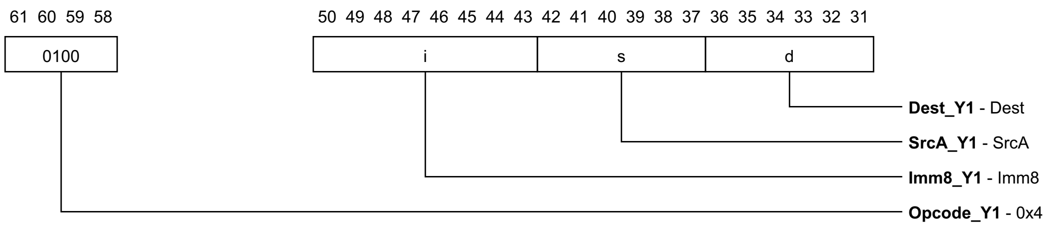

Figure 5-114: cmpeqi in Y1 Bits Encoding

cmples

Compare Less Than or Equal Signed

Syntax

```
cmples Dest, SrcA, SrcB
```

Example

```
cmples r5, r6, r7
```

Description

Sets the result to 1 if the first source operand is less than or equal to the second source operand. Otherwise the result is set to 0. This instruction treats both source operands as signed values.

Functional Description

```
rf[Dest] = ((int64_t) rf[SrcA] <= (int64_t) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

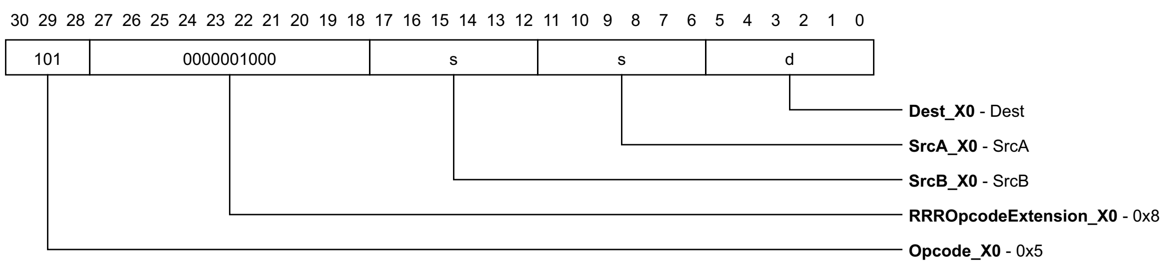

Figure 5-115: cmples in X0 Bits Encoding

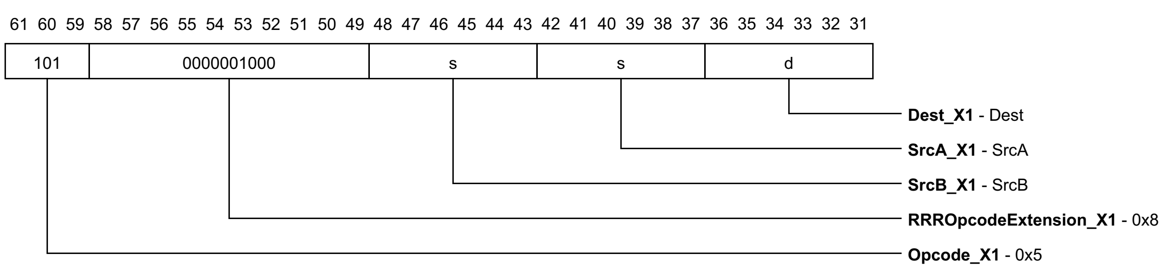

Figure 5-116: cmples in X1 Bits Encoding

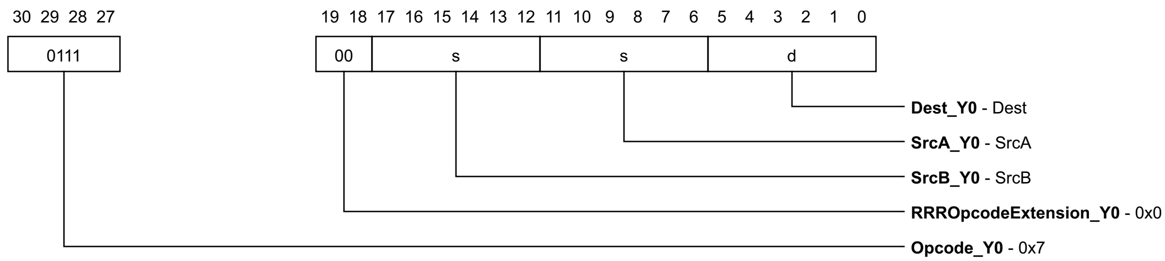

Figure 5-117: cmples in Y0 Bits Encoding

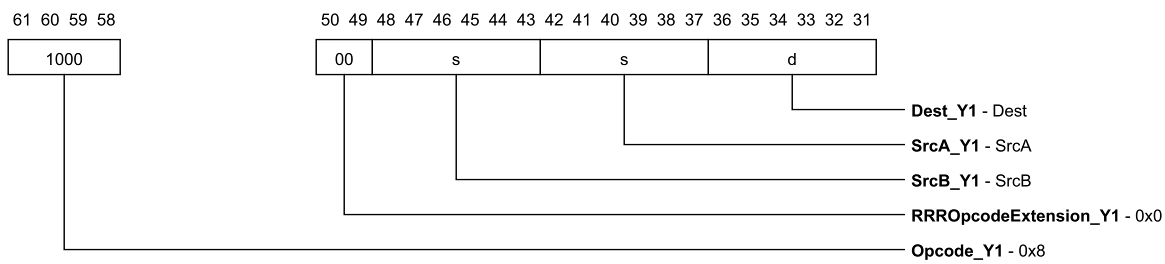

Figure 5-118: cmples in Y1 Bits Encoding

cmpleu

Compare Less Than or Equal Unsigned

Syntax

```
cmpleu Dest, SrcA, SrcB
```

Example

```
cmpleu r5, r6, r7
```

Description

Sets the result to 1 if the first source operand is less than or equal to the second source operand. Otherwise the result is set to 0. This instruction treats both source operands as unsigned values.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] <= (uint64_t) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

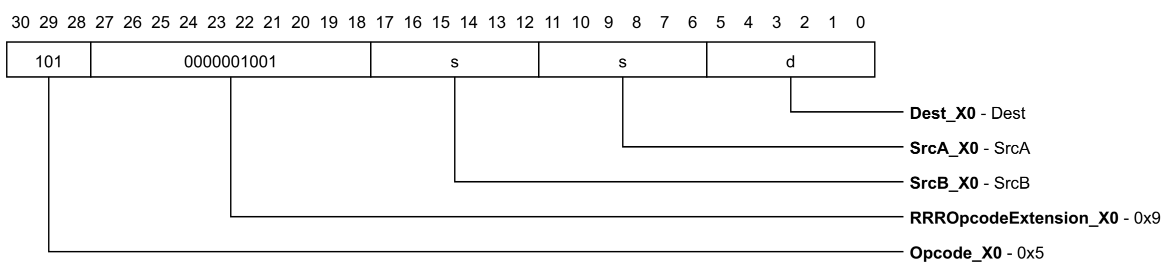

Figure 5-119: cmpleu in X0 Bits Encoding

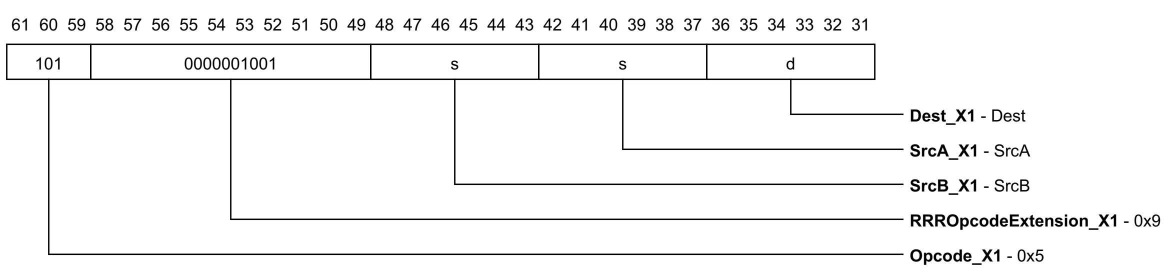

Figure 5-120: cmpleu in X1 Bits Encoding

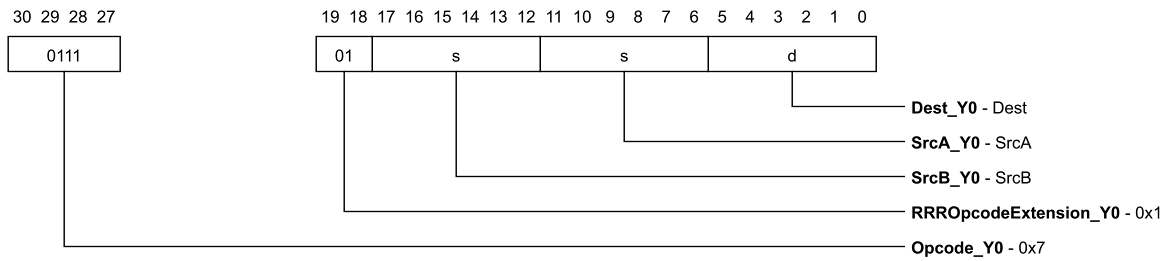

Figure 5-121: cmpleu in Y0 Bits Encoding

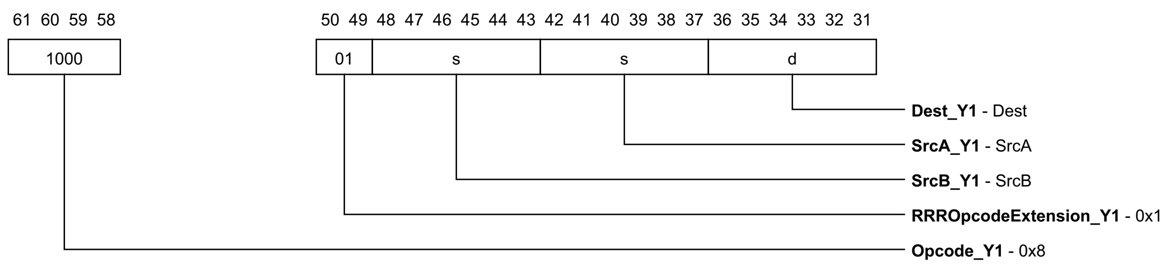

Figure 5-122: cmpleu in Y1 Bits Encoding

cmplts

Compare Less Than Signed

Syntax

```
cmplts Dest, SrcA, SrcB
```

Example

```
cmplts r5, r6, r7
```

Description

Sets the result to 1 if the first source operand is less than the second source operand. Otherwise the result is set to 0. This instruction treats both source operands as signed values.

Functional Description

```
rf[Dest] = ((int64_t) rf[SrcA] < (int64_t) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

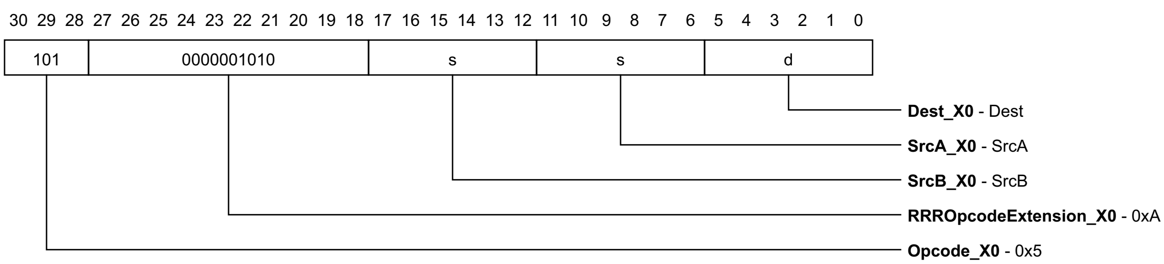

Figure 5-123: cmplts in X0 Bits Encoding

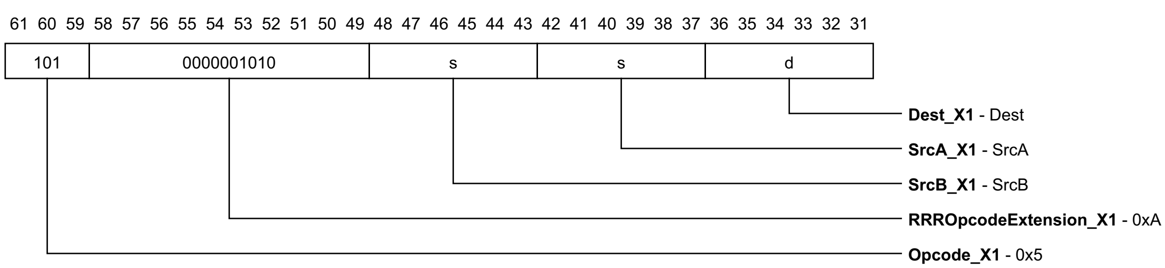

Figure 5-124: cmplts in X1 Bits Encoding

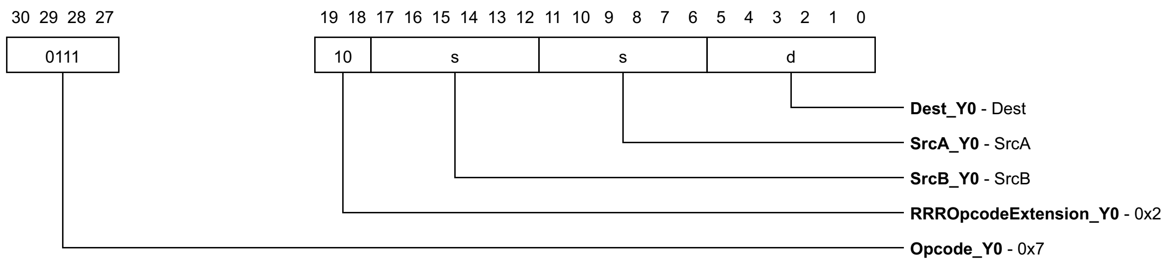

Figure 5-125: cmplts in Y0 Bits Encoding

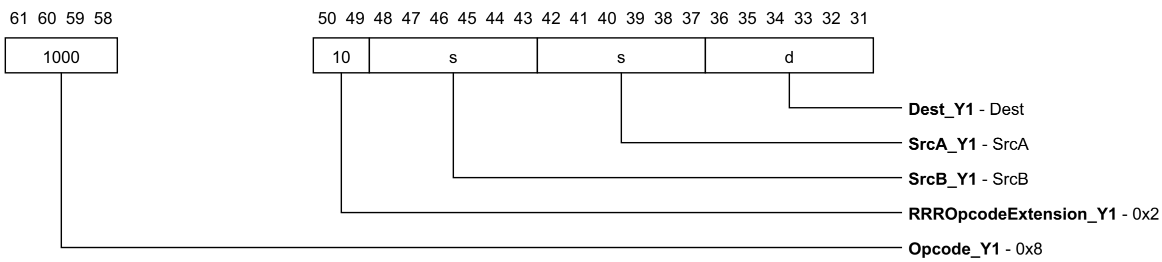

Figure 5-126: cmplts in Y1 Bits Encoding

cmpltsi

Compare Less Than Signed Immediate

Syntax

```
cmpltsi Dest, SrcA, Imm8
```

Example

```
cmpltsi r5, r6, 5
```

Description

Sets the result to 1 if the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source operands as signed values.

Functional Description

```
rf[Dest] = ((int64_t) rf[SrcA] < ((int64_t) signExtend8 (Imm8))) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

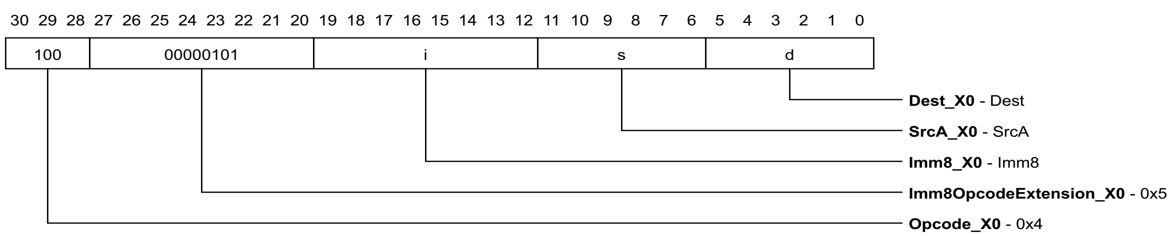

Figure 5-127: cmpltsi in X0 Bits Encoding

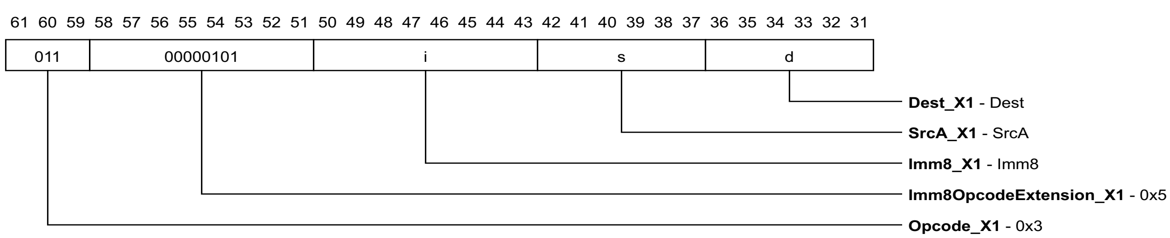

Figure 5-128: cmpltsi in X1 Bits Encoding

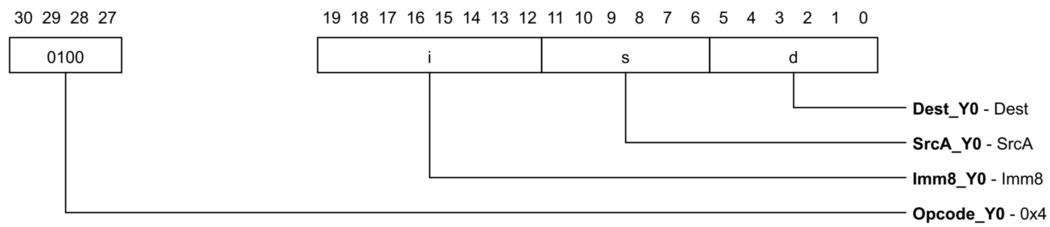

Figure 5-129: cmpltsi in Y0 Bits Encoding

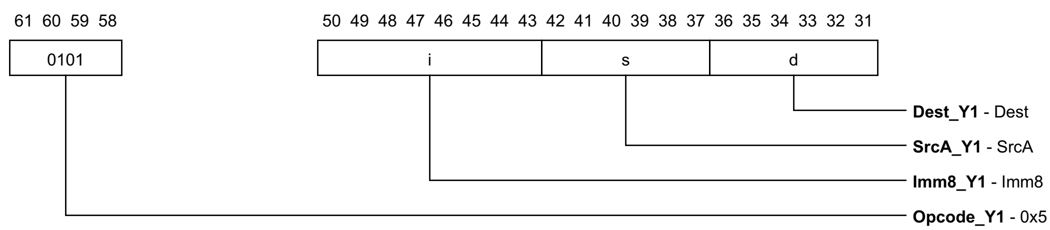

Figure 5-130: cmpltsi in Y1 Bits Encoding

cmpltu

Compare Less Than Unsigned

Syntax

```
cmpltu Dest, SrcA, SrcB
```

Example

```
cmpltu r5, r6, r7
```

Description

Sets the result to 1 if the first source operand is less than the second source operand or sign extended immediate. Otherwise the result is set to 0. This instruction treats both source operands as unsigned values.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] < (uint64_t) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

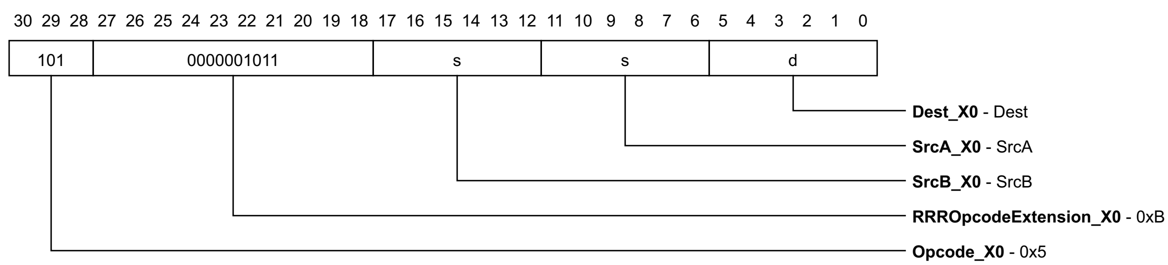

Figure 5-131: cmpltu in X0 Bits Encoding

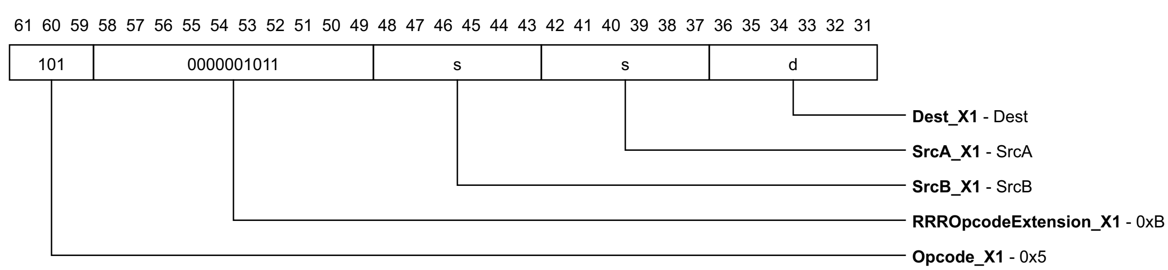

Figure 5-132: cmpltu in X1 Bits Encoding

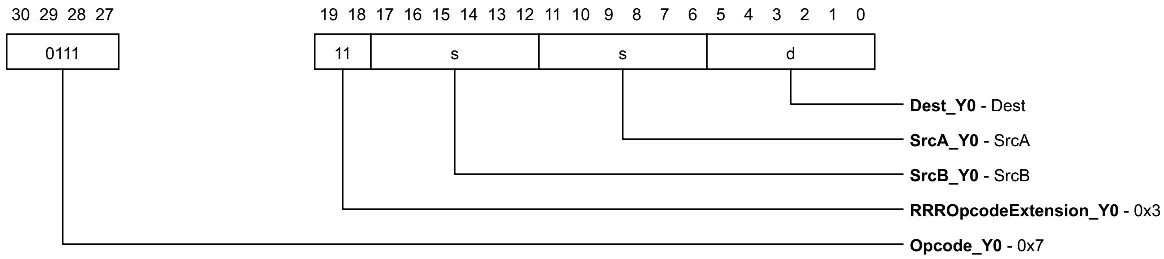

Figure 5-133: cmpltu in Y0 Bits Encoding

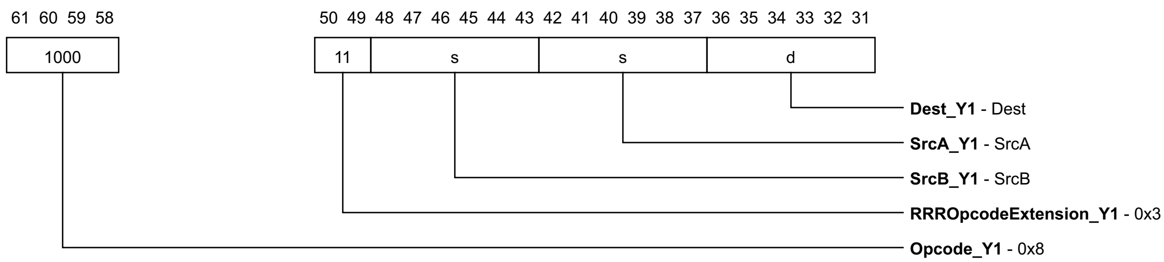

Figure 5-134: cmpltu in Y1 Bits Encoding

cmpltui

Compare Less Than Unsigned Immediate

Syntax

```
cmpltui Dest, SrcA, Imm8
```

Example

```
cmpltui r5, r6, 5
```

Description

Sets the result to 1 if the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source operands as unsigned values.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] < ((uint64_t) signExtend8 (Imm8))) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

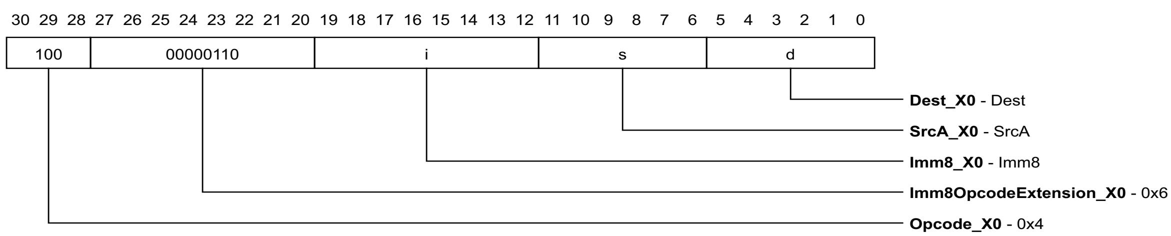

Figure 5-135: cmpltui in X0 Bits Encoding

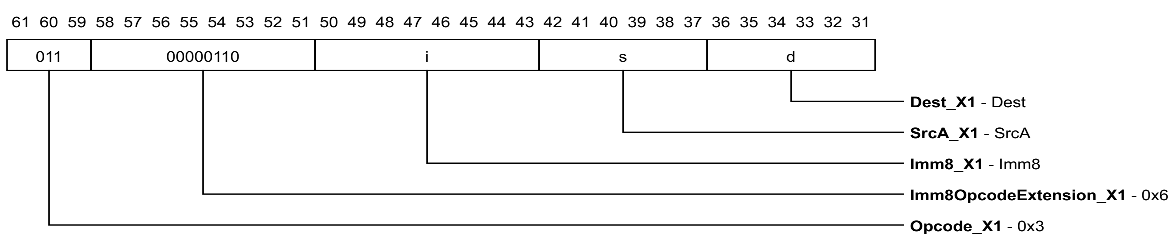

Figure 5-136: cmpltui in X1 Bits Encoding

cmpne

Compare Not Equal

Syntax

```
cmpne Dest, SrcA, SrcB
```

Example

```
cmpne r5, r6, r7
```

Description

Sets the result to 1 if the first source operand is not equal to the second source operand. Otherwise the result is set to 0.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] != (uint64_t) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

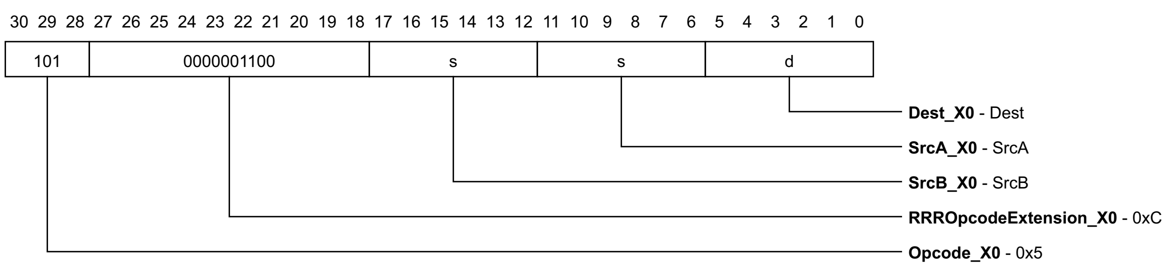

Figure 5-137: cmpne in X0 Bits Encoding

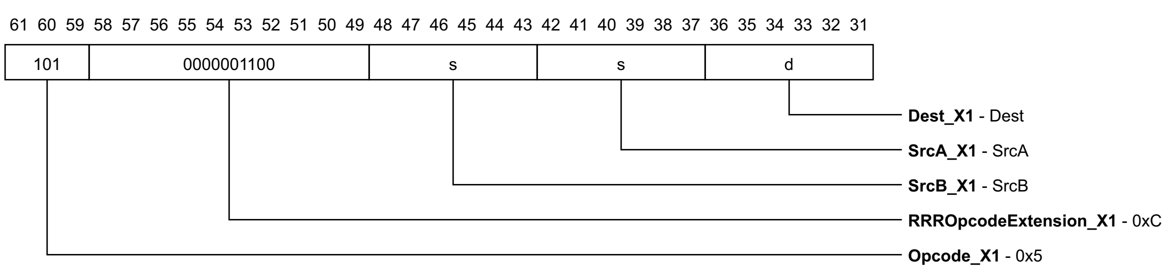

Figure 5-138: cmpne in X1 Bits Encoding

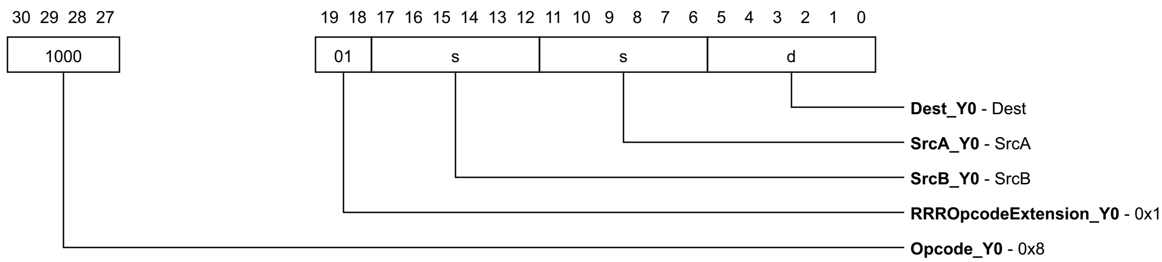

Figure 5-139: cmpne in Y0 Bits Encoding

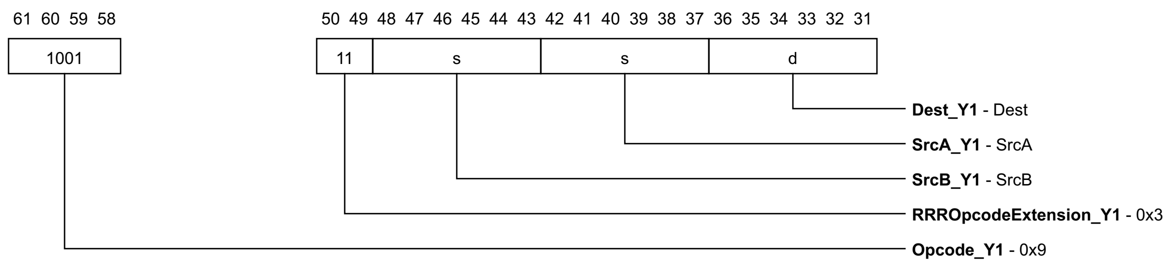

Figure 5-140: cmpne in Y1 Bits Encoding

CHAPTER 6 CONTROL INSTRUCTIONS

6.1 Overview

The following sections provide detailed descriptions of control instructions listed alphabetically.

6.2 Instructions

Control instructions are described in the sections that follow.

beqz

Branch Equal Zero

Syntax

```
beqz SrcA, BrOff
```

Example

```
beqz r5, target
```

Description

Branches to the target if the source operand is equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch zero hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] == 0)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

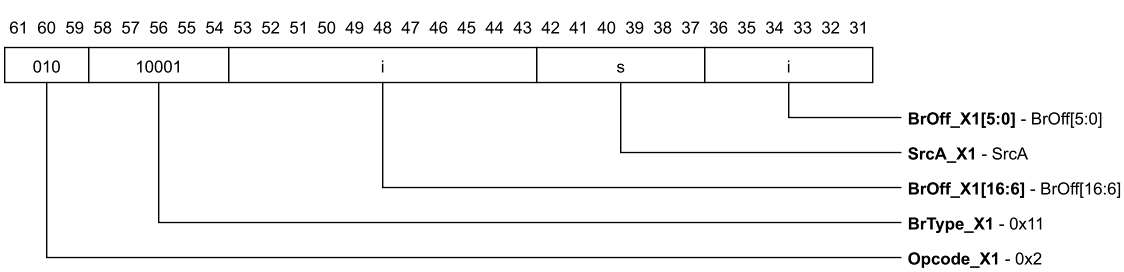

Figure 6-141: beqz in X1 Bits Encoding

bgez

Branch Greater Than or Equal to Zero

Syntax

```
bgez SrcA, BrOff
```

Example

```
bgez r5, target
```

Description

Branches to the target if the source operand is greater than or equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch greater than or equal to zero hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] >= 0)
  {
    setNextPC (getCurrentPC () +
        (signExtend17 (BrOff) <<
        (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

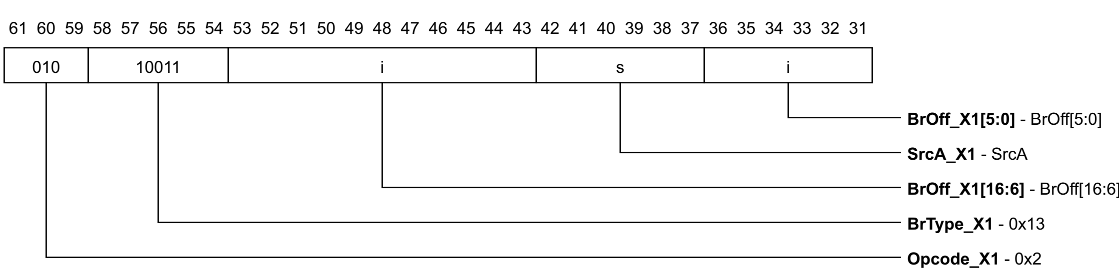

Figure 6-142: bgez in X1 Bits Encoding

beqzt

Branch Zero Predict Taken

Syntax

```
beqzt SrcA, BrOff
```

Example

```
beqzt r5, target
```

Description

Branches to the target if the source operand is equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch zero predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] == 0)
  {
    setNextPC (getCurrentPC () +
        (signExtend17 (BrOff) <<
        (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

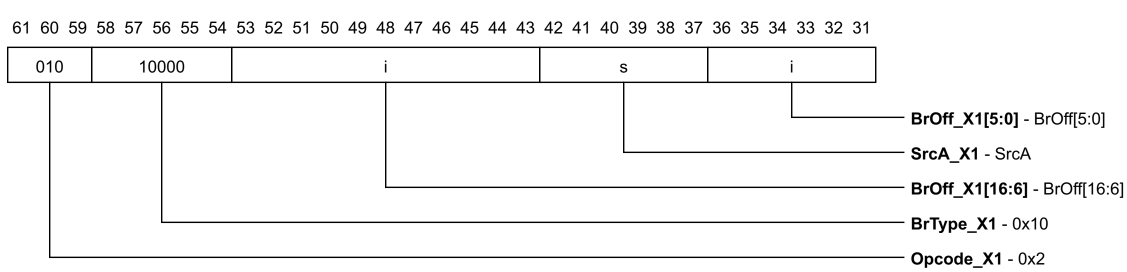

Figure 6-143: beqzt in X1 Bits Encoding

bgezt

Branch Greater Than or Equal to Zero Predict Taken

Syntax

```
bgezt SrcA, BrOff
```

Example

```
bgezt r5, target
```

Description

Branches to the target if the source operand is greater than or equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch greater than or equal to zero predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] >= 0)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

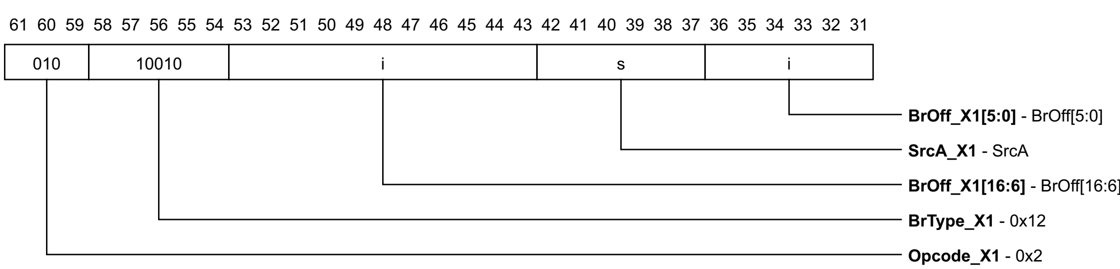

Figure 6-144: bgezt in X1 Bits Encoding

bgtz

Branch Greater Than Zero

Syntax

```
bgtz SrcA, BrOff
```

Example

```
bgtz r5, target
```

Description

Branches to the target if the source operand is greater than zero. Otherwise, the program counter advances to the next instruction in program order. Branch greater than zero hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] > 0)
  {
    setNextPC (getCurrentPC () +
        (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

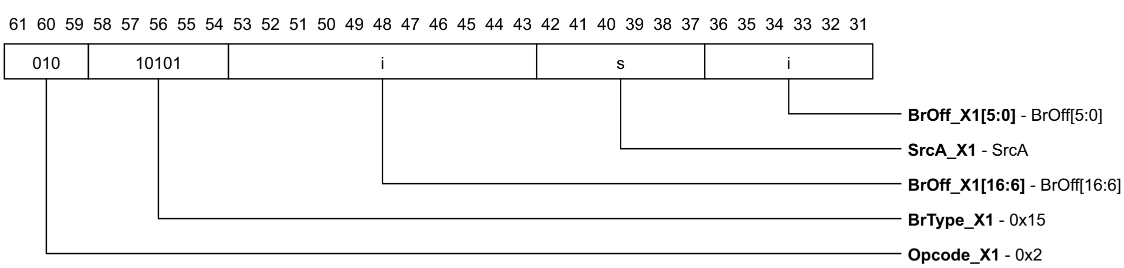

Figure 6-145: bgtz in X1 Bits Encoding

bgtzt

Branch Greater Than Zero Predict Taken

Syntax

```
bgtzt SrcA, BrOff
```

Example

```
bgtzt r5, target
```

Description

Branches to the target if the source operand is greater than zero. Otherwise, the program counter advances to the next instruction in program order. Branch greater than zero predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] > 0)
  {
    setNextPC (getCurrentPC () +
        (signExtend17 (BrOff) <<
        (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

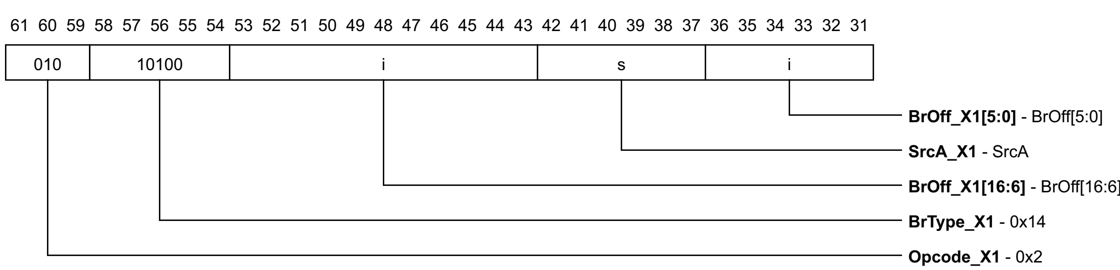

Figure 6-146: bgtzt in X1 Bits Encoding

blbc

Branch Low Bit Clear

Syntax

```
blbc SrcA, BrOff
```

Example

```
blbc r5, target
```

Description

Branches to the target if the source operand's bit zero is clear (0). Otherwise, the program counter advances to the next instruction in program order. Branch bit not set hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register ZERO_REGISTER.

Functional Description

```
if (!(rf[SrcA] & 0x1))
  {
    setNextPC (getCurrentPC () +
        (signExtend17 (BrOff) <<
        (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

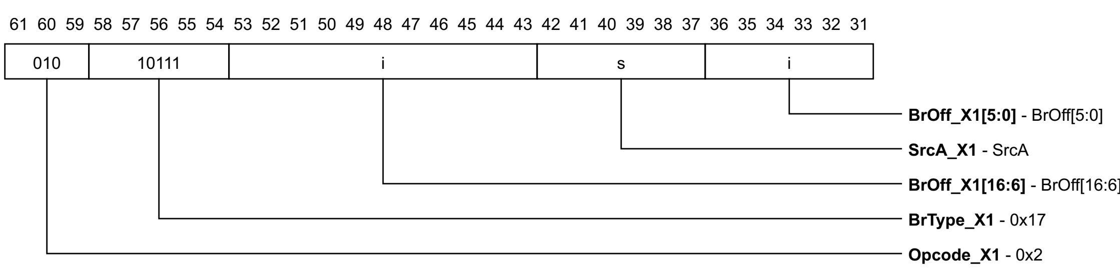

Figure 6-147: blbc in X1 Bits Encoding

blbct

Branch Low Bit Clear Taken

Syntax

```
blbct SrcA, BrOff
```

Example

```
blbct r5, target
```

Description

Branches to the target if the source operand's bit zero is clear (0). Otherwise, the program counter advances to the next instruction in program order. Branch bit not set predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (!(rf[SrcA] & 0x1))
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

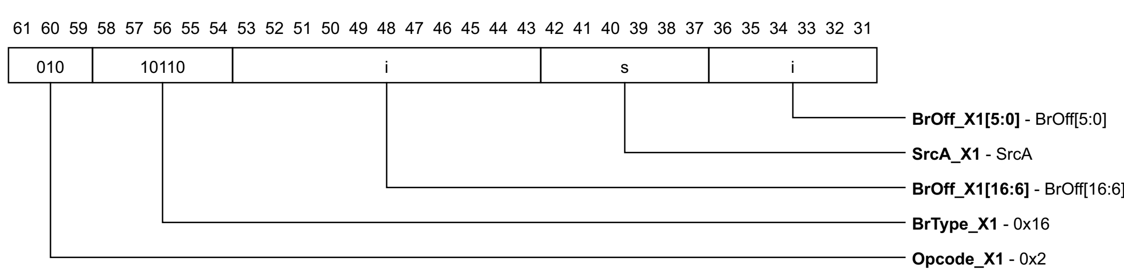

Figure 6-148: blbct in X1 Bits Encoding

blbs

Branch Low Bit Set

Syntax

```
blbs SrcA, BrOff
```

Example

```
blbs r5, target
```

Description

Branches to the target if the source operand's bit zero is set (1). Otherwise, the program counter advances to the next instruction in program order. Branch bit set hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] & 0x1)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

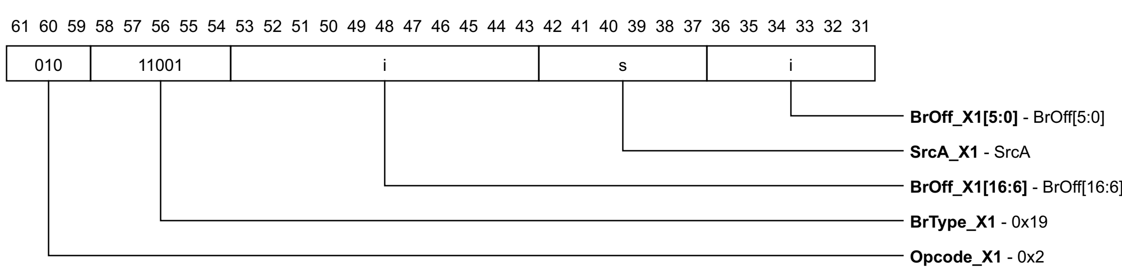

Figure 6-149: blbs in X1 Bits Encoding

blbst

Branch Low Bit Set Taken

Syntax

```
blbst SrcA, BrOff
```

Example

```
blbst r5, target
```

Description

Branches to the target if the source operand's bit zero is set (1). Otherwise, the program counter advances to the next instruction in program order. Branch bit set predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] & 0x1)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

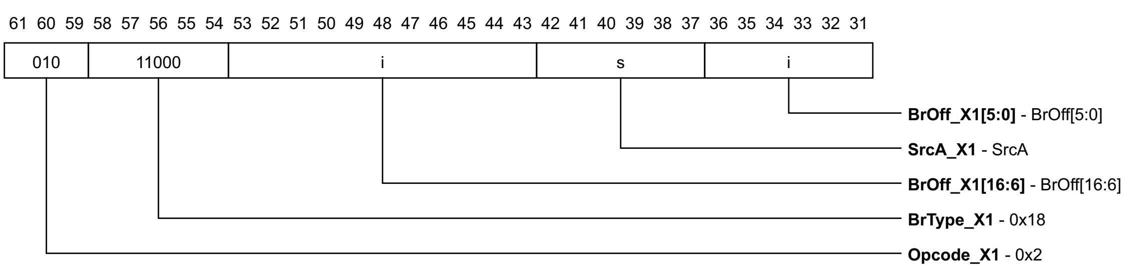

Figure 6-150: blbst in X1 Bits Encoding

blez

Branch Less Than or Equal to Zero

Syntax

```
blez SrcA, BrOff
```

Example

```
blez r5, target
```

Description

Branches to the target if the source operand is less than or equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch less than or equal to zero hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register ZERO_REGISTER.

Functional Description

```
if (rf[SrcA] <= 0)
  {
    setNextPC (getCurrentPC () +
        (signExtend17 (BrOff) <<
        (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

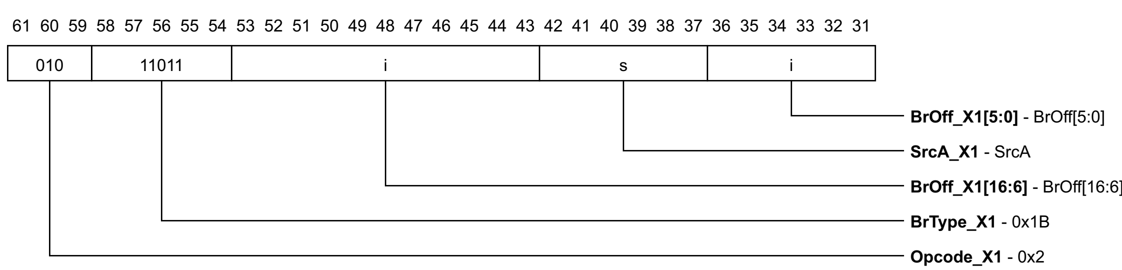

Figure 6-151: blez in X1 Bits Encoding

blezt

Branch Less Than or Equal to Zero Taken

Syntax

```
blezt SrcA, BrOff
```

Example

```
blezt r5, target
```

Description

Branches to the target if the source operand is less than or equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch less than or equal to zero predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register ZERO_REGISTER.

Functional Description

```
if (rf[SrcA] <= 0)
  {
    setNextPC (getCurrentPC () +
        (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

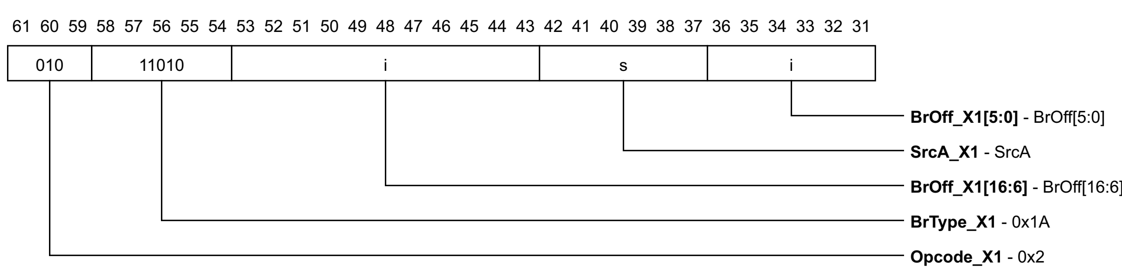

Figure 6-152: blezt in X1 Bits Encoding

bltz

Branch Less Than Zero

Syntax

```
bltz SrcA, BrOff
```

Example

```
bltz r5, target
```

Description

Branches to the target if the source operand is less than zero. Otherwise, the program counter advances to the next instruction in program order. Branch less than zero hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register ZERO_REGISTER.

Functional Description

```
if (rf[SrcA] < 0)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

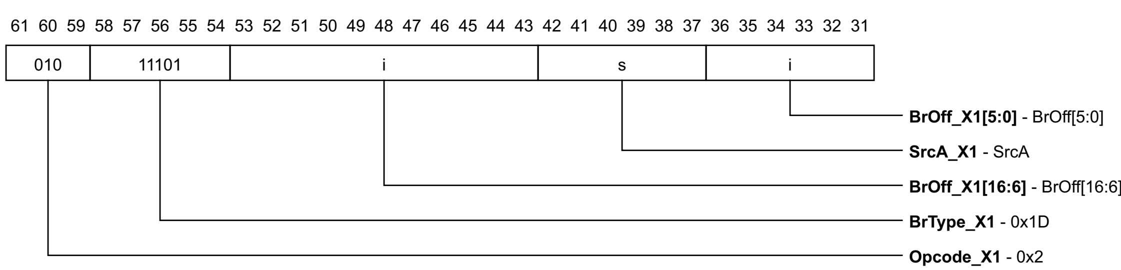

Figure 6-153: bltz in X1 Bits Encoding

bltzt

Branch Less Than Zero Taken

Syntax

```
bltzt SrcA, BrOff
```

Example

```
bltzt r5, target
```

Description

Branches to the target if the source operand is less than zero. Otherwise, the program counter advances to the next instruction in program order. Branch less than zero predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] < 0)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

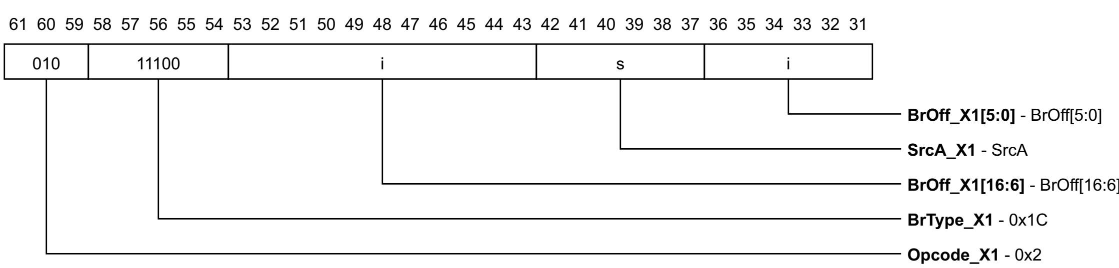

Figure 6-154: bltzt in X1 Bits Encoding

bnez

Branch Not Equal Zero

Syntax

```
bnez SrcA, BrOff
```

Example

```
bnez r5, target
```

Description

Branches to the target if the source operand is not equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch not zero hints to a branch prediction mechanism that the branch is not taken. This branch does an implicit move of the source operand to register `ZERO_REGISTER`.

Functional Description

```
if (rf[SrcA] != 0)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
       (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedIncorrect ();
  }
else
  {
    branchHintedCorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

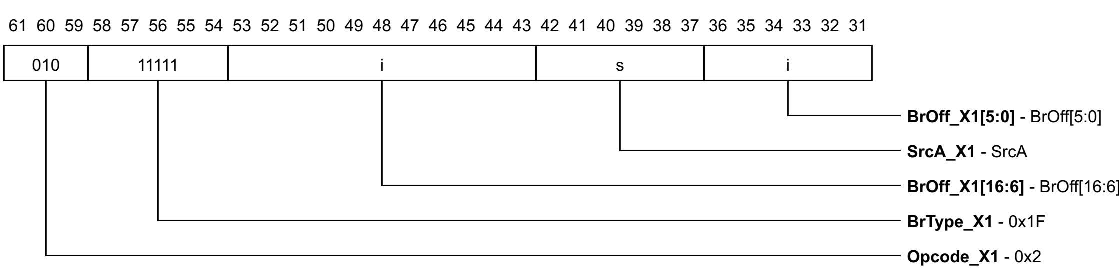

Figure 6-155: bnez in X1 Bits Encoding

bnezt

Branch Not Equal Zero Predict Taken

Syntax

```
bnezt SrcA, BrOff
```

Example

```
bnezt r5, target
```

Description

Branches to the target if the source operand is not equal to zero. Otherwise, the program counter advances to the next instruction in program order. Branch not zero predict taken hints to a branch prediction mechanism that the branch is taken. This branch does an implicit move of the source operand to register ZERO_REGISTER.

Functional Description

```
if (rf[SrcA] != 0)
  {
    setNextPC (getCurrentPC () +
       (signExtend17 (BrOff) <<
      (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
    branchHintedCorrect ();
  }
else
  {
    branchHintedIncorrect ();
  }

rf[ZERO_REGISTER] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

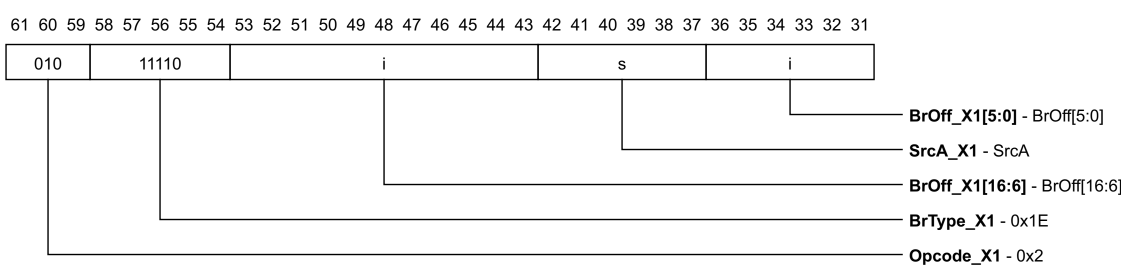

Figure 6-156: bnezt in X1 Bits Encoding

j

Jump

Syntax

```
j JumpOff
```

Example

```
j target
```

Description

Unconditionally jumps to a target. The jump hints to the prediction mechanism that this jump is taken.

Functional Description

```
setNextPC (getCurrentPC () +
   (signExtend27 (JumpOff) <<
    (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
jumped ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

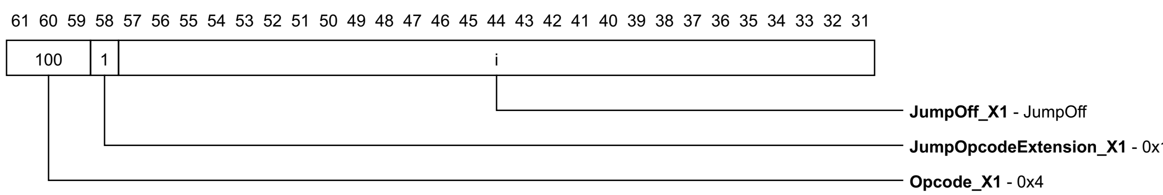

Figure 6-157: j in X1 Bits Encoding

jal

Jump and Link

Syntax

```
jal JumpOff
```

Example

```
jal target
```

Description

Unconditionally jumps to a target and puts the address of the subsequent instruction into register `LINK_REGISTER`. The jump hints to the prediction mechanism that this jump is taken. Signals to the hardware that it should attempt to push the link address on the return stack if available.

Functional Description

```
rf[LINK_REGISTER] = getCurrentPC () + (INSTRUCTION_SIZE / BYTE_SIZE);
pushReturnStack (getCurrentPC () + (INSTRUCTION_SIZE / BYTE_SIZE));
setNextPC (getCurrentPC () +
   (signExtend27 (JumpOff) <<
    (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
jumped ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

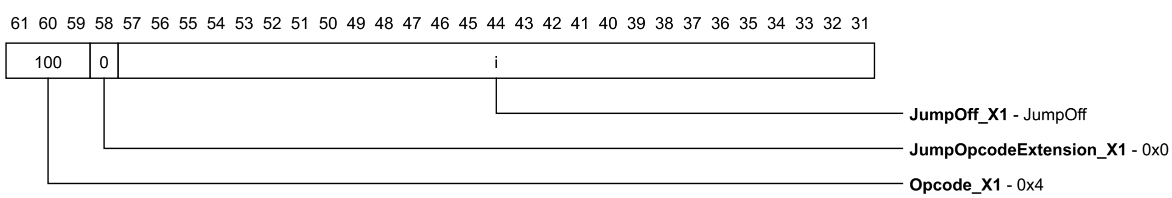

Figure 6-158: jal in X1 Bits Encoding

jalr

Jump and Link Register

Syntax

```
jalr SrcA
```

Example

```
jalr r5
```

Description

Unconditionally jumps to an address stored in a register and puts the address of the subsequent instruction into register `LINK_REGISTER`. Signals to the hardware that it should attempt to push the link address on the return stack if available.

Functional Description

```
rf[LINK_REGISTER] = getCurrentPC () + (INSTRUCTION_SIZE / BYTE_SIZE);
pushReturnStack (getCurrentPC () + (INSTRUCTION_SIZE / BYTE_SIZE));
setNextPC (rf[SrcA] & ALIGNED_INSTRUCTION_MASK);
indirectBranchHintedIncorrect ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X		X	

Encoding

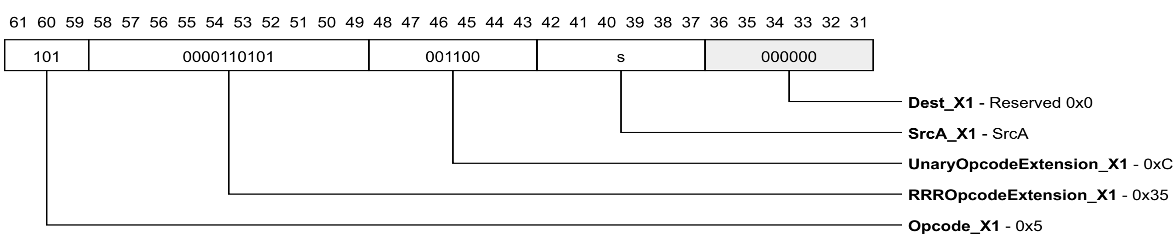

Figure 6-159: jalr in X1 Bits Encoding

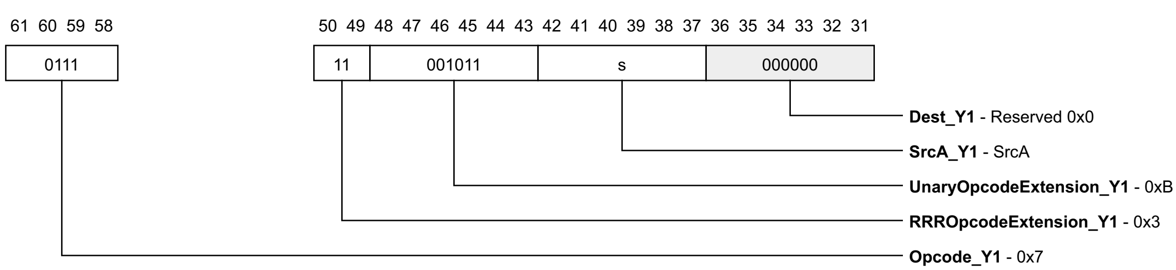

Figure 6-160: jalr inY1 Bits Encoding

jalrp

Jump and Link Register Predict

Syntax

```
jalrp SrcA
```

Example

```
jalrp r5
```

Description

Unconditionally jumps to an address stored in a register and puts the address of the subsequent instruction into register `LINK_REGISTER`. Signals to the hardware that it should attempt to predict the target with an address stack if available.

Functional Description

```
UnsignedMachineWord predictAddress = popReturnStack ();
rf[LINK_REGISTER] = getCurrentPC () + (INSTRUCTION_SIZE / BYTE_SIZE);
pushReturnStack (getCurrentPC () + (INSTRUCTION_SIZE / BYTE_SIZE));
setNextPC (rf[SrcA] & ALIGNED_INSTRUCTION_MASK);
if (predictAddress == (rf[SrcA] & ALIGNED_INSTRUCTION_MASK))
  {
    indirectBranchHintedCorrect ();
  }
else
  {
    indirectBranchHintedIncorrect ();
  }
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X		X	

Encoding

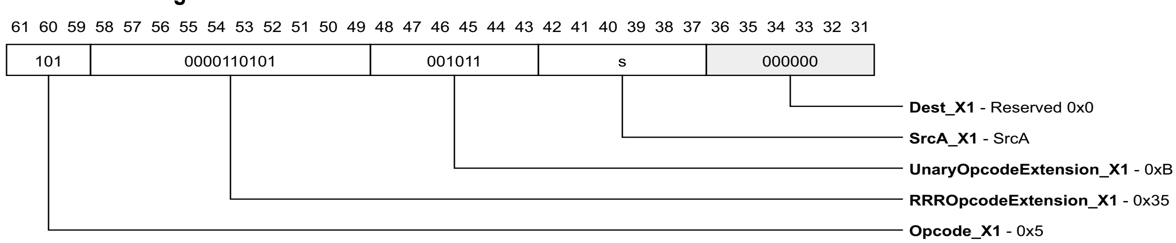

Figure 6-161: jalrp in X1 Bits Encoding

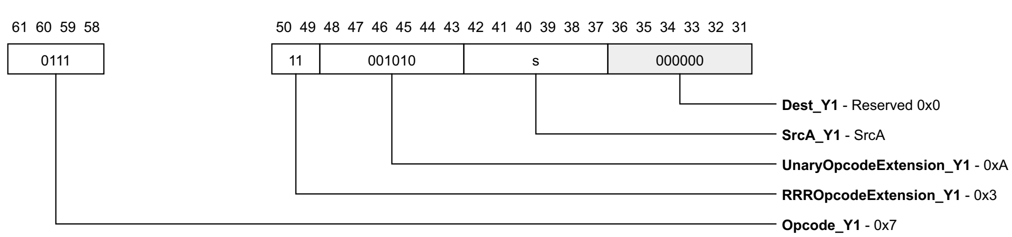

Figure 6-162: jalrp in Y1 Bits Encoding

jr

Jump Register

Syntax

```
jr SrcA
```

Example

```
jr r5
```

Description

Unconditionally jumps to an address stored in a register.

Functional Description

```
setNextPC (rf[SrcA] & ALIGNED_INSTRUCTION_MASK);
indirectBranchHintedIncorrect ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X		X	

Encoding

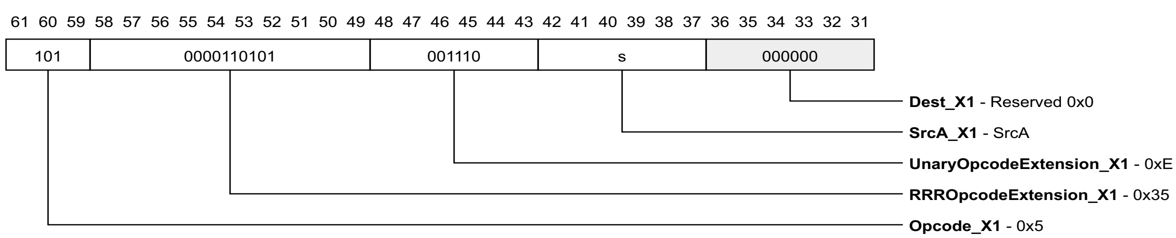

Figure 6-163: jr in X1 Bits Encoding

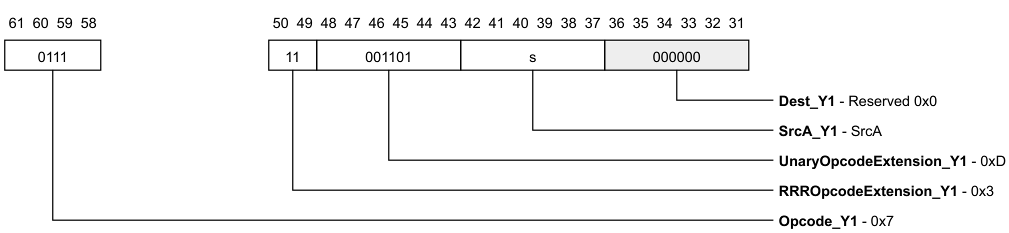

Figure 6-164: jr in Y1 Bits Encoding

jrp

Jump Register Predict

Syntax

```
jrp SrcA
```

Example

```
jrp r5
```

Description

Unconditionally jumps to an address stored in a register. Signals to the hardware that it should attempt to predict the target with an address stack if available.

Functional Description

```
setNextPC (rf[SrcA] & ALIGNED_INSTRUCTION_MASK);
if (popReturnStack () == (rf[SrcA] & ALIGNED_INSTRUCTION_MASK))
  {
    indirectBranchHintedCorrect ();
  }
else
  {
    indirectBranchHintedIncorrect ();
  }
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X		X	

Encoding

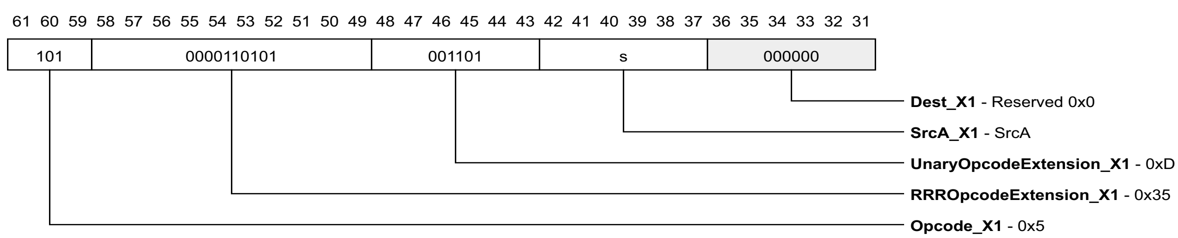

Figure 6-165: jrp in X1 Bits Encoding

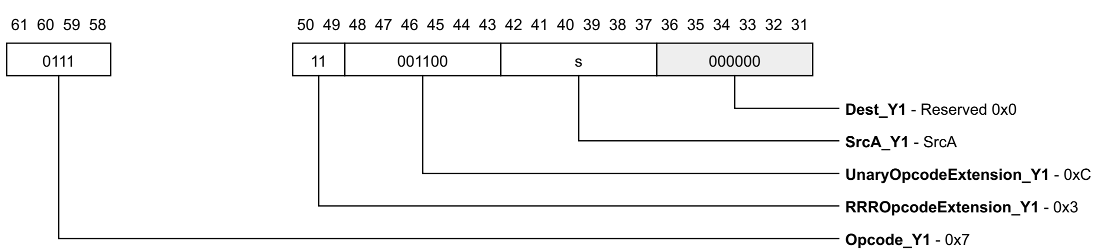

Figure 6-166: jrp in Y1 Bits Encoding

lnk

Link

Syntax

```
lnk Dest
```

Example

```
lnk r5
```

Description

Moves the address of the subsequent instruction into the destination operand. Does not effect the address stack if available.

Functional Description

```
rf[Dest] = getCurrentPC () + (INSTRUCTION_SIZE / BYTE_SIZE);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X		X	

Encoding

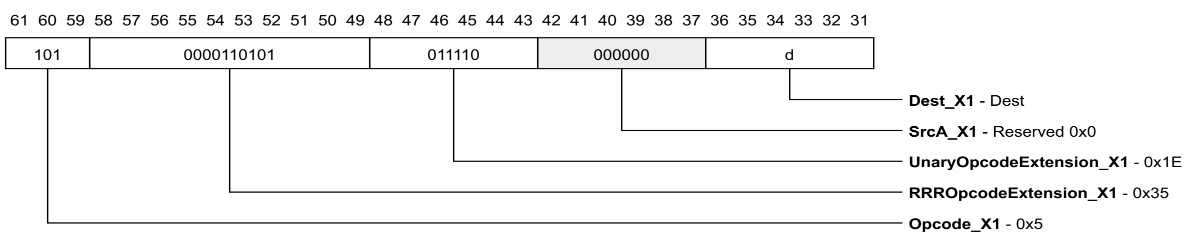

Figure 6-167: lnk in X1 Bits Encoding

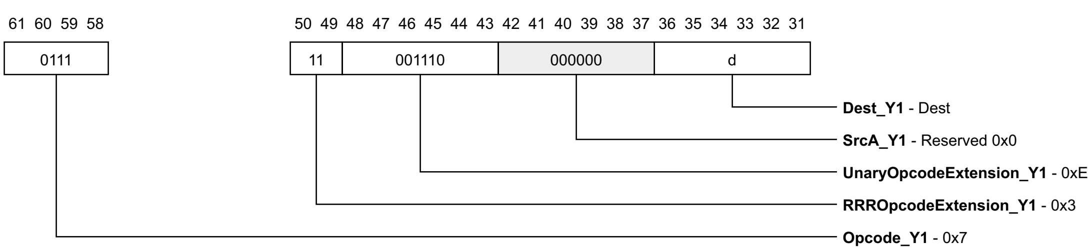

Figure 6-168: lnk in Y1 Bits Encoding

CHAPTER 7 FLOATING POINT INSTRUCTIONS

7.1 Overview

The following sections provide detailed descriptions of floating point instructions listed alphabetically.

fdouble_add_flags	Floating Point Double Precision Add Flags
fdouble_addsub	Floating Point Double Precision Add or Subtract
fdouble_mul_flags	Floating Point Double Precision Multiply Flags
fdouble_pack1	Floating Point Double Precision Pack Part 1
fdouble_pack2	Floating Point Double Precision Pack Part 2
fdouble_sub_flags	Floating Point Double Precision Subtract Flags
fdouble_unpack_max	Floating Point Double Precision Unpack Max
fdouble_unpack_min	Floating Point Double Precision Unpack Min
fsingle_add1	Floating Point Single Precision Add Part 1
fsingle_addsub2	Floating Point Single Precision Add or Subtract Part 2
fsingle_mul1	Floating Point Single Precision Multiply Part 1
fsingle_mul2	Floating Point Single Precision Multiply Part 2
fsingle_pack1	Floating Point Single Precision Pack Part 1
fsingle_pack2	Floating Point Single Precision Pack Part 2
fsingle_sub1	Floating Point Single Precision Subtract Part 1

The fsingle_add1, fsingle_sub1, fdouble_add_flags, and fdouble_sub_flags instructions set flags in the destination register, which are used to implement floating point comparison operators. The flags are described in Table 7-2.

Table 7-2. Floating Point Comparison Flags

Bit	Name	Description
25	unordered	The two operands are unordered.
26	lt	The first operand is less than the second.
27	le	The first operand is less than or equal to the second.

Table 7-2. Floating Point Comparison Flags (continued)

Bit	Name	Description
28	gt	The first operand is greater than the second.
29	ge	The first operand is greater than or equal to the second.
30	eq	The two operands are equal.
31	ne	The two operands are not equal.

7.2 Instructions

Floating point instructions are described in the sections that follow.

fdouble_add_flags

Floating Point Double Precision Add Flags

Syntax

```
fdouble_add_flags Dest, SrcA, SrcB
```

Example

```
fdouble_add_flags r5, r6, r7
```

Description

Compute the flags for a floating point double precision add. The flags are computed from the same operands as are used in `fdouble_unpack` instructions (`fdouble_unpack_max` or `fdouble_unpack_min`). The computed flags include the floating point comparison flags listed in Table 7-2 on page 135.

Functional Description

```
rf[Dest] = fdouble_addsub_flags (rf[SrcA], rf[SrcB], false);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

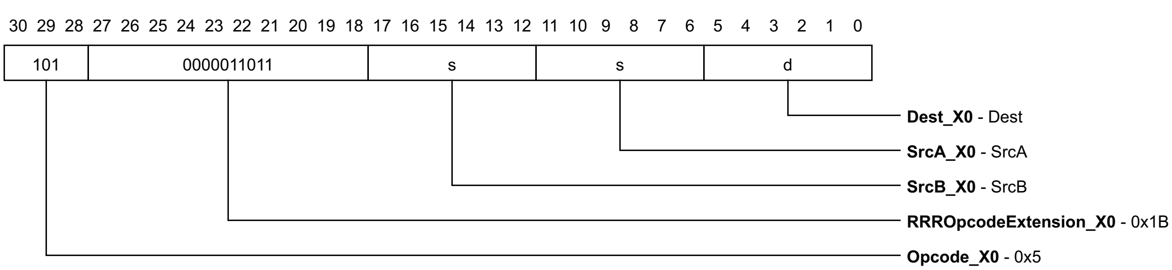

Figure 7-169: fdouble_add_flags in X0 Bits Encoding

fdouble_addsub

Floating Point Double Precision Add or Subtract

Syntax

```
fdouble_addsub Dest, SrcA, SrcB
```

Example

```
fdouble_addsub r5, r6, r7
```

Description

Performs the floating point double precision add or subtract after the arguments are unpacked and the flags are computed. The first operand is the unpacked addend with smaller magnitude, and the second operand is the computed flags.

Functional Description

```
rf[Dest] = fdouble_addsub (rf[Dest], rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

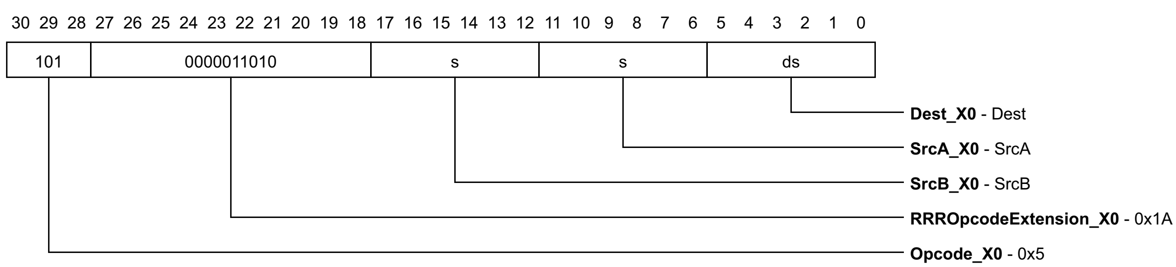

Figure 7-170: fdouble_addsub in X0 Bits Encoding

fdouble_mul_flags

Floating Point Double Precision Multiply Flags

Syntax

```
fdouble_mul_flags Dest, SrcA, SrcB
```

Example

```
fdouble_mul_flags r5, r6, r7
```

Description

Compute the flags for a floating point double precision multiply. The flags are computed from the same operands as are used in `fdouble_unpack` instructions (`fdouble_unpack_max` or `fdouble_unpack_min`).

Functional Description

```
rf[Dest] = fdouble_mul_flags (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

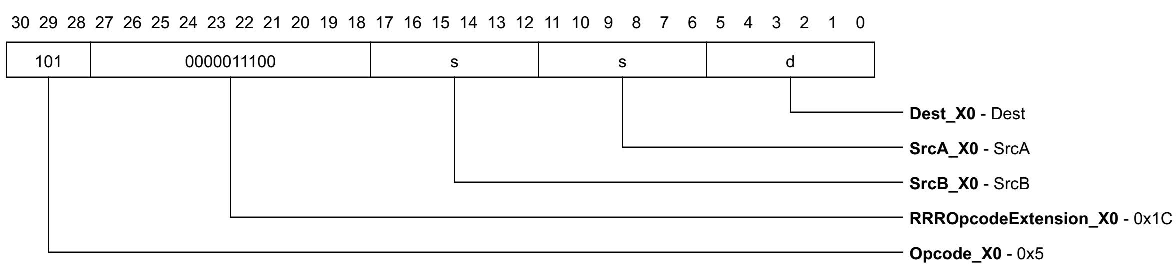

Figure 7-171: fdouble_mul_flags in X0 Bits Encoding

fdouble_pack1

Floating Point Double Precision Pack Part 1

Syntax

```
fdouble_pack1 Dest, SrcA, SrcB
```

Example

```
fdouble_pack1 r5, r6, r7
```

Description

Performs the first part of a floating point double precision normalize and pack.

Functional Description

```
rf[Dest] = fdouble_pack1 (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

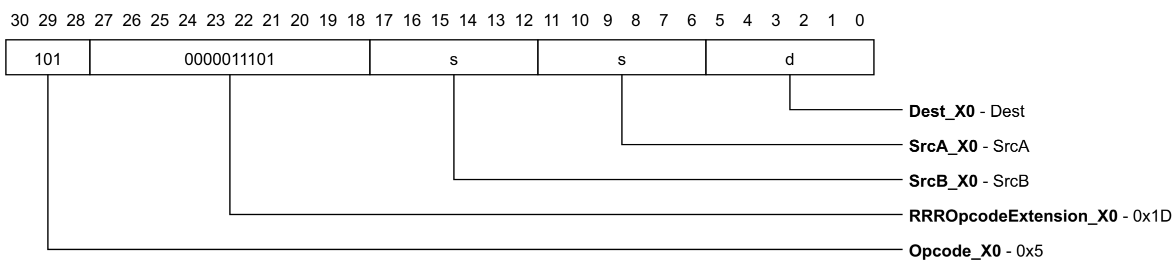

Figure 7-172: fdouble_pack1 in X0 Bits Encoding

fdouble_pack2

Floating Point Double Precision Pack Part 2

Syntax

```
fdouble_pack2 Dest, SrcA, SrcB
```

Example

```
fdouble_pack2 r5, r6, r7
```

Description

Performs the second and final part of a floating point double precision normalize and pack.

Functional Description

```
rf[Dest] = fdouble_pack2 (rf[Dest], rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

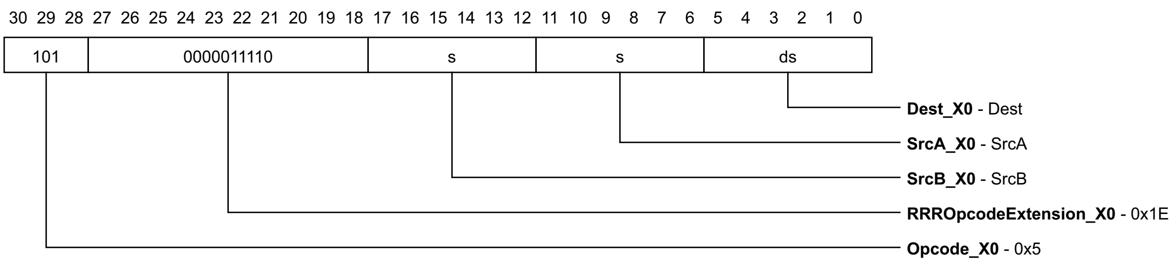

Figure 7-173: fdouble_pack2 in X0 Bits Encoding

fdouble_sub_flags

Floating Point Double Precision Subtract Flags

Syntax

```
fdouble_sub_flags Dest, SrcA, SrcB
```

Example

```
fdouble_sub_flags r5, r6, r7
```

Description

Compute the flags for a floating point double precision subtract. The flags are computed from the same operands as are used in `fdouble_unpack` instructions (`fdouble_unpack_max` or `fdouble_unpack_min`). The computed flags include the floating point comparison flags listed in Table 7-2 on page 135.

Functional Description

```
rf[Dest] = fdouble_addsub_flags (rf[SrcA], rf[SrcB], true);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

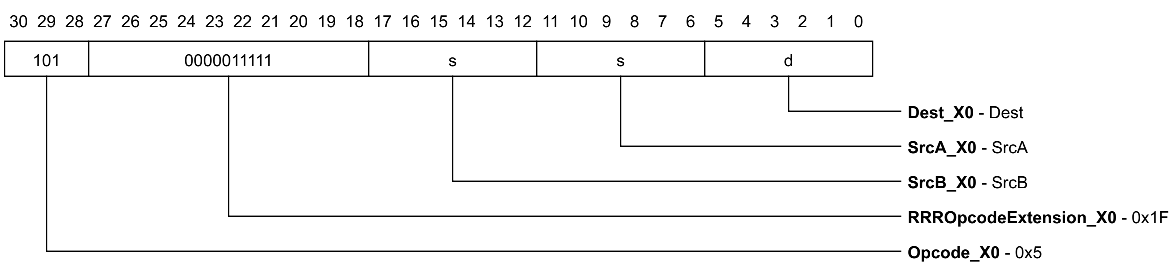

Figure 7-174: fdouble_sub_flags in X0 Bits Encoding

fdouble_unpack_max

Floating Point Double Precision Unpack Max

Syntax

```
fdouble_unpack_max Dest, SrcA, SrcB
```

Example

```
fdouble_unpack_max r5, r6, r7
```

Description

Extracts the mantissa of the source operand, which has the largest magnitude.

Functional Description

```
rf[Dest] = fdouble_unpack_minmax (rf[SrcA], rf[SrcB], false);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

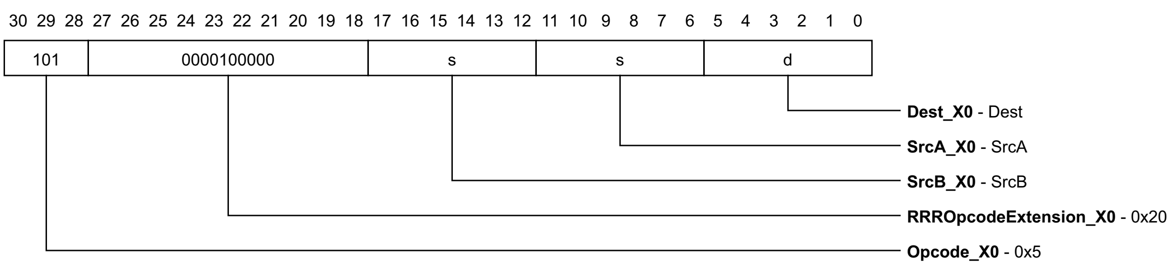

Figure 7-175: fdouble_unpack_max in X0 Bits Encoding

fdouble_unpack_min

Floating Point Double Precision Unpack Min

Syntax

```
fdouble_unpack_min Dest, SrcA, SrcB
```

Example

```
fdouble_unpack_min r5, r6, r7
```

Description

Extracts the mantissa of the source operand, which has the smallest magnitude.

Functional Description

```
rf[Dest] = fdouble_unpack_minmax (rf[SrcA], rf[SrcB], true);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

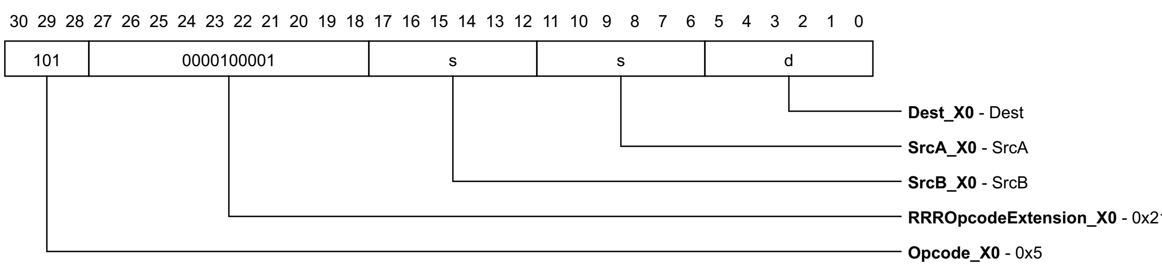

Figure 7-176: fdouble_unpack_min in X0 Bits Encoding

fsingle_add1

Floating Point Single Precision Add Part 1

Syntax

```
fsingle_add1 Dest, SrcA, SrcB
```

Example

```
fsingle_add1 r5, r6, r7
```

Description

Performs the first part of a floating point single precision add. This instruction also sets the floating point comparison flags in the destination register (see Table 7-2 on page 135).

Functional Description

```
rf[Dest] = fsingle_addsub1 (rf[SrcA], rf[SrcB], false);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

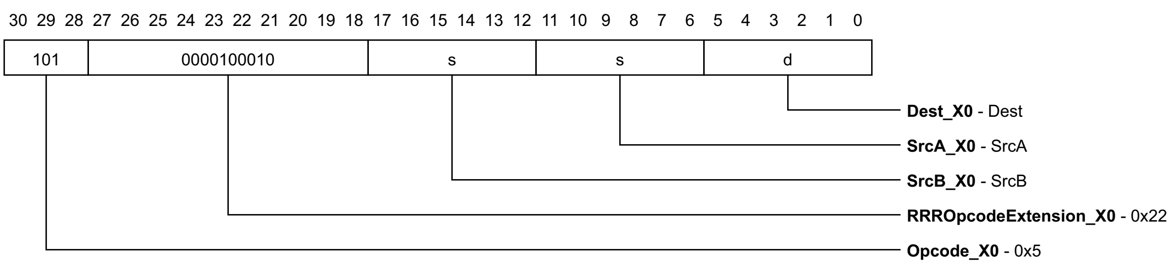

Figure 7-177: fsingle_add1 in X0 Bits Encoding

fsingle_addsub2

Floating Point Single Precision Add or Subtract Part 2

Syntax

```
fsingle_addsub2 Dest, SrcA, SrcB
```

Example

```
fsingle_addsub2 r5, r6, r7
```

Description

Performs the second part of a floating point single precision add or subtract.

Functional Description

```
rf[Dest] = fsingle_addsub2 (rf[Dest], rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

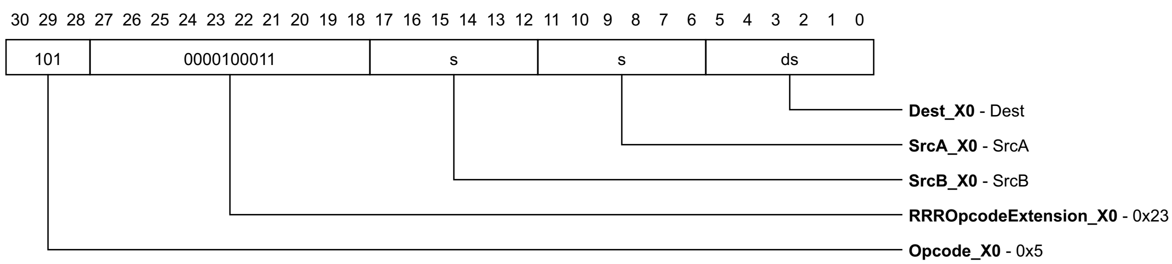

Figure 7-178: fsingle_addsub2 in X0 Bits Encoding

fsingle_mul1

Floating Point Single Precision Multiply Part 1

Syntax

```
fsingle_mul1 Dest, SrcA, SrcB
```

Example

```
fsingle_mul1 r5, r6, r7
```

Description

Performs the first part of a floating point single precision multiply.

Functional Description

```
rf[Dest] = fsingle_mul1 (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

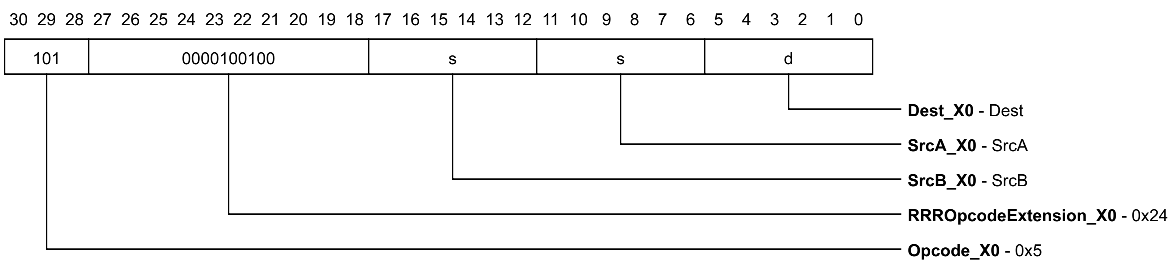

Figure 7-179: fsingle_mul1 in X0 Bits Encoding

fsingle_mul2

Floating Point Single Precision Multiply Part 2

Syntax

```
fsingle_mul2 Dest, SrcA, SrcB
```

Example

```
fsingle_mul2 r5, r6, r7
```

Description

Performs the first part of a floating point single precision multiply.

Functional Description

```
rf[Dest] = fsingle_mul2 (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

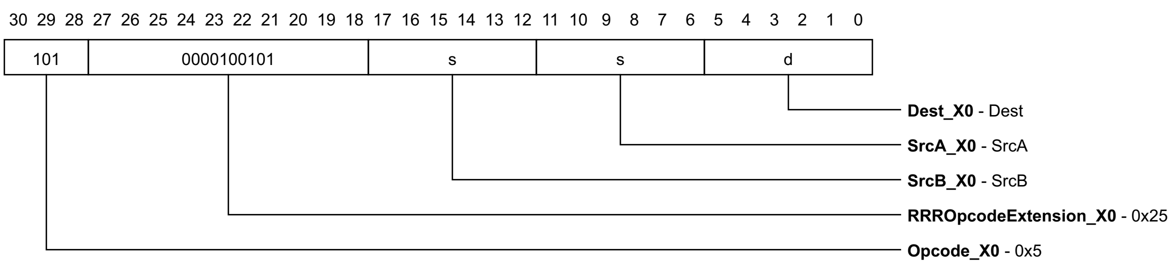

Figure 7-180: fsingle_mul2 in X0 Bits Encoding

fsingle_pack1

Floating Point Single Precision Pack Part 1

Syntax

```
fsingle_pack1 Dest, SrcA
```

Example

```
fsingle_pack1 r5, r6
```

Description

Performs the first part of a floating point single precision normalize and pack.

Functional Description

```
rf[Dest] = fsingle_pack1 (rf[SrcA]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

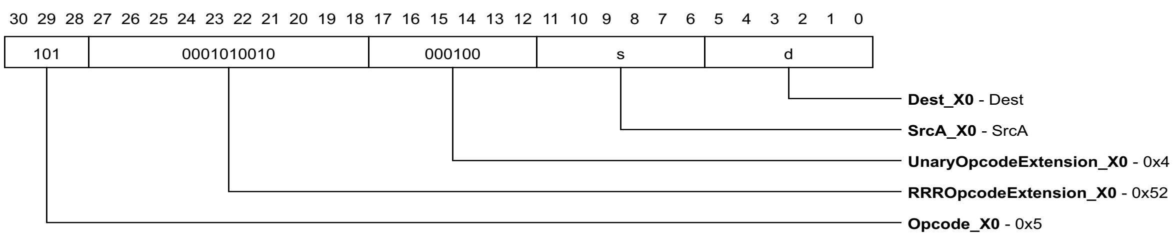

Figure 7-181: fsingle_pack1 in X0 Bits Encoding

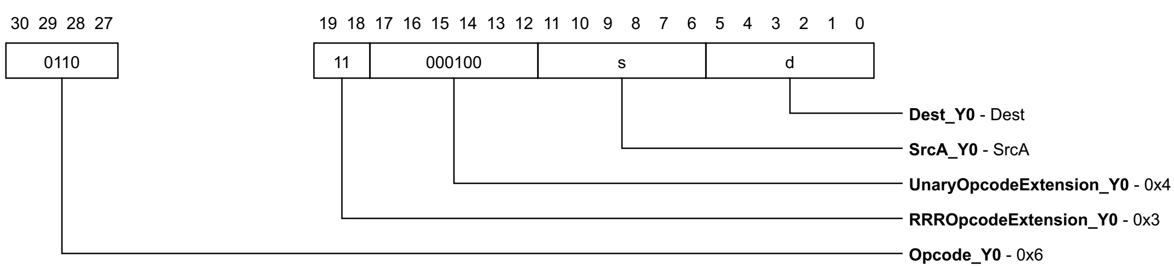

Figure 7-182: fsingle_pack1 in Y0 Bits Encoding

fsingle_pack2

Floating Point Single Precision Pack Part 2

Syntax

```
fsingle_pack2 Dest, SrcA, SrcB
```

Example

```
fsingle_pack2 r5, r6, r7
```

Description

Performs the second and final part of a floating point single precision normalize and pack.

Functional Description

```
rf[Dest] = fsingle_pack2 (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

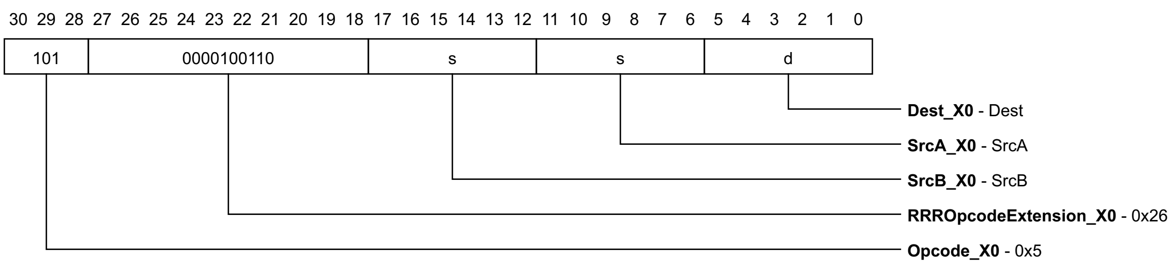

Figure 7-183: fsingle_pack2 in X0 Bits Encoding

fsingle_sub1

Floating Point Single Precision Subtract Part 1

Syntax

```
fsingle_sub1 Dest, SrcA, SrcB
```

Example

```
fsingle_sub1 r5, r6, r7
```

Description

Performs the first part of a floating point single precision subtract. This instruction also sets the floating point comparison flags in the destination register (see Table 7-2 on page 135).

Functional Description

```
rf[Dest] = fsingle_addsub1 (rf[SrcA], rf[SrcB], true);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

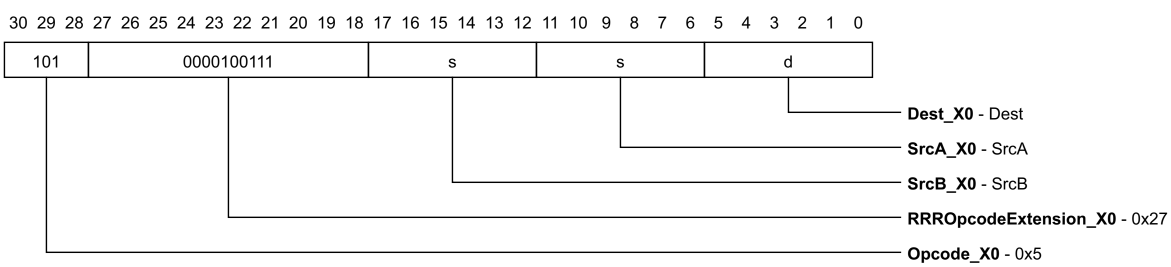

Figure 7-184: fsingle_sub1 in X0 Bits Encoding

CHAPTER 8 LOGICAL INSTRUCTIONS

8.1 Overview

The following sections provide detailed descriptions of logical instructions listed alphabetically.

and	And
andi	And Immediate
bfexts	Bit Field Extract Signed
bfextu	Bit field Extract Unsigned
bfins	Bit field Insert
cmoveqz	Conditional Move If Equal Zero
cmovnez	Conditional Move If Not Equal Zero
mm	Masked Merge
mnz	Mask Not Zero
mz	Mask Zero
nor	Nor
or	Or
ori	Or Immediate
rotl	Rotate Left
rotli	Rotate Left Immediate
shl	Shift Left
shli	Shift Left Immediate
shlx	Shift Left and Extend
shlxi	Shift Left and Extend Immediate
shrs	Shift Right Signed
shrsi	Shift Right Signed Immediate
shru	Shift Right Unsigned
shrui	Shift Right Unsigned Immediate
shrux	Shift Right Unsigned and Extend
shruxi	Shift Right Unsigned and Extend Immediate
tblidxb0	Table Index Byte 0

8.2 Instructions

Logical instructions are described in the sections that follow.

and

And

Syntax

```
and Dest, SrcA, SrcB
```

Example

```
and r5, r6, r7
```

Description

Compute the boolean AND of two operands.

Functional Description

```
rf[Dest] = rf[SrcA] & rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

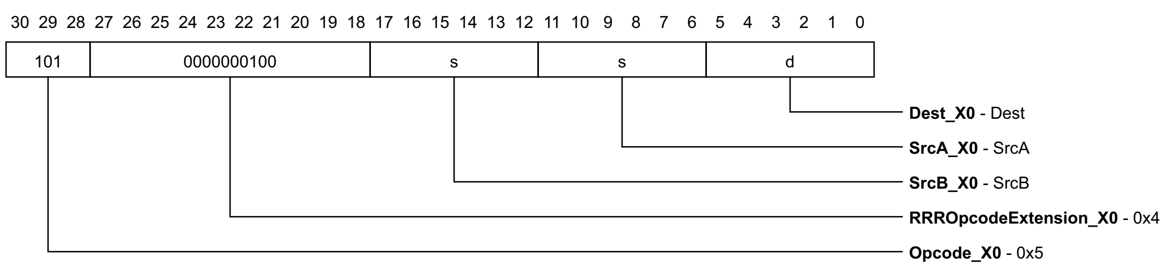

Figure 8-185: and in X0 Bits Encoding

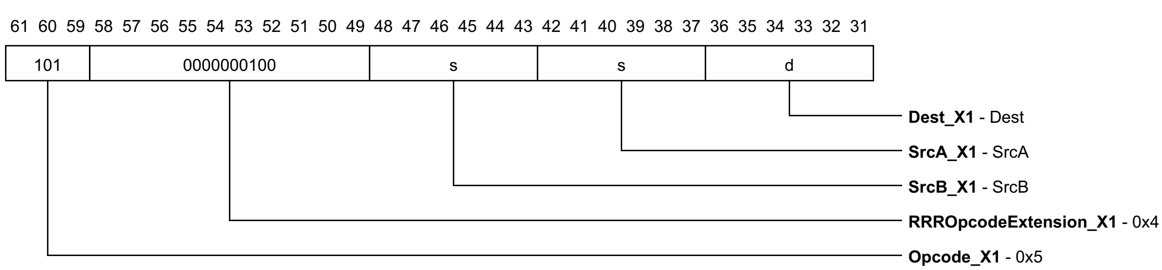

Figure 8-186: and in X1 Bits Encoding

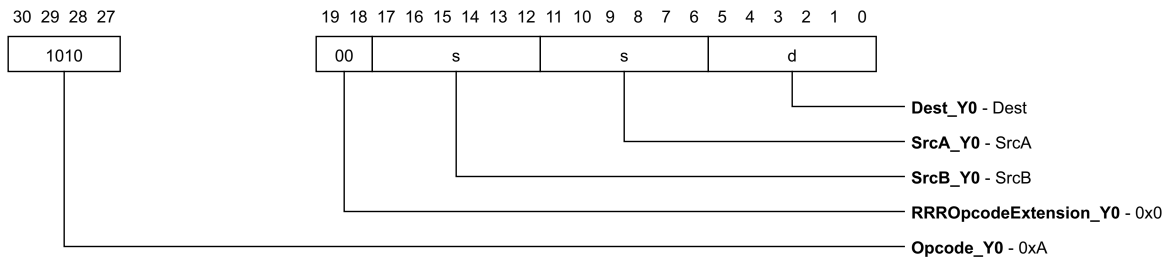

Figure 8-187: and in Y0 Bits Encoding

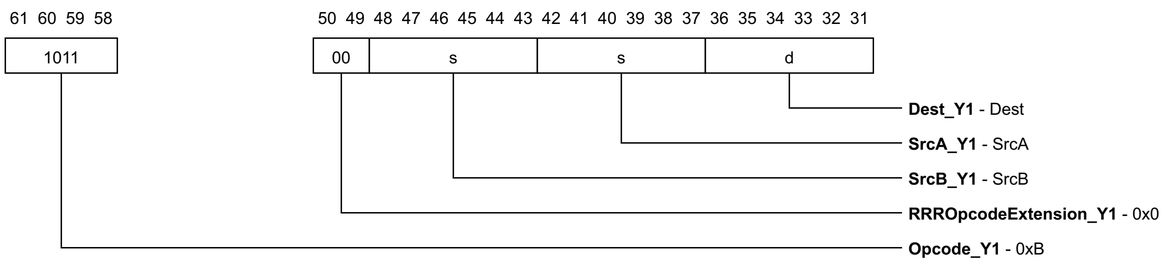

Figure 8-188: and in Y1 Bits Encoding

andi

And Immediate

Syntax

```
andi Dest, SrcA, Imm8
```

Example

```
andi r5, r6, 5
```

Description

Compute the boolean AND of an operand and a sign extended immediate.

Functional Description

```
rf[Dest] = rf[SrcA] & signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

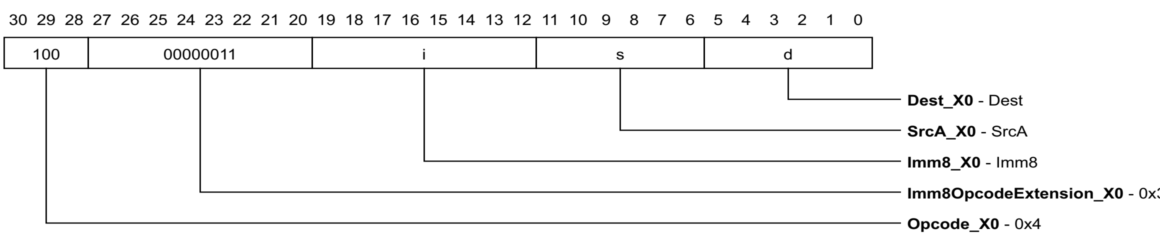

Figure 8-189: andi in X0 Bits Encoding

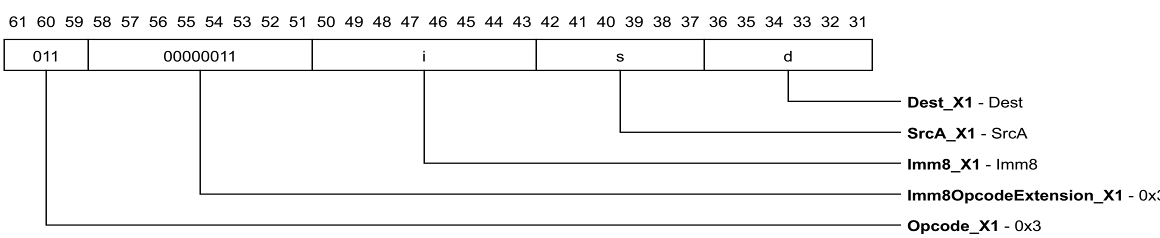

Figure 8-190: andi in X1 Bits Encoding

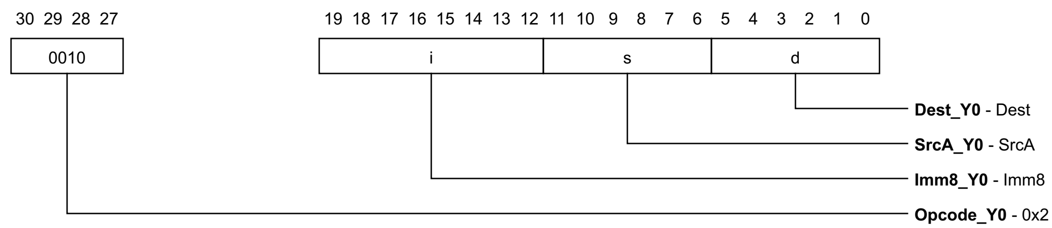

Figure 8-191: andi in Y0 Bits Encoding

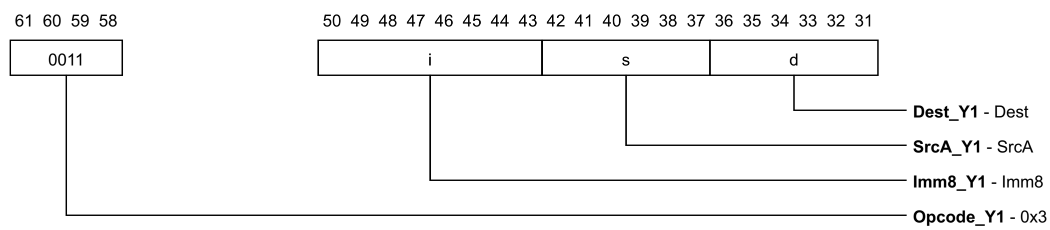

Figure 8-192: andi in Y1 Bits Encoding

bfexts

Bit Field Extract Signed

Syntax

```
bfexts Dest, SrcA, BFStart, BFEnd
```

Example

```
bfexts r5, r6, 5, 7
```

Description

Extract, right justify and sign-extend the specified bit field of the destination operand. The bit field is specified by the `BFStart` and `BFEnd` immediate operands, which contain the bit fields starting and ending bit positions inclusive. If the start position is less than or equal to the end position, then the field contains bits from start bit position up to and including the ending bit position. If the start position is greater than the end position, then the field contains the bits start bit position up to the `WORD_SIZE` bit position, and from the zero bit position up to the end bit position.

Functional Description

```
uint64_t mask = 0;
int64_t background = ((rf[SrcA] >> BFEnd) & 1) ? -1ULL : 0ULL;
mask = ((-1ULL) ^ ((-1ULL << ((BFEnd - BFStart) & 63)) << 1));
uint64_t rot_src =
  (((uint64_t) rf[SrcA]) >> BFStart) | (rf[SrcA] << (64 - BFStart));
rf[Dest] = (rot_src & mask) | (background & ~mask);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

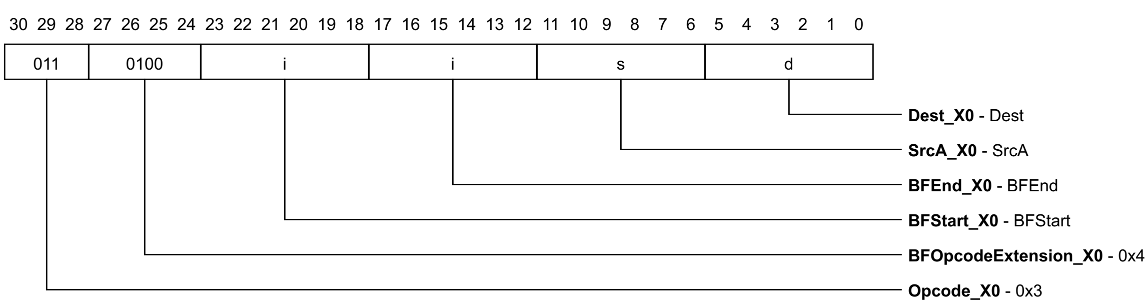

Figure 8-193: bfexts in X0 Bits Encoding

bfextu

Bit field Extract Unsigned

Syntax

```
bfextu Dest, SrcA, BFStart, BFEnd
```

Example

```
bfextu r5, r6, 5, 7
```

Description

Extract and right justify the specified bit field of the destination operand. The bit field is specified by the `BFStart` and `BFEnd` immediate operands, which contain the bit fields starting and ending bit positions. If the start position is less than or equal to the end position, then the field contains bits from start bit position up to and including the ending bit position. If the start position is greater than the end position, then the field contains the bits start bit position up to the `WORD_SIZE` bit position, and from the zero bit position up to the end bit position.

Functional Description

```
uint64_t mask = 0;
mask = ((-1ULL) ^ ((-1ULL << ((BFEnd - BFStart) & 63)) << 1));
uint64_t rot_src =
  (((uint64_t) rf[SrcA]) >> BFStart) | (rf[SrcA] << (64 - BFStart));
rf[Dest] = rot_src & mask;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

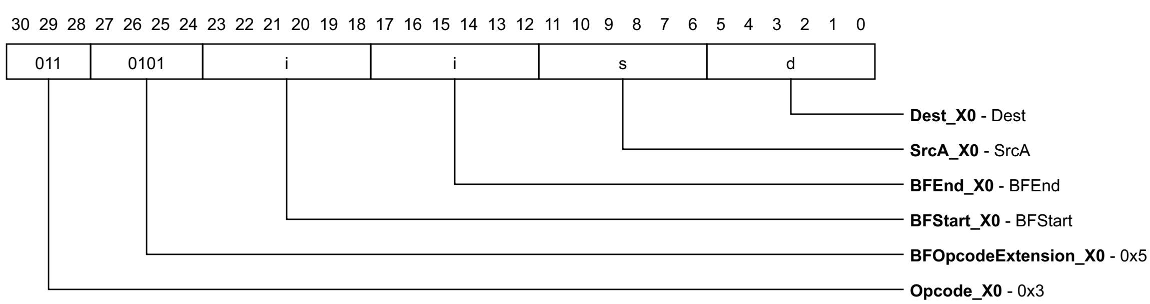

Figure 8-194: bfextu in X0 Bits Encoding

bfins

Bit field Insert

Syntax

```
bfins Dest, SrcA, BFStart, BFEnd
```

Example

```
bfins r5, r6, 5, 7
```

Description

Insert the low-order bits of the source operand into the specified bit field of the destination operand. The bit field is specified by the `BFStart` and `BFEnd` immediate operands, which contain the bit fields starting and ending bit positions. If the start position is less than or equal to the end position, then the field contains bits from start bit position up to and including the ending bit position. If the start position is greater than the end position, then the field contains the bits start bit position up to the `WORD_SIZE` bit position, and from the zero bit position up to the end bit position.

Functional Description

```
uint64_t mask = 0;
int start;
int end;
start = BFStart;
end = BFEnd;
mask =
  (start <=
   end) ? ((-1ULL << start) ^ ((-1ULL << end) << 1)) : ((-1ULL << start) |
                                                   (-1ULL >>
                                                    ((64 - 1) - end)));
uint64_t rot_src =
  (rf[SrcA] << start) | ((uint64_t) rf[SrcA] >> (64 - start));
rf[Dest] = (rot_src & mask) | (rf[Dest] & (-1ULL ^ mask));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

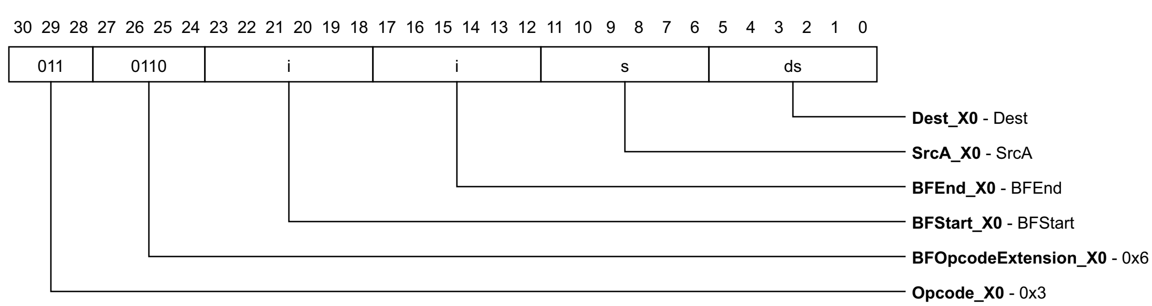

Figure 8-195: bfins in X0 Bits Encoding

cmoveqz

Conditional Move If Equal Zero

Syntax

```
cmoveqz Dest, SrcA, SrcB
```

Example

```
cmoveqz r5, r6, r7
```

Description

If the first source operand is zero, move the second operand to the destination. Otherwise, move the contents of the destination register to the destination. This instruction unconditionally reads the first input operand, the second input operand, and the destination operand.

Functional Description

```
uint64_t localSrcB = rf[SrcB];
uint64_t localDest = rf[Dest];
rf[Dest] = (rf[SrcA] == 0) ? (localSrcB) : (localDest);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

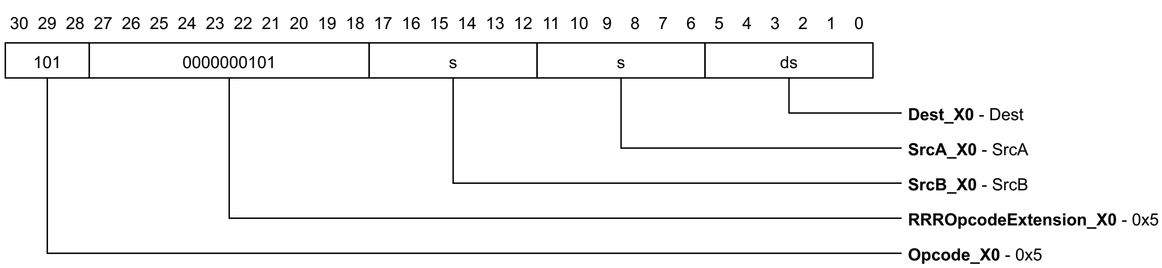

Figure 8-196: cmoveqz in X0 Bits Encoding

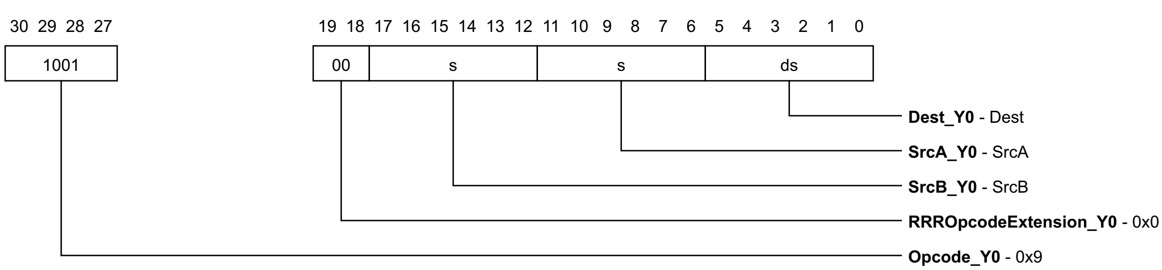

Figure 8-197: cmoveqz in Y0 Bits Encoding

cmovnez

Conditional Move If Not Equal Zero

Syntax

```
cmovnez Dest, SrcA, SrcB
```

Example

```
cmovnez r5, r6, r7
```

Description

If the first source operand is not zero, move the second operand to the destination. Otherwise, move the contents of the destination register to the destination. This instruction unconditionally reads the first input operand, the second input operand, and the destination operand.

Functional Description

```
uint64_t localSrcB = rf[SrcB];
uint64_t localDest = rf[Dest];
rf[Dest] = (rf[SrcA] != 0) ? (localSrcB) : (localDest);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

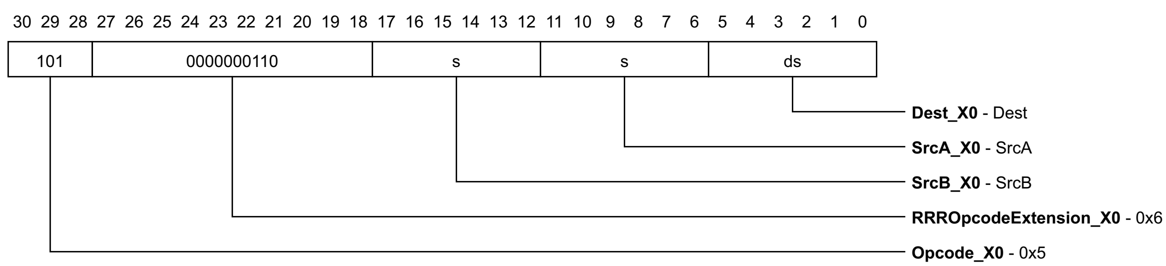

Figure 8-198: cmovnez in X0 Bits Encoding

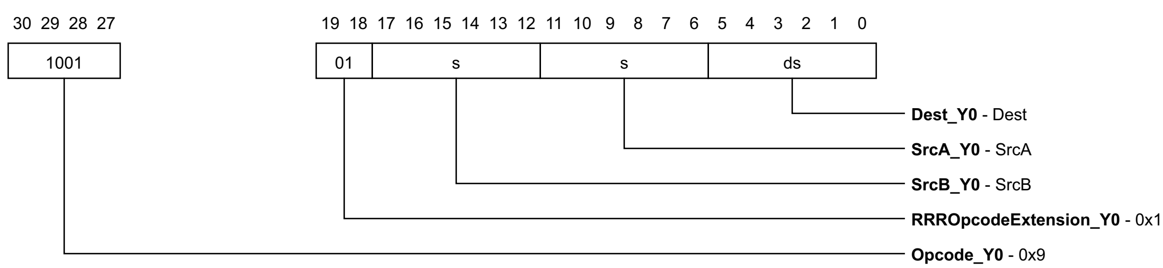

Figure 8-199: cmovnez in Y0 Bits Encoding

mm

Masked Merge

Syntax

```
mm Dest, SrcA, BFStart, BFEnd
```

Example

```
mm r5, r6, 5, 7
```

Description

Merge source operand and the destination operand based off of a running mask. The mask is specified by the `BFStart` and `BFEnd` fields, which contain the mask's starting and ending bit positions inclusive. If the start position is less than or equal to the end position, then the mask contains bits set (1) from start bit position up to and including the ending bit position. If the start position is greater than the end position, then the mask contains the bits set (1) from the start bit position up to the `WORD_SIZE` bit position, and from the zero bit position up to the end bit position. The mask selects bits out of the destination operand and the inverse of the mask selects bits out of the source operand.

Functional Description

```
uint64_t mask = 0;
int start;
int end;
start = BFStart;
end = BFEnd;
mask =
  (start <=
   end) ? ((-1ULL << start) ^ ((-1ULL << end) << 1)) : ((-1ULL << start) |
                                                        (-1ULL >>
                                                         ((64 - 1) - end)));
rf[Dest] = (rf[Dest] & mask) | (rf[SrcA] & (-1ULL ^ mask));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

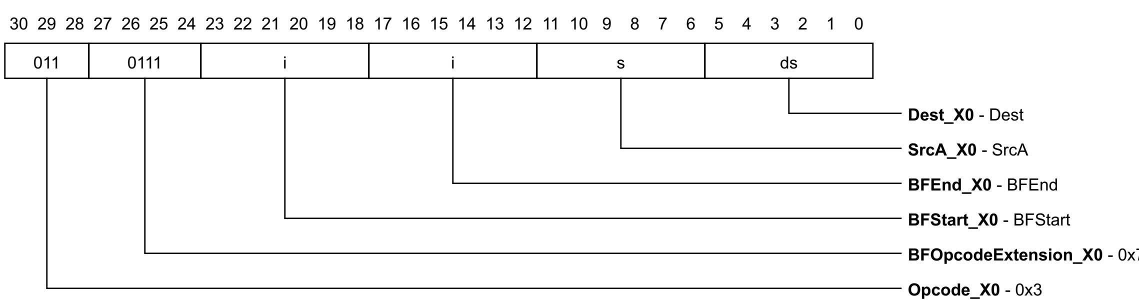

Figure 8-200: mm in X0 Bits Encoding

mnz

Mask Not Zero

Syntax

```
mnz Dest, SrcA, SrcB
```

Example

```
mnz r5, r6, r7
```

Description

If the first operand is not zero, then compute the boolean AND of the second operand and a value of all ones (1's), otherwise return zero (0).

Functional Description

```
rf[Dest] = signExtend1 ((rf[SrcA] != 0) ? 1 : 0) & rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

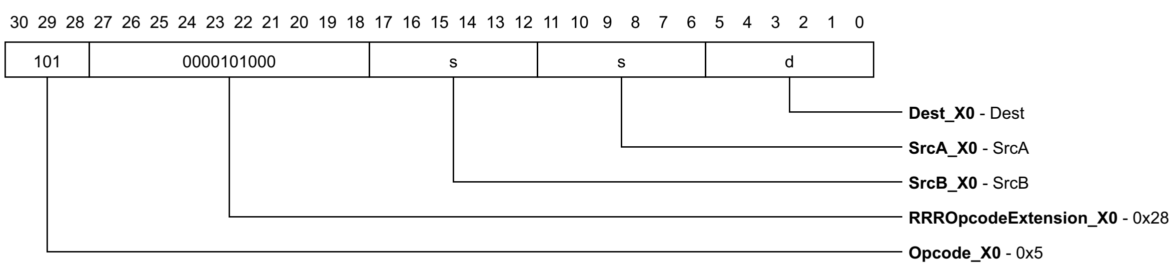

Figure 8-201: mnz in X0 Bits Encoding

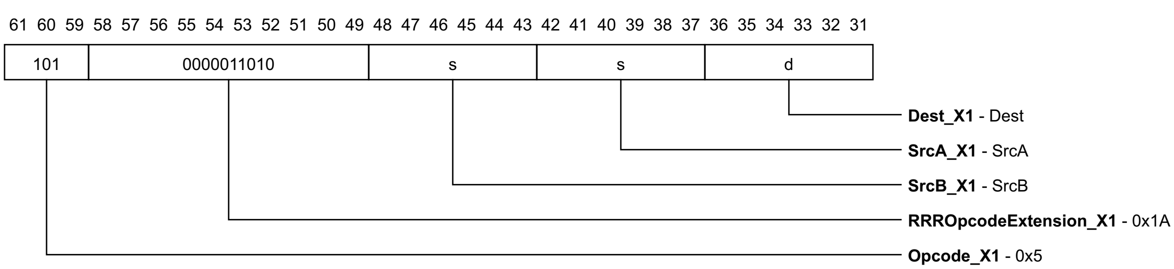

Figure 8-202: mnz in X1 Bits Encoding

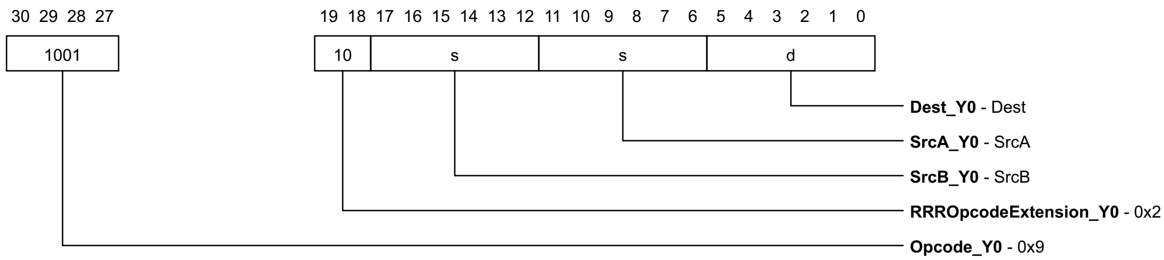

Figure 8-203: mnz in Y0 Bits Encoding

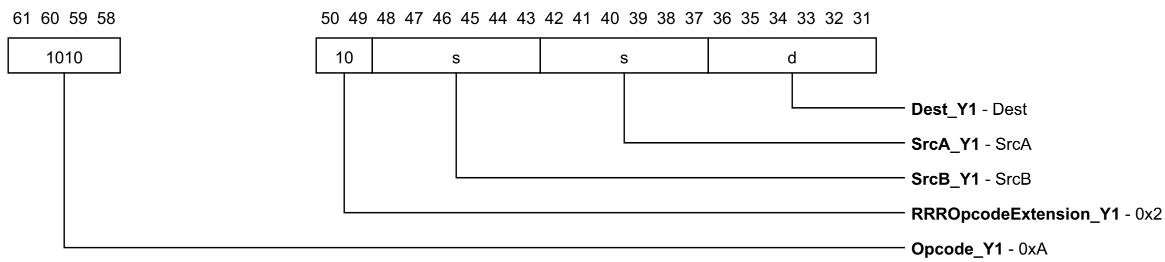

Figure 8-204: mnz in Y1 Bits Encoding

mz

Mask Zero

Syntax

```
mz Dest, SrcA, SrcB
```

Example

```
mz r5, r6, r7
```

Description

If the first operand is zero, then compute the boolean AND of the second operand and a value of all ones (1's), otherwise return zero (0).

Functional Description

```
rf[Dest] = signExtend1 ((rf[SrcA] == 0) ? 1 : 0) & rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

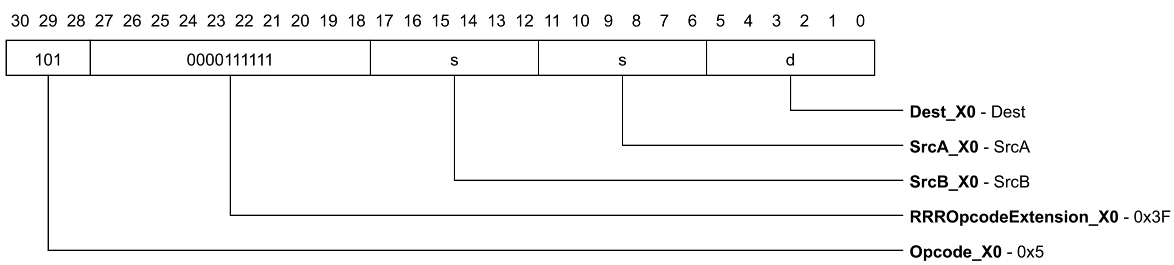

Figure 8-205: mz in X0 Bits Encoding

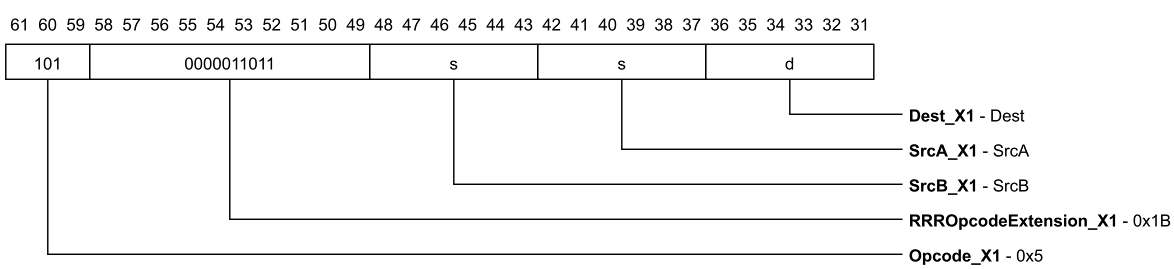

Figure 8-206: mz in X1 Bits Encoding

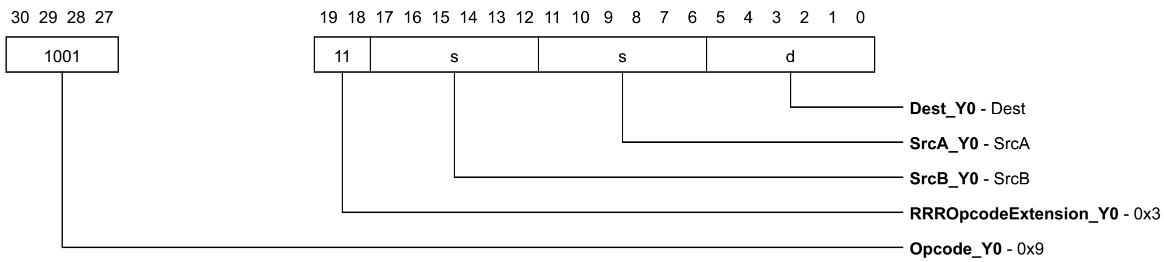

Figure 8-207: mz in Y0 Bits Encoding

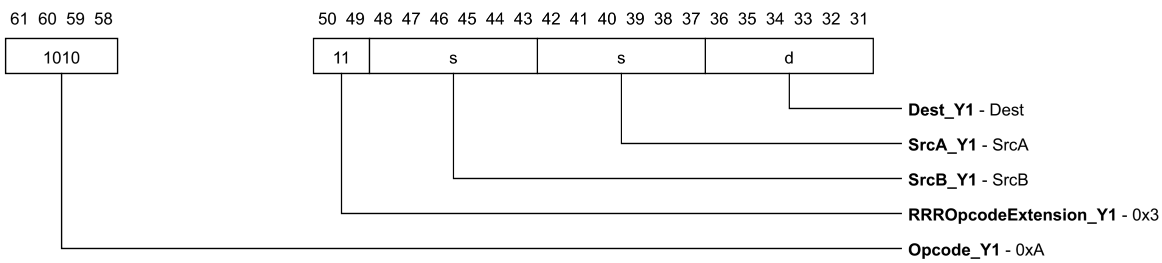

Figure 8-208: mz in Y1 Bits Encoding

nor

Nor

Syntax

```
nor Dest, SrcA, SrcB
```

Example

```
nor r5, r6, r7
```

Description

Computer the boolean NOR of two operands.

Functional Description

```
rf[Dest] = ~(rf[SrcA] | rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

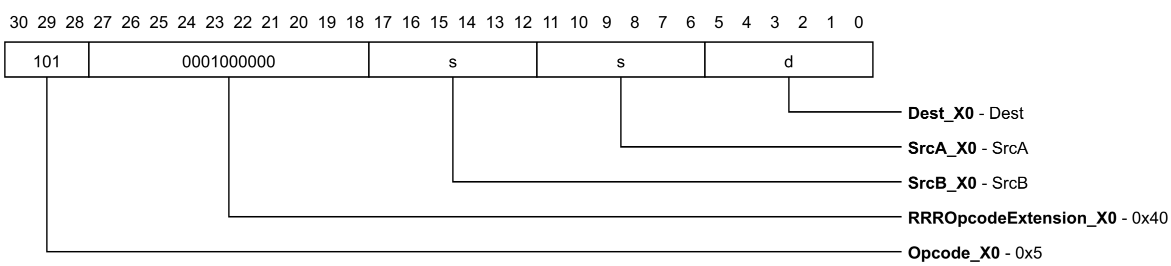

Figure 8-209: nor in X0 Bits Encoding

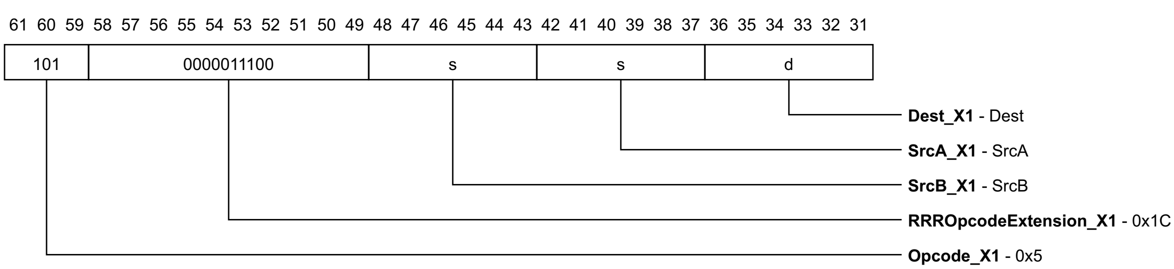

Figure 8-210: nor in X1 Bits Encoding

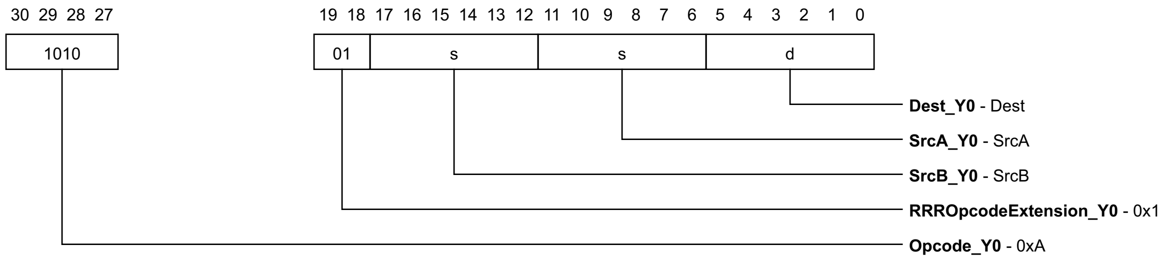

Figure 8-211: nor in Y0 Bits Encoding

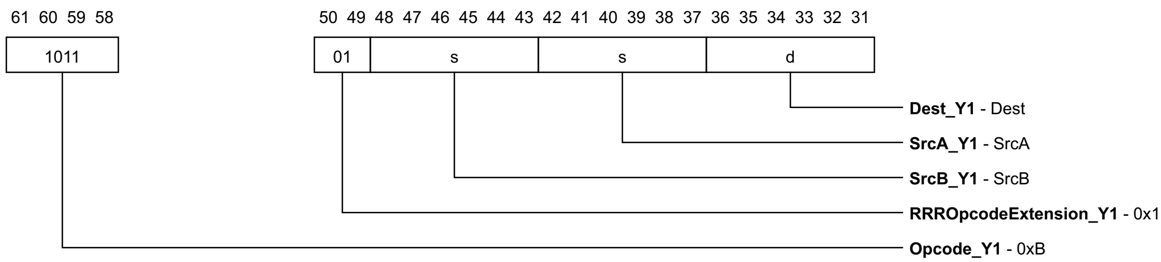

Figure 8-212: nor in Y1 Bits Encoding

or

Or

Syntax

```
or Dest, SrcA, SrcB
```

Example

```
or r5, r6, r7
```

Description

Compute the boolean OR of two operands.

Functional Description

```
rf[Dest] = rf[SrcA] | rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

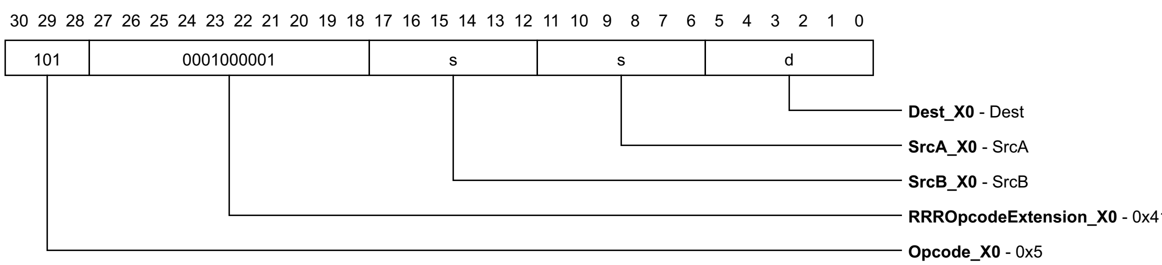

Figure 8-213: or in X0 Bits Encoding

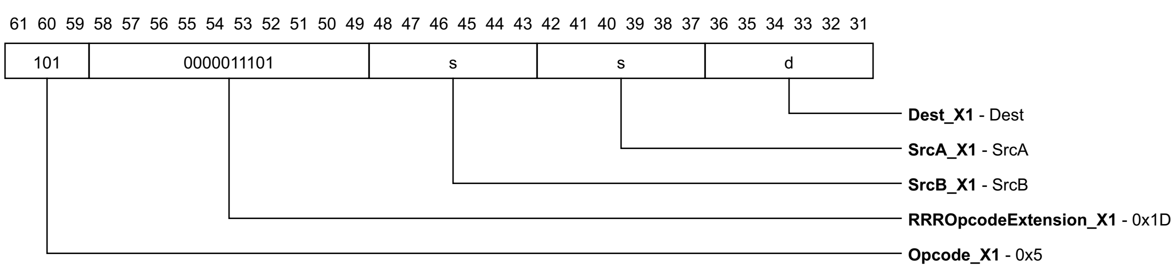

Figure 8-214: or in X1 Bits Encoding

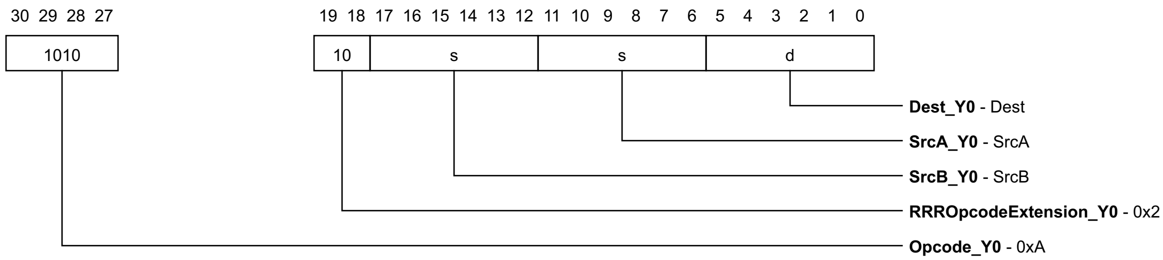

Figure 8-215: or in Y0 Bits Encoding

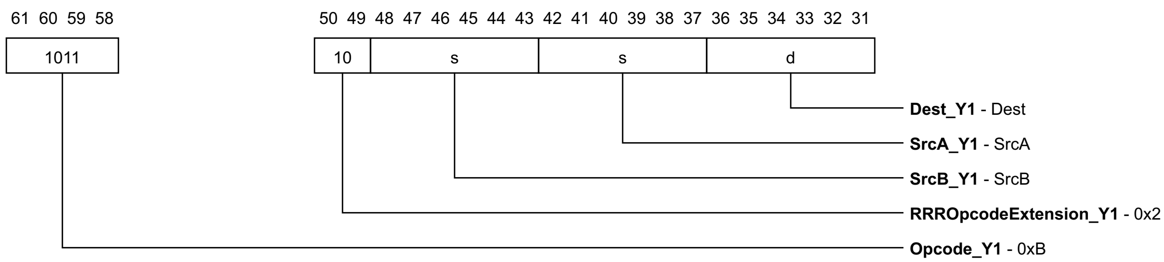

Figure 8-216: or in Y1 Bits Encoding

ori

Or Immediate

Syntax

```
ori Dest, SrcA, Imm8
```

Example

```
ori r5, r6, 5
```

Description

Compute the boolean OR of an operand and a sign extended immediate.

Functional Description

```
rf[Dest] = rf[SrcA] | signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

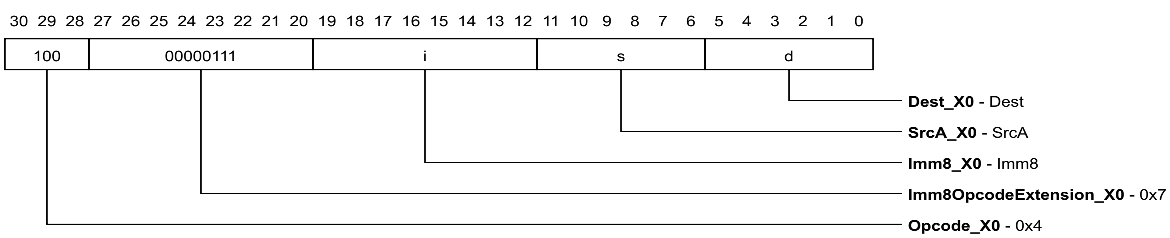

Figure 8-217: ori in X0 Bits Encoding

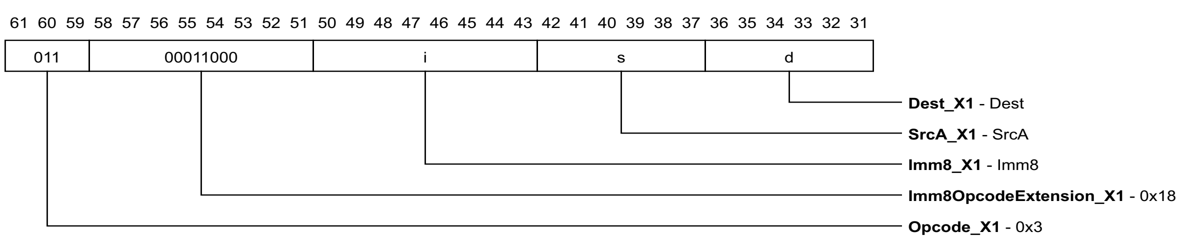

Figure 8-218: ori in X1 Bits Encoding

rotl

Rotate Left

Syntax

```
rotl Dest, SrcA, SrcB
```

Example

```
rotl r5, r6, r7
```

Description

Rotate the first source operand to the left by the second source operand. The effective rotate amount is the specified operand modulo the number of bits in a word.

Functional Description

```
rf[Dest] =
  (rf[SrcA] << (rf[SrcB] & 63)) | (((uint64_t) rf[SrcA]) >> (-rf[SrcB] & 63));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

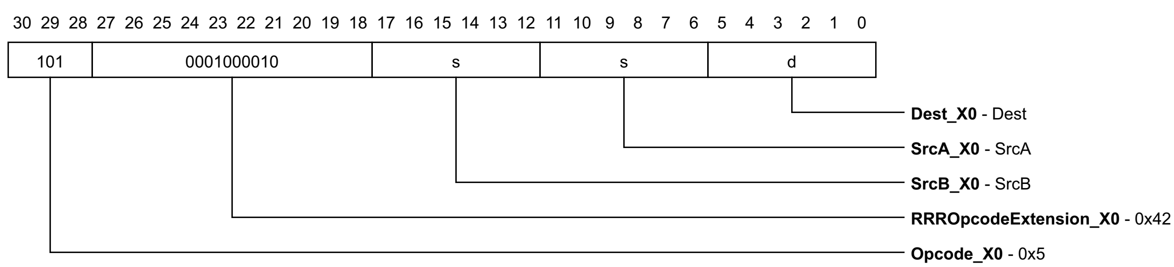

Figure 8-219: rotl in X0 Bits Encoding

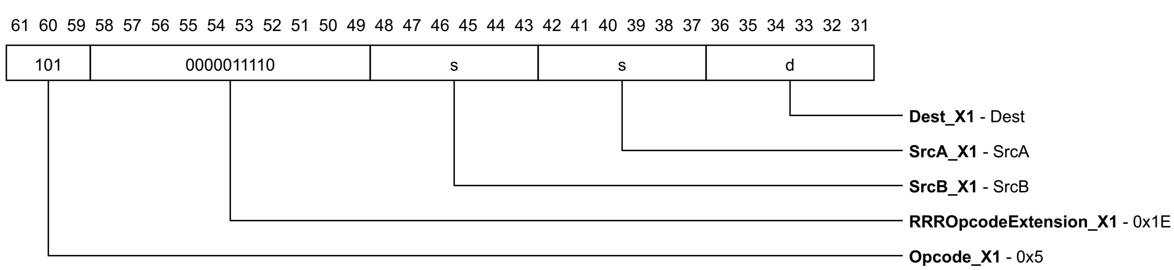

Figure 8-220: rotl in X1 Bits Encoding

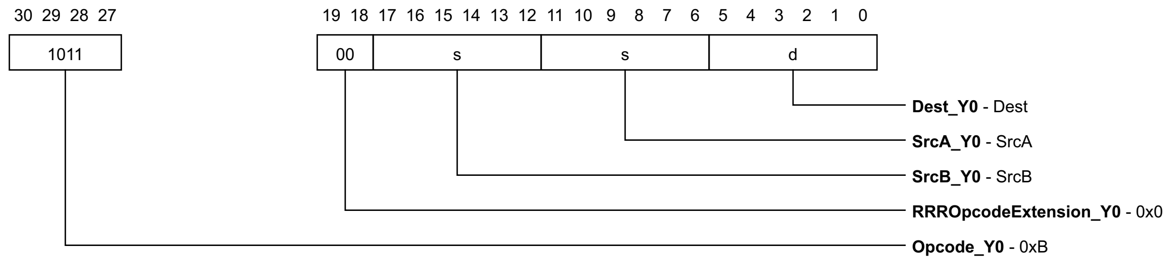

Figure 8-221: rotl in Y0 Bits Encoding

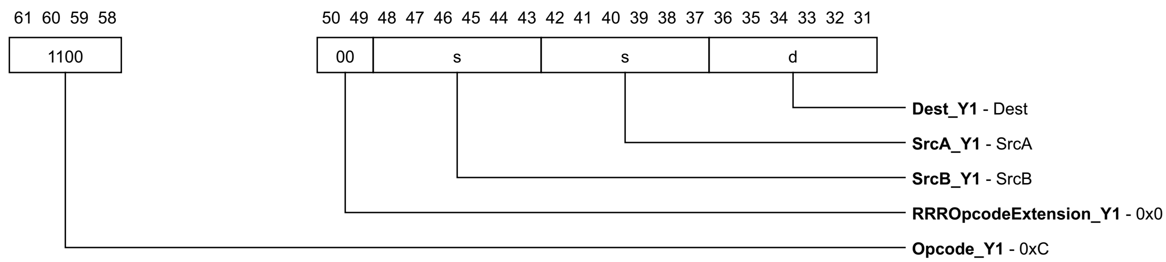

Figure 8-222: rotl in Y1 Bits Encoding

rotli

Rotate Left Immediate

Syntax

```
rotli Dest, SrcA, ShAmt
```

Example

```
rotli r5, r6, 5
```

Description

Rotate the first source operand to the left by an immediate. The effective rotate amount is the specified immediate modulo the number of bits in a word.

Functional Description

```
rf[Dest] = (rf[SrcA] << ShAmt) | (((uint64_t) rf[SrcA]) >> (-ShAmt & 63));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

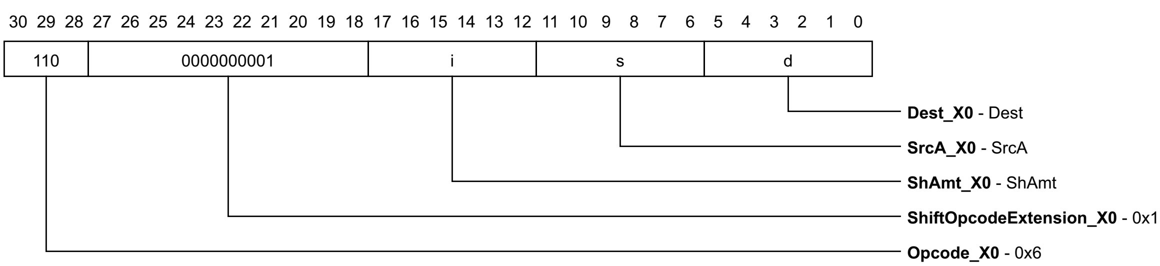

Figure 8-223: rotli in X0 Bits Encoding

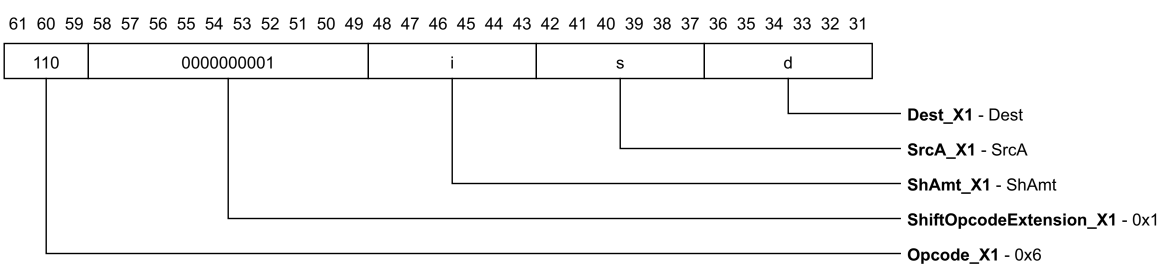

Figure 8-224: rotli in X1 Bits Encoding

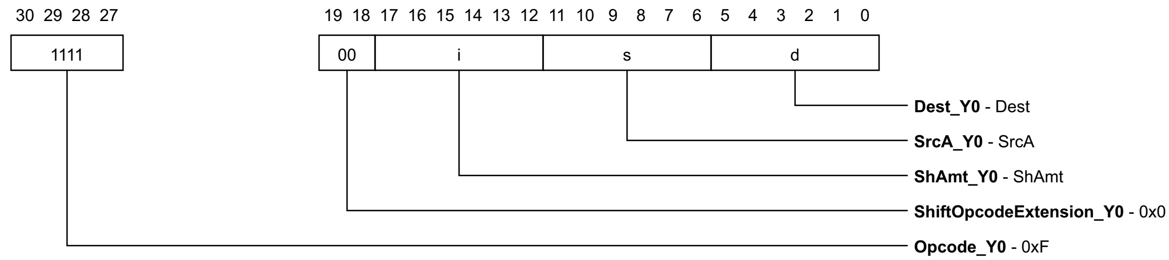

Figure 8-225: rotli in Y0 Bits Encoding

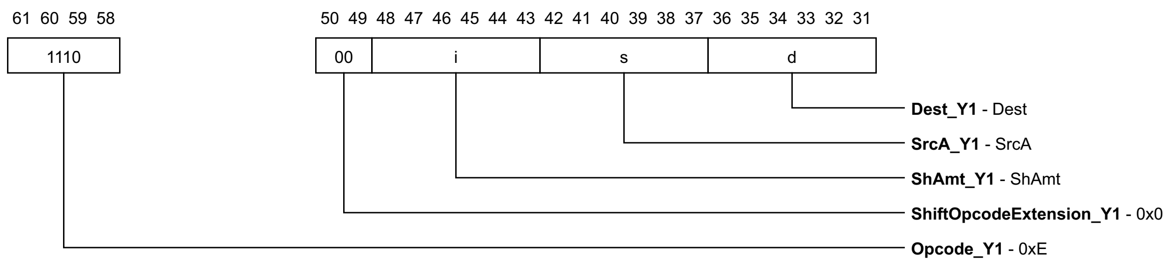

Figure 8-226: rotli in Y1 Bits Encoding

shl

Shift Left

Syntax

```
shl Dest, SrcA, SrcB
```

Example

```
shl r5, r6, r7
```

Description

Shift the first source operand to the left by the second source operand. The effective shift amount is the specified operand modulo the number of bits in a word. Left shifts shift zeros into the low ordered bits in a word and is suitable to be used as unsigned multiplication by powers of two.

Functional Description

```
rf[Dest] = rf[SrcA] << (rf[SrcB] & 63);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

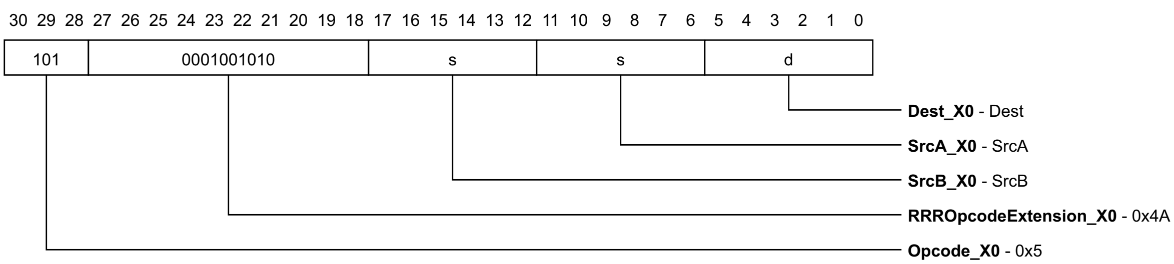

Figure 8-227: shl in X0 Bits Encoding

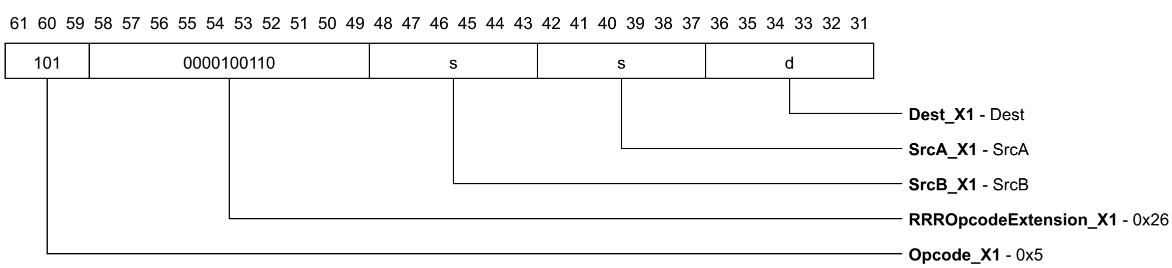

Figure 8-228: shl in X1 Bits Encoding

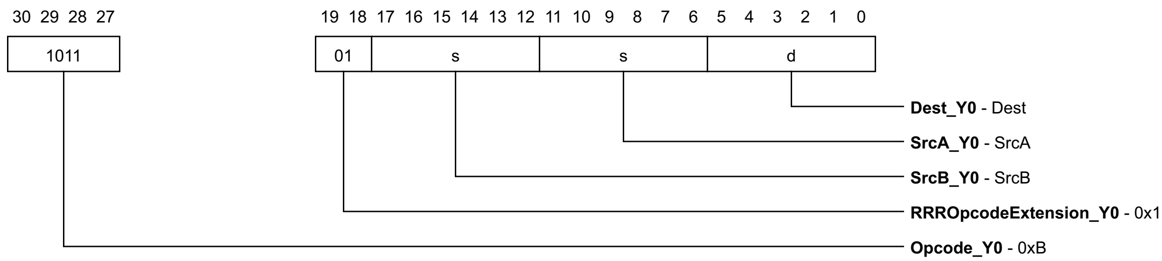

Figure 8-229: shl in Y0 Bits Encoding

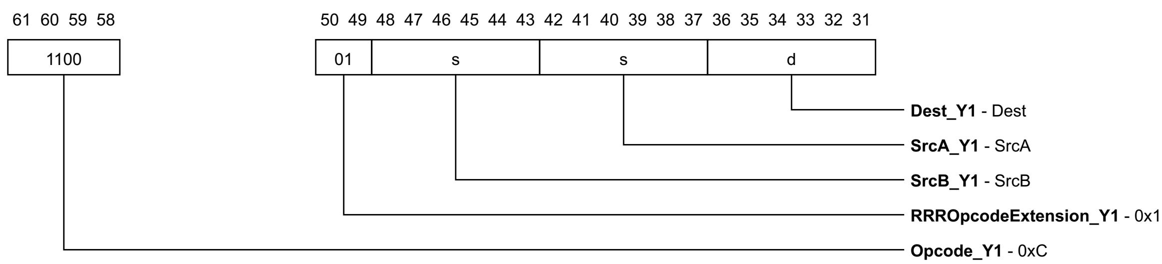

Figure 8-230: shl in Y1 Bits Encoding

shli

Shift Left Immediate

Syntax
```
shli Dest, SrcA, ShAmt
```

Example
```
shli r5, r6, 5
```

Description

Shift the first source operand to the left by an immediate. The effective shift amount is the specified immediate modulo the number of bits in a word. Left shifts shift zeros into the low ordered bits in a word and is suitable to be used as unsigned multiplication by powers of two.

Functional Description
```
rf[Dest] = rf[SrcA] << ShAmt;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

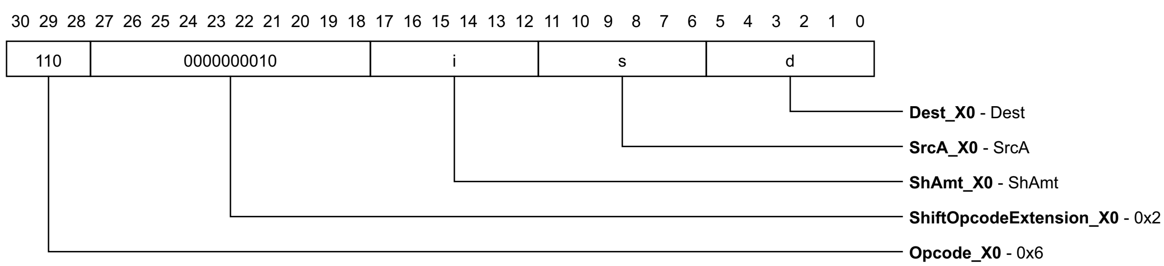

Figure 8-231: shli in X0 Bits Encoding

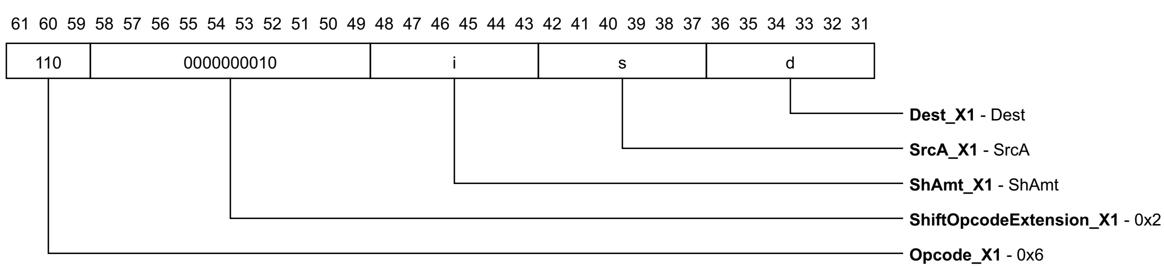

Figure 8-232: shli in X1 Bits Encoding

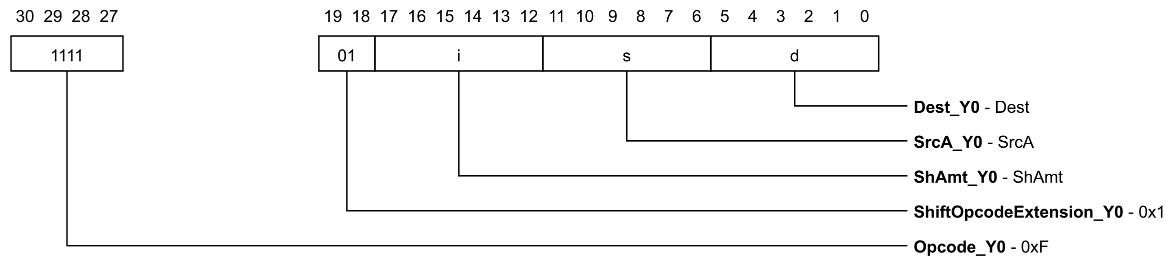

Figure 8-233: shli in Y0 Bits Encoding

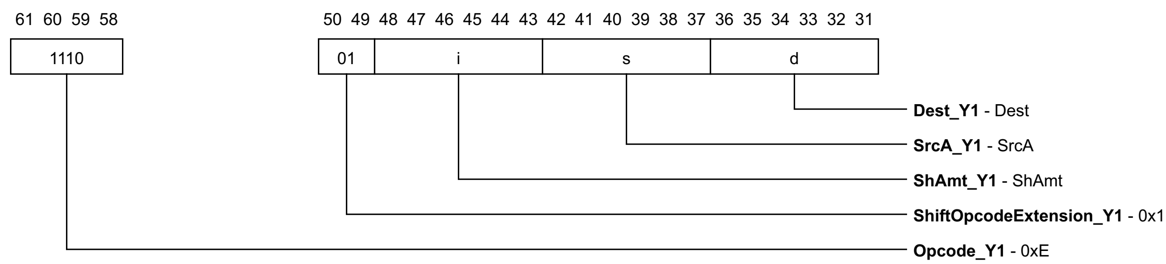

Figure 8-234: shli in Y1 Bits Encoding

shlx

Shift Left and Extend

Syntax

```
shlx Dest, SrcA, SrcB
```

Example

```
shlx r5, r6, r7
```

Description

Shift the bottom 4 bytes of the first source operand to the left by the second source operand and the 4-byte result is sign-extended. The effective shift amount is the specified operand modulo 32. This instruction shifts zeros into the low ordered bits in a word and is suitable to be used as unsigned multiplication by powers of two.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] << (rf[SrcB] & 31));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

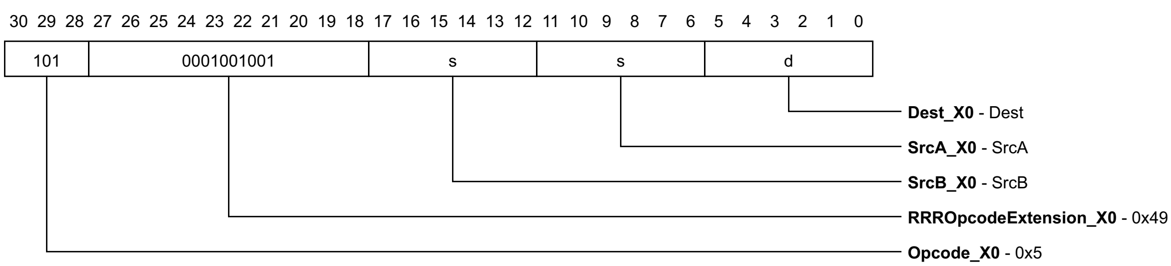

Figure 8-235: shlx in X0 Bits Encoding

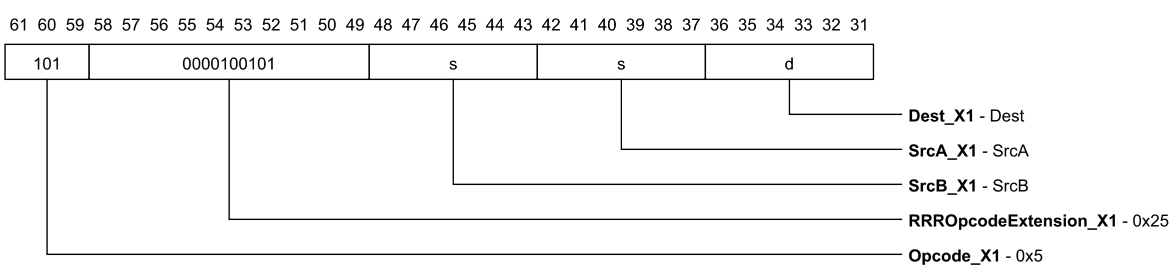

Figure 8-236: shlx in X1 Bits Encoding

shlxi

Shift Left and Extend Immediate

Syntax

```
shlxi Dest, SrcA, ShAmt
```

Example

```
shlxi r5, r6, 5
```

Description

Shift the bottom four bytes of the first source operand to the left by an immediate and the 4-byte result is sign-extended. If the shift amount is larger than 32, the effective shift amount is computed to be the specified shift amount modulo 32. This instruction shift zeros into the low ordered bits in a word and is suitable to be used as unsigned multiplication by powers of 2.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] << (ShAmt & 31));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

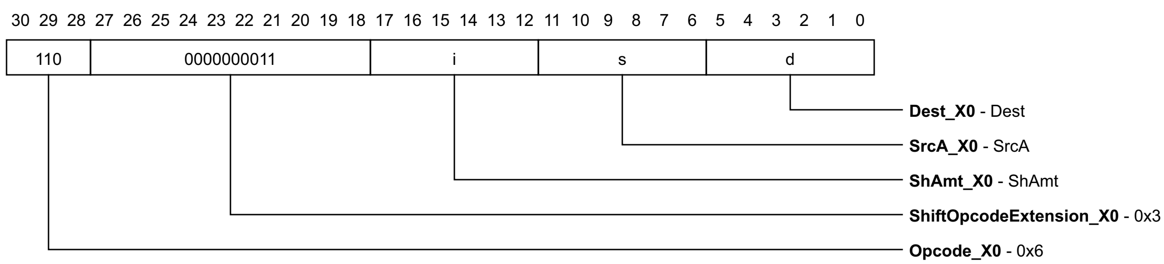

Figure 8-237: shlxi in X0 Bits Encoding

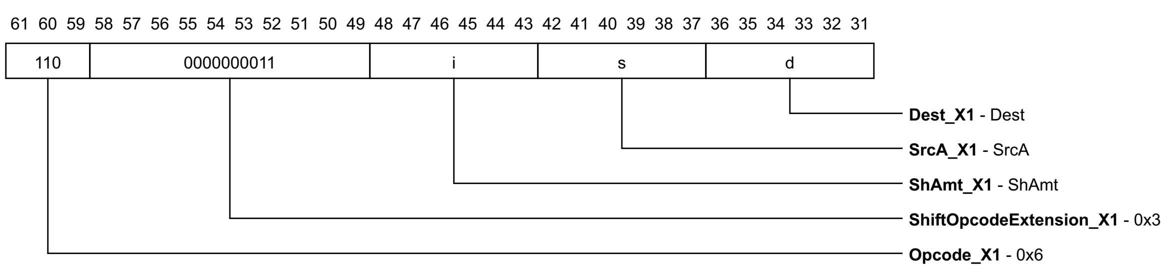

Figure 8-238: shlxi in X1 Bits Encoding

shrs

Shift Right Signed

Syntax

```
shrs Dest, SrcA, SrcB
```

Example

```
shrs r5, r6, r7
```

Description

Shift the first source operand to the right by the second source operand. The effective shift amount is the specified operand modulo the number of bits in a word. The first operand is treated as a signed value and the high-ordered bit is shifted into the high ordered bits in a word.

Functional Description

```
rf[Dest] = ((int64_t) rf[SrcA]) >> (rf[SrcB] & 63);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

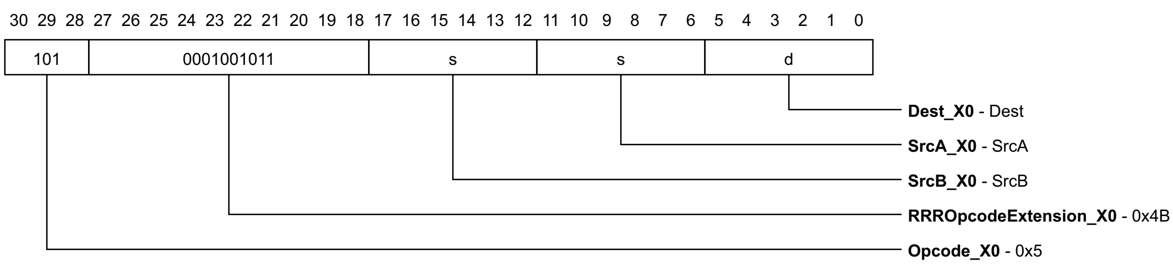

Figure 8-239: shrs in X0 Bits Encoding

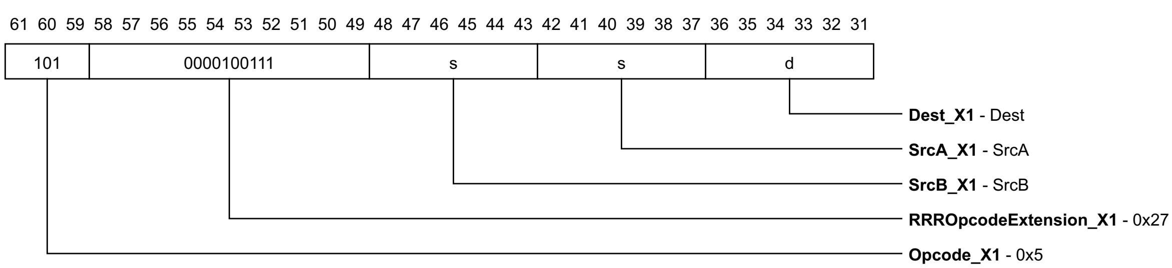

Figure 8-240: shrs in X1 Bits Encoding

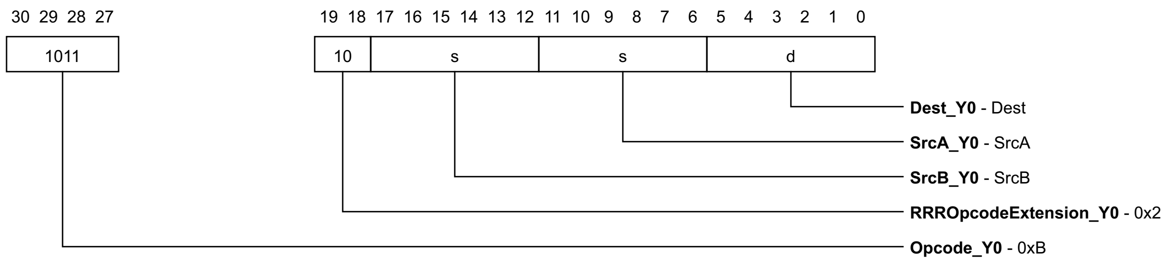

Figure 8-241: shrs in Y0 Bits Encoding

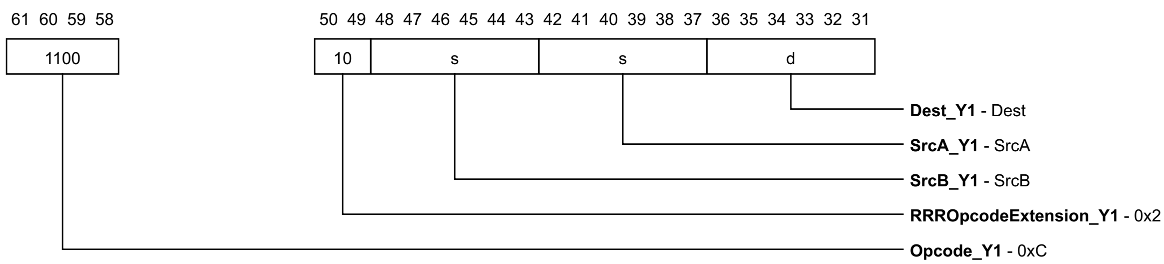

Figure 8-242: shrs in Y1 Bits Encoding

shrsi

Shift Right Signed Immediate

Syntax

```
shrsi Dest, SrcA, ShAmt
```

Example

```
shrsi r5, r6, 5
```

Description

Shift the first source operand to the right by an immediate. The effective shift amount is the specified immediate modulo the number of bits in a word. The first operand is treated as a signed value and the high-ordered bit is shifted into the high ordered bits in a word.

Functional Description

```
rf[Dest] = ((int64_t) rf[SrcA]) >> ShAmt;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

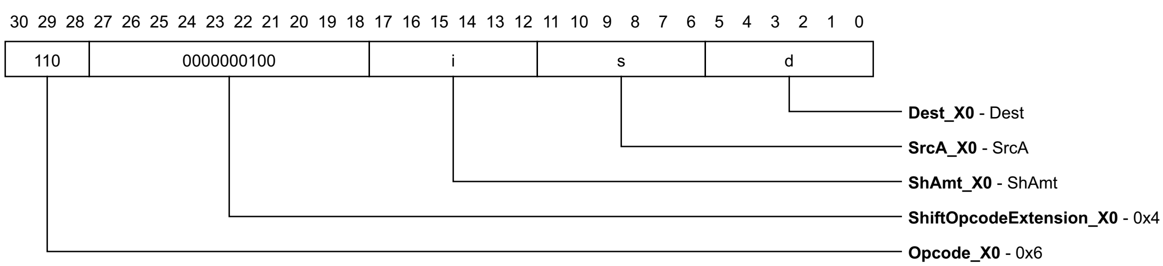

Figure 8-243: shrsi in X0 Bits Encoding

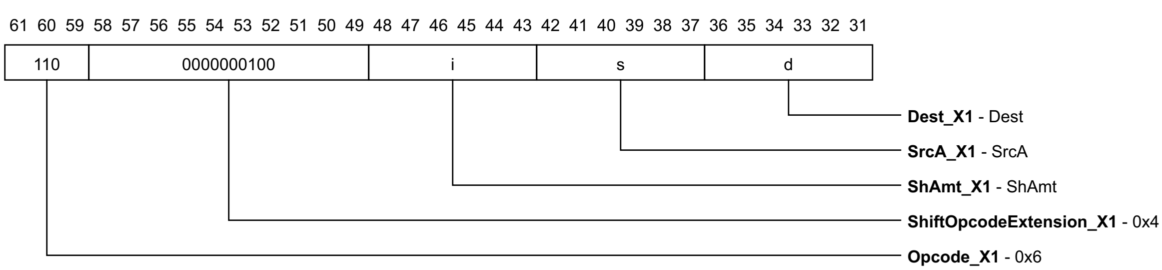

Figure 8-244: shrsi in X1 Bits Encoding

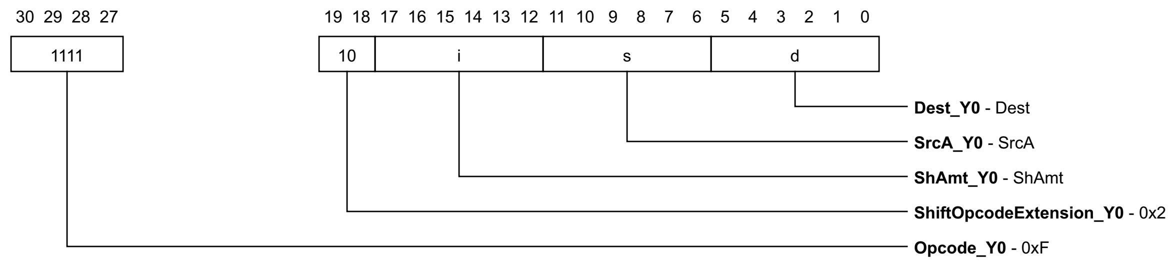

Figure 8-245: shrsi in Y0 Bits Encoding

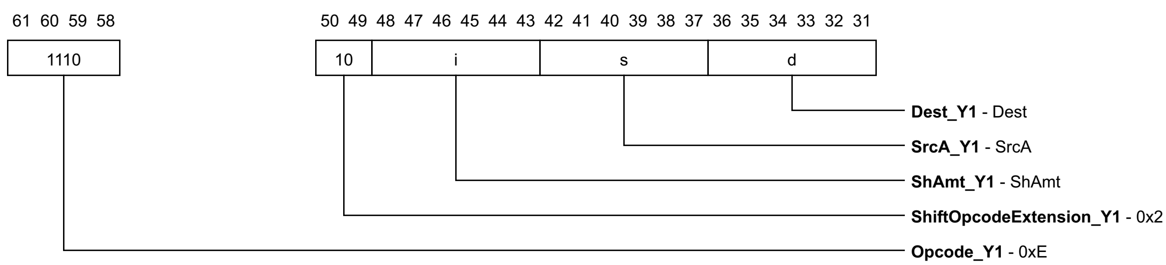

Figure 8-246: shrsi in Y1 Bits Encoding

shru

Shift Right Unsigned

Syntax

```
shru Dest, SrcA, SrcB
```

Example

```
shru r5, r6, r7
```

Description

Shift the first source operand to the right by the second source operand. The effective shift amount is the specified operand modulo the number of bits in a word. The first operand is treated as an unsigned quantity and shift zeros into the high ordered bits in a word. This instruction is suitable to be used as unsigned integer division by powers of two.

Functional Description

```
rf[Dest] = (uint64_t) rf[SrcA] >> (rf[SrcB] & 63);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

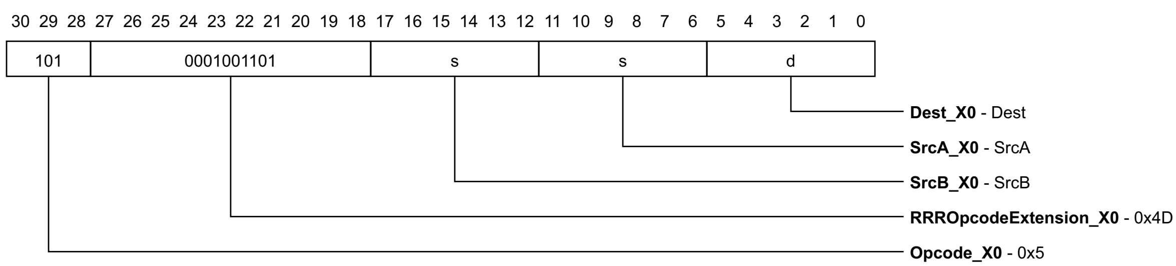

Figure 8-247: shru in X0 Bits Encoding

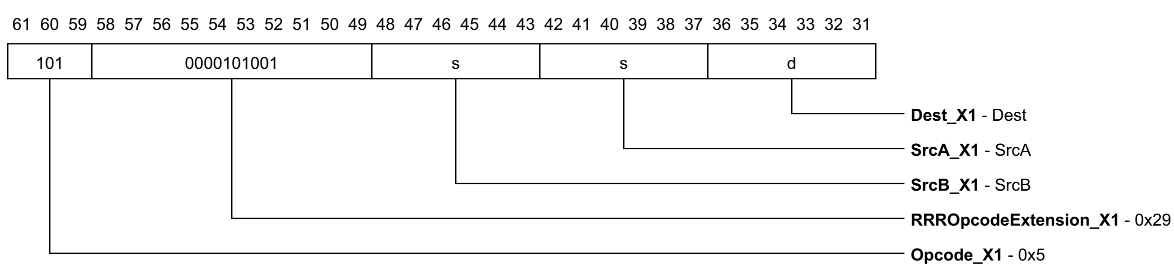

Figure 8-248: shru in X1 Bits Encoding

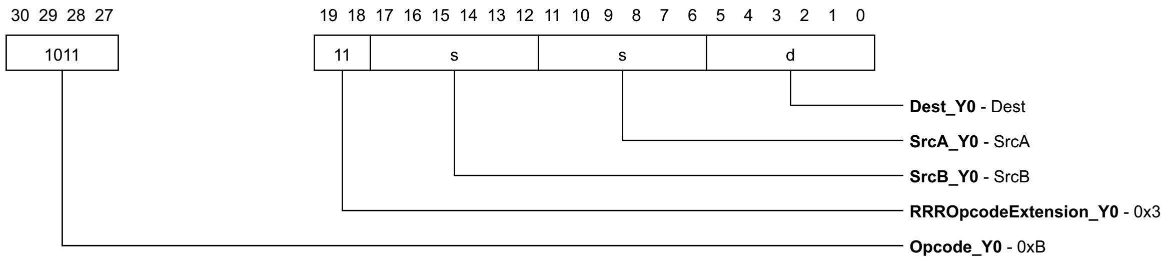

Figure 8-249: shru in Y0 Bits Encoding

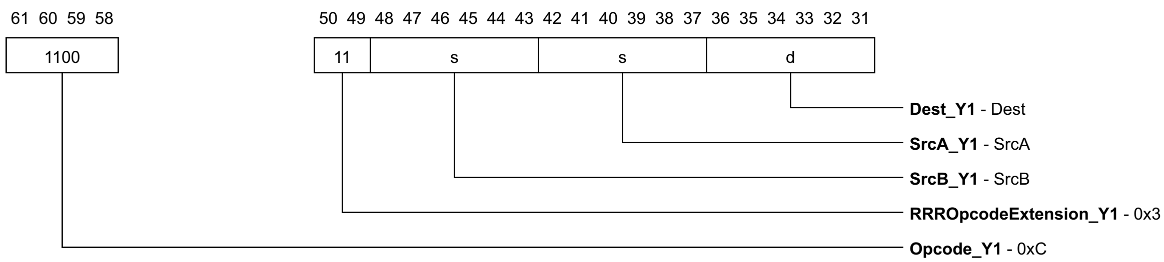

Figure 8-250: shru in Y1 Bits Encoding

shrui

Shift Right Unsigned Immediate

Syntax

```
shrui Dest, SrcA, ShAmt
```

Example

```
shrui r5, r6, 5
```

Description

Shift the first source operand to the right by an immediate. The effective shift amount is the specified immediate modulo the number of bits in a word. The first operand is treated as an unsigned quantity and zeros are shifted into the high ordered bits in a word. This instruction is suitable to be used as unsigned integer division by powers of two.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA]) >> ShAmt;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

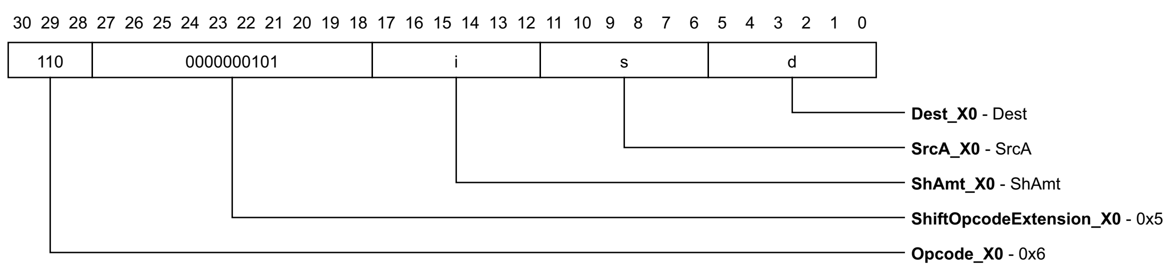

Figure 8-251: shrui in X0 Bits Encoding

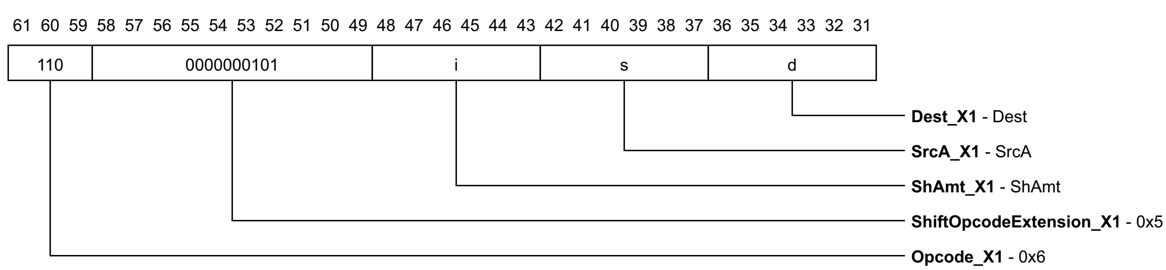

Figure 8-252: shrui in X1 Bits Encoding

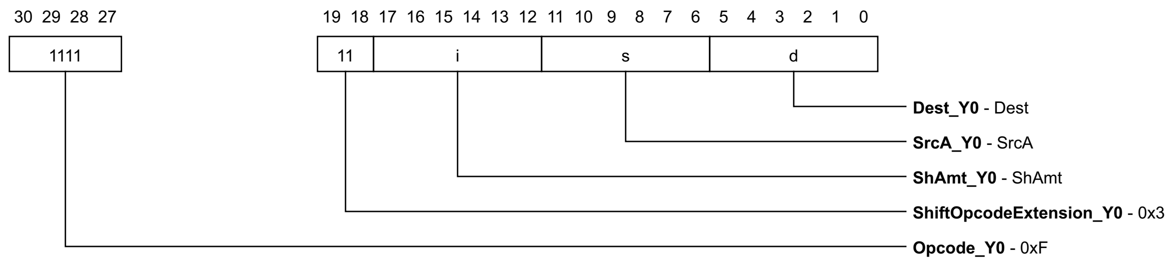

Figure 8-253: shrui in Y0 Bits Encoding

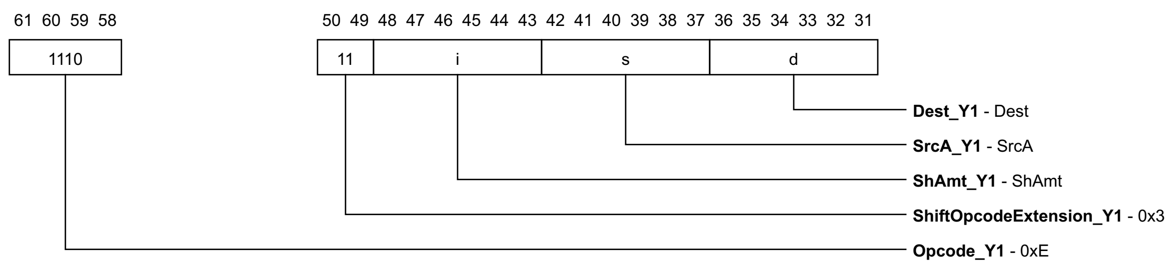

Figure 8-254: shrui in Y1 Bits Encoding

shrux

Shift Right Unsigned and Extend

Syntax

```
shrux Dest, SrcA, SrcB
```

Example

```
shrux r5, r6, r7
```

Description

Shift the bottom 4 bytes of the first source operand to the right by the second source operand and the 4-byte result is sign-extended. The effective shift amount is the specified operand modulo 32. The first operand is treated as an unsigned quantity and shift zeros into the high ordered bits. This instruction is suitable to be used as unsigned integer division by powers of two.

Functional Description

```
rf[Dest] = signExtend32 ((uint32_t) rf[SrcA] >> (rf[SrcB] & 31));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

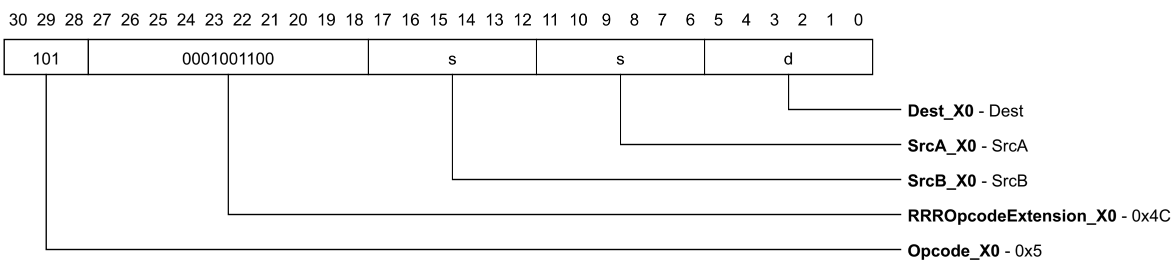

Figure 8-255: shrux in X0 Bits Encoding

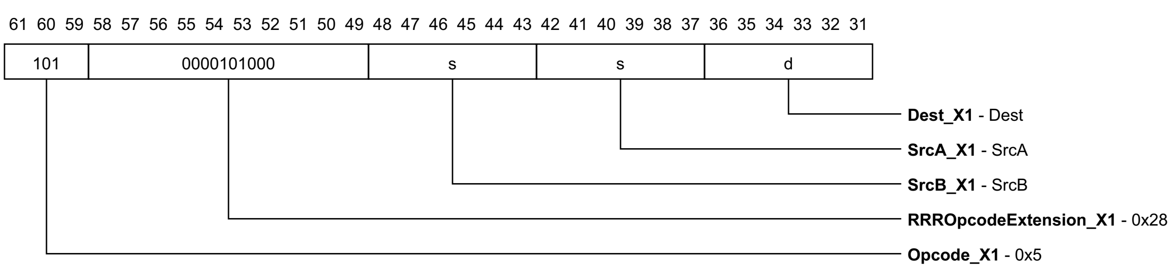

Figure 8-256: shrux in X1 Bits Encoding

shruxi

Shift Right Unsigned and Extend Immediate

Syntax

```
shruxi Dest, SrcA, ShAmt
```

Example

```
shruxi r5, r6, 5
```

Description

Shift the bottom 4 bytes of the first source operand to the right by an immediate and the 4-byte result is sign-extended. The effective shift amount is the specified immediate modulo 32. The first operand is treated as an unsigned quantity and zeros are shifted into the high ordered bits in a word. This instruction is suitable to be used as unsigned integer division by powers of two.

Functional Description

```
rf[Dest] = signExtend32 (((uint32_t) rf[SrcA]) >> (ShAmt & 31));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

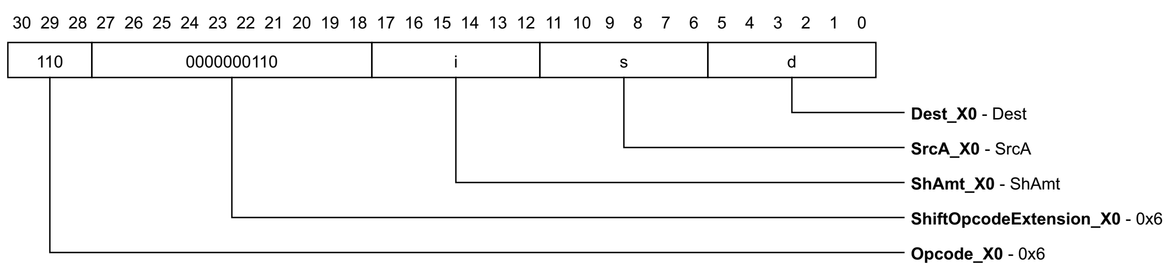

Figure 8-257: shruxi in X0 Bits Encoding

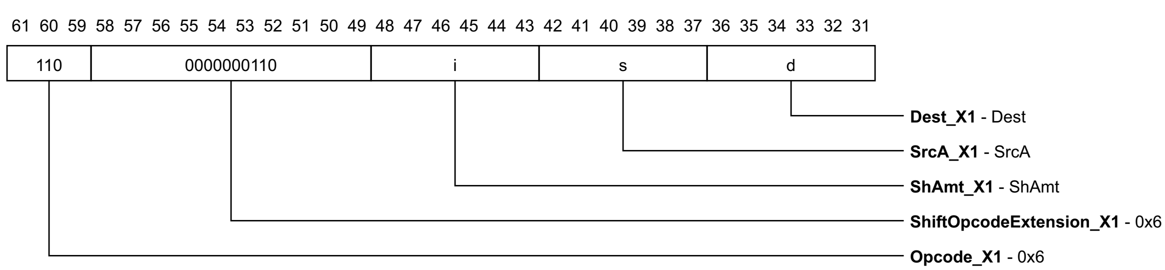

Figure 8-258: shruxi in X1 Bits Encoding

tblidxb0

Table Index Byte 0

Syntax

```
tblidxb0 Dest, SrcA
```

Example

```
tblidxb0 r5, r6
```

Description

Modify the table pointer stored in the destination operand to point to the word indexed by the contents of byte 0 of the source operand. The table is assumed to be aligned to a 1024 byte boundary, and bits 9:2 of the destination are replaced by the contents of bits 7:0 of the source operand.

Functional Description

```
rf[Dest] = (rf[Dest] & ~0x3FC) | (((rf[SrcA] >> 0) & BYTE_MASK) << 2);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

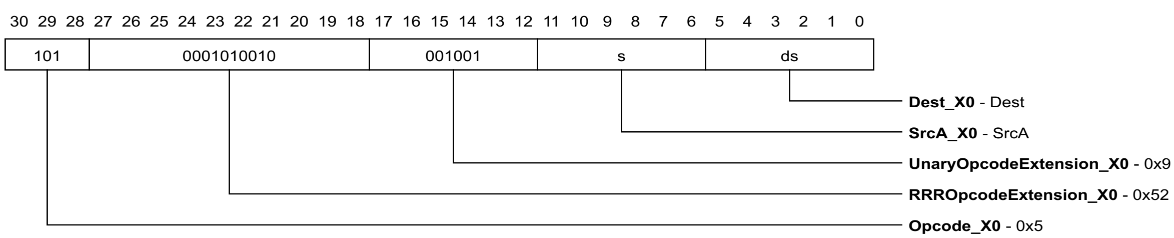

Figure 8-259: tblidxb0 in X0 Bits Encoding

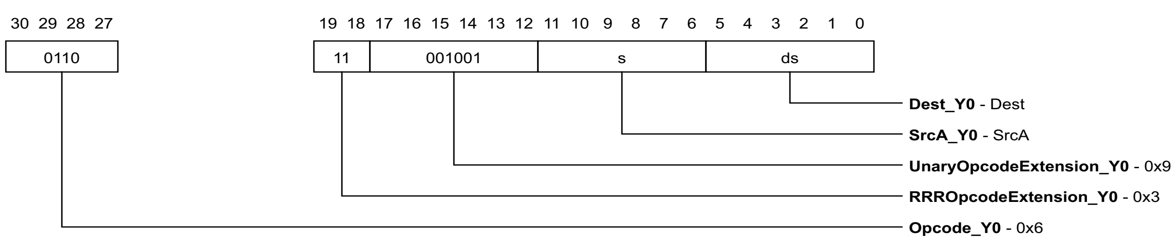

Figure 8-260: tblidxb0 in Y0 Bits Encoding

tblidxb1

Table Index Byte 1

Syntax

```
tblidxb1 Dest, SrcA
```

Example

```
tblidxb1 r5, r6
```

Description

Modify the table pointer stored in the destination operand to point to the word indexed by the contents of byte 1 of the source operand. The table is assumed to be aligned to a 1024 byte boundary, and bits 9:2 of the destination are replaced by the contents of bits 15:8 of the source operand.

Functional Description

```
rf[Dest] = (rf[Dest] & ~0x3FC) | (((rf[SrcA] >> 8) & BYTE_MASK) << 2);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

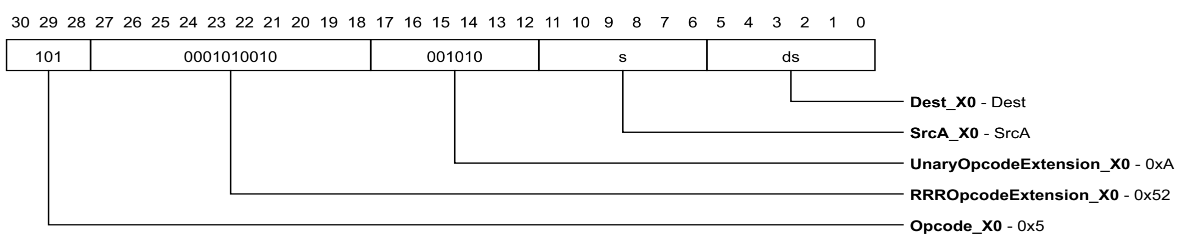

Figure 8-261: tblidxb1 in X0 Bits Encoding

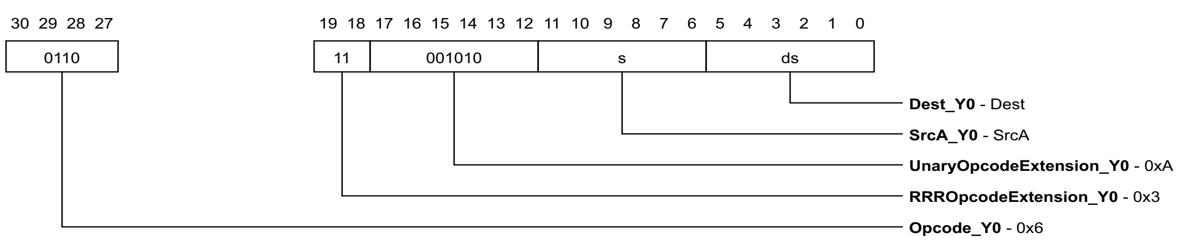

Figure 8-262: tblidxb1 in Y0 Bits Encoding

tblidxb2

Table Index Byte 2

Syntax

```
tblidxb2 Dest, SrcA
```

Example

```
tblidxb2 r5, r6
```

Description

Modify the table pointer stored in the destination operand to point to the word indexed by the contents of byte 2 of the source operand. The table is assumed to be aligned to a 1024 byte boundary, and bits 9:2 of the destination are replaced by the contents of bits 23:16 of the source operand.

Functional Description

```
rf[Dest] = (rf[Dest] & ~0x3FC) | (((rf[SrcA] >> 16) & BYTE_MASK) << 2);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

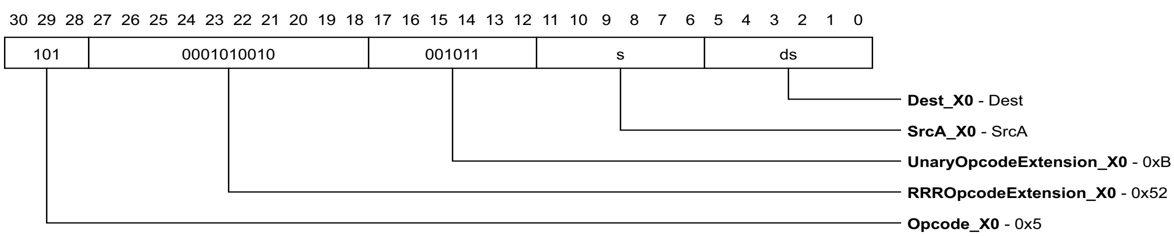

Figure 8-263: tblidxb2 in X0 Bits Encoding

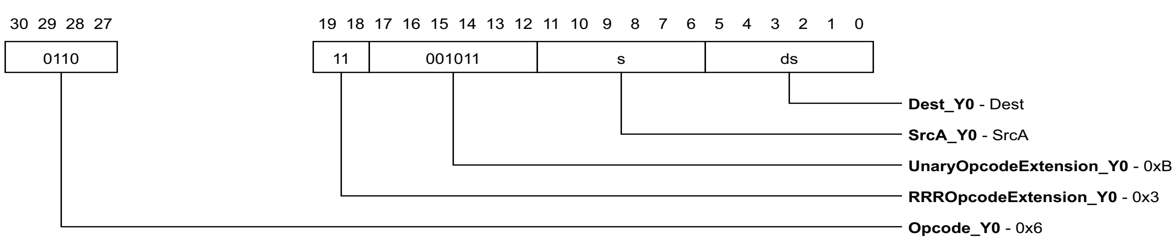

Figure 8-264: tblidxb2 in Y0 Bits Encoding

tblidxb3

Table Index Byte 3

Syntax

```
tblidxb3 Dest, SrcA
```

Example

```
tblidxb3 r5, r6
```

Description

Modify the table pointer stored in the destination operand to point to the word indexed by the contents of byte 3 of the source operand. The table is assumed to be aligned to a 1024 byte boundary, and bits 9:2 of the destination are replaced by the contents of bits 31:24 of the source operand.

Functional Description

```
rf[Dest] = (rf[Dest] & ~0x3FC) | (((rf[SrcA] >> 24) & BYTE_MASK) << 2);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

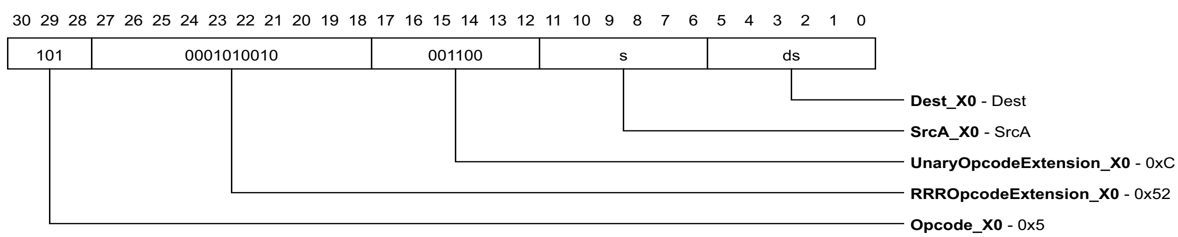

Figure 8-265: tblidxb3 in X0 Bits Encoding

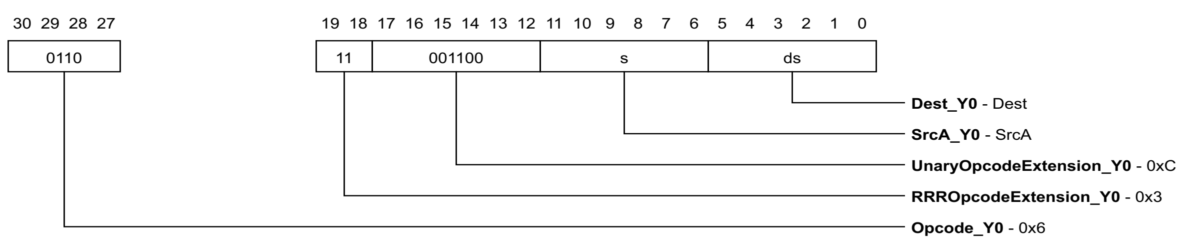

Figure 8-266: tblidxb3 in Y0 Bits Encoding

xor

Exclusive Or

Syntax
```
xor Dest, SrcA, SrcB
```

Example
```
xor r5, r6, r7
```

Description
Compute the boolean XOR of two operands.

Functional Description
```
rf[Dest] = rf[SrcA] ^ rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

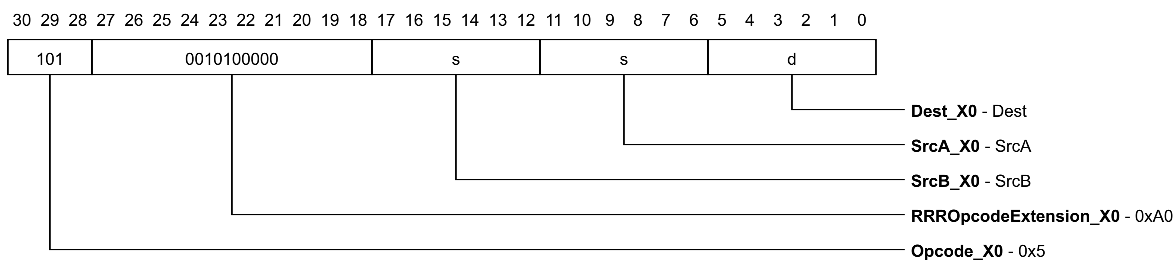

Figure 8-267: xor in X0 Bits Encoding

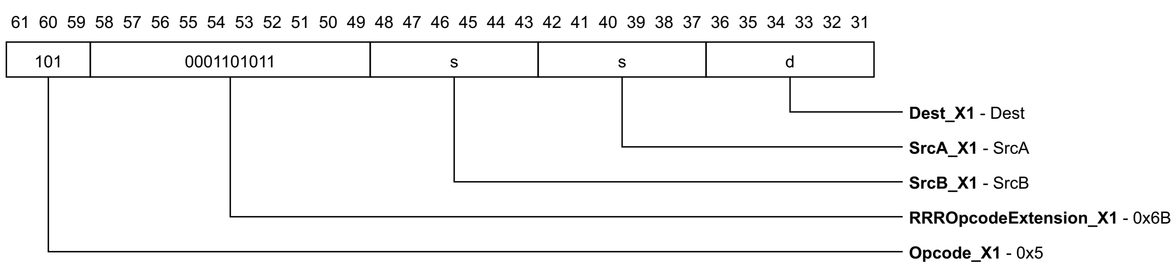

Figure 8-268: xor in X1 Bits Encoding

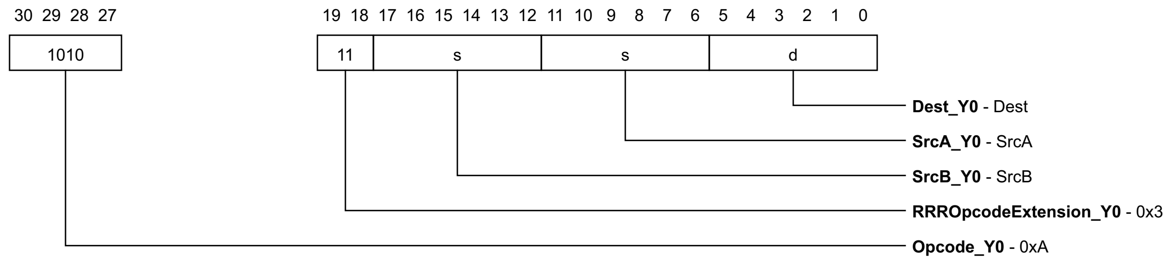

Figure 8-269: xor in Y0 Bits Encoding

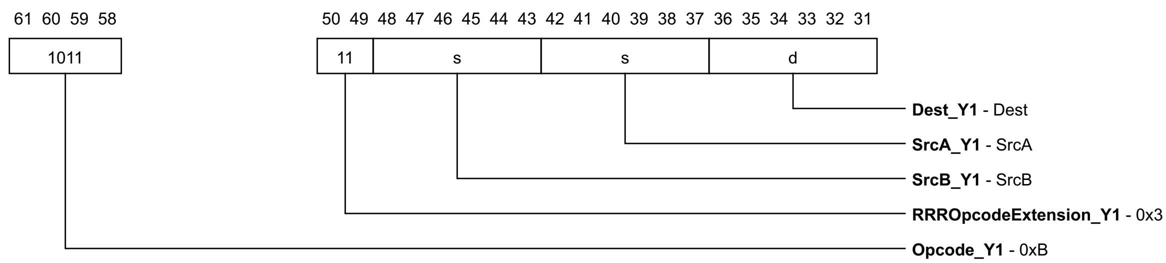

Figure 8-270: xor in Y1 Bits Encoding

xori

Exclusive Or Immediate

Syntax

```
xori Dest, SrcA, Imm8
```

Example

```
xori r5, r6, 5
```

Description

Compute the boolean XOR of an operand and a sign extended immediate.

Functional Description

```
rf[Dest] = rf[SrcA] ^ signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

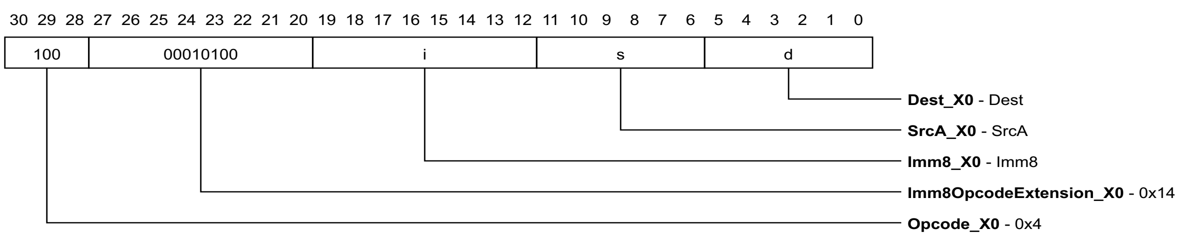

Figure 8-271: xori in X0 Bits Encoding

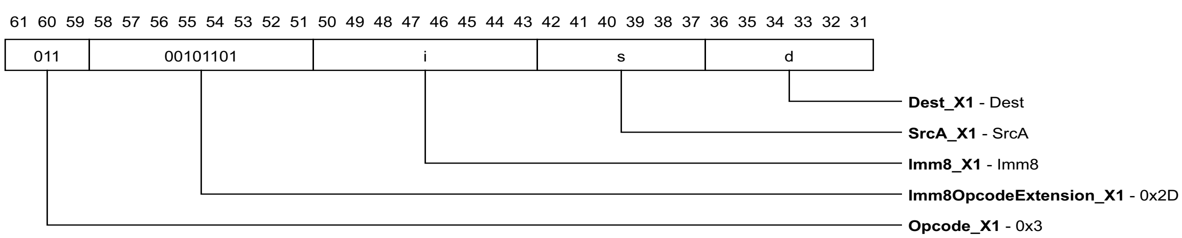

Figure 8-272: xori in X1 Bits Encoding

CHAPTER 9 MEMORY MAINTENANCE INSTRUCTIONS

9.1 Overview

The following sections provide detailed descriptions of memory maintenance instructions listed alphabetically.

9.2 Instructions

Memory maintenance instructions are described in the sections that follow.

dtlbpr

Data TLB Probe

Syntax

```
dtlbpr SrcA
```

Example

```
dtlbpr r5
```

Description

Probe the Data TLB and return the results as a unary encoded result for each matching entry into the `DTLB_MATCH_X` SPR. This probe uses the data CPL and ignores the `D_ASID`.

Functional Description

```
dtlbProbe (rf[SrcA]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

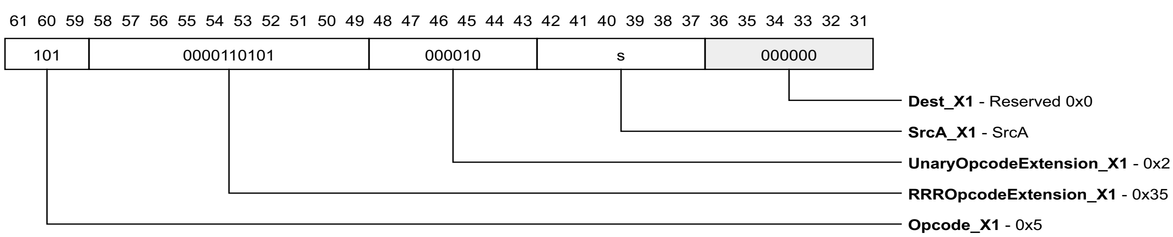

Figure 9-273: dtlbpr in X1 Bits Encoding

finv

Flush and Invalidate Cache Line

Syntax

```
finv SrcA
```

Example

```
finv r5
```

Description

Flush and Invalidates the cache line in the data cache that contains the address stored in the source operand. If a cache line that contains the address is not in the cache, this instruction has no effect. The line size that is flushed and invalidated is at minimum 16B. An implementation is free to flush and invalidate a larger region.

Functional Description

```
flushAndInvalidateCacheLine (rf[SrcA]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

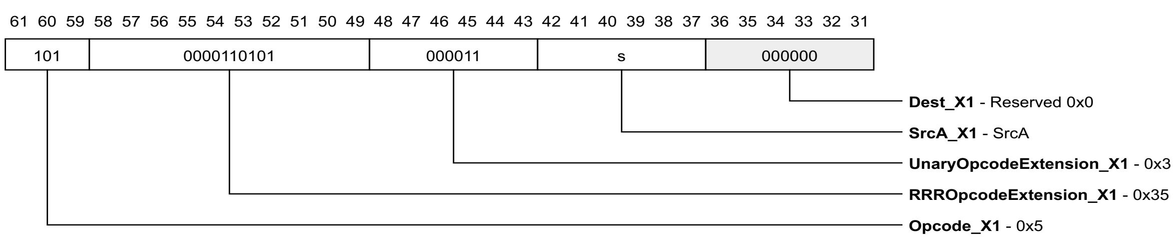

Figure 9-274: finv in X1 Bits Encoding

flush

Flush Cache Line

Syntax

```
flush SrcA
```

Example

```
flush r5
```

Description

Flushes the cache line in the data cache that contains the address stored in the source operand. If a cache line that contains the address is not in the cache, this instruction has no effect. If a cache line that contains the address is not dirty in the cache, this instruction has no effect. The line size that is flushed is at minimum 16B. An implementation is free to flush a larger region.

Functional Description

```
flushCacheLine (rf[SrcA]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

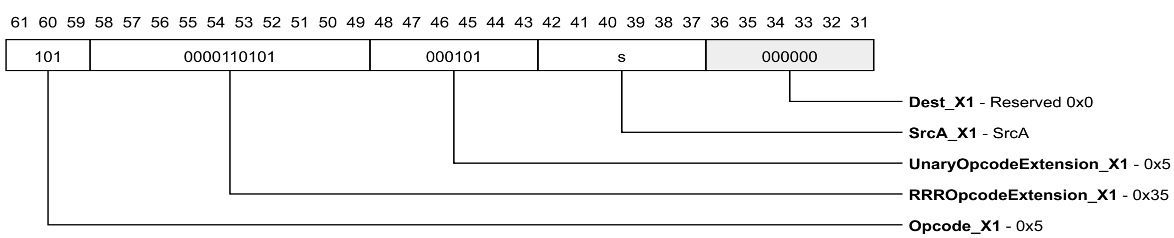

Figure 9-275: flush in X1 Bits Encoding

flushwb

Flush Write Buffers

Syntax

```
flushwb
```

Example

```
flushwb
```

Description

Flush all write buffers internal to the processor. This instruction is a hint to make all writes performed by this processor visible to all other processors in the system as soon as possible.

Functional Description

```
flushWriteBuffers ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

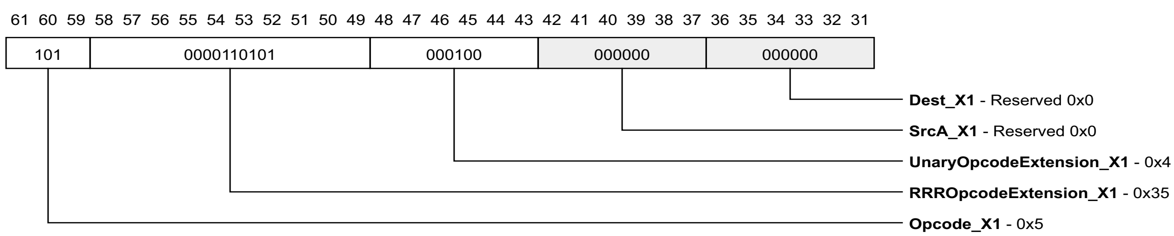

Figure 9-276: flushwb in X1 Bits Encoding

inv

Invalidate Cache Line

Syntax
```
inv SrcA
```

Example
```
inv r5
```

Description

Invalidates the cache line in the data cache that contains the address stored in the source operand. If a cache line that contains the address is not in the cache, this instruction has no effect. This instruction causes an access violation if the current privilege level is not allowed to write to the specified cache line. The line size that is invalidated is at minimum 16B. An implementation is free to invalidate a larger region.

Functional Description
```
invalidateCacheLine (rf[SrcA] & BYTE_16_ADDR_MASK);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

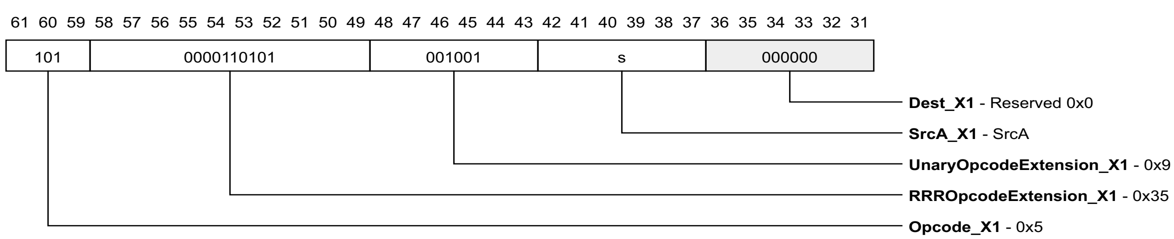

Figure 9-277: inv in X1 Bits Encoding

mf

Memory Fence

Syntax

```
mf
```

Example

```
mf
```

Description

The memory fence instruction is used to establish ordering between prior memory operations and subsequent instructions. The exact orderings that are established depend on the page attributes of the pages that the memory operations are targetting.

Functional Description

```
memoryFence ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

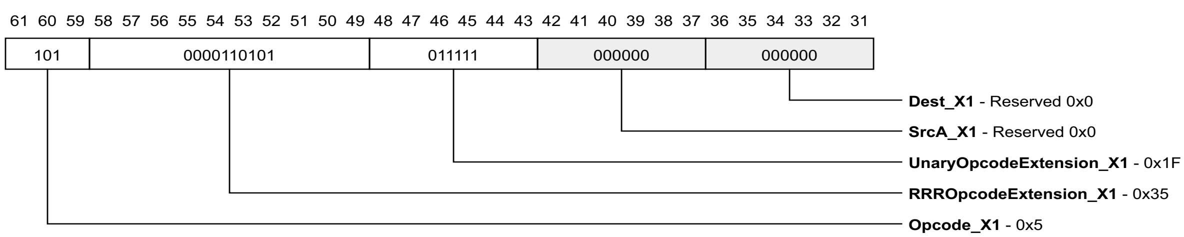

Figure 9-278: mf in X1 Bits Encoding

wh64

Write Hint 64 Bytes

Syntax

```
wh64 SrcA
```

Example

```
wh64 r5
```

Description

Hint that software intends to write every byte of the specified 64B cache line before reading it. The processor may use this hint to allocate the 64B line into the cache without fetching the current contents from main memory. The processor may set the contents of the block to any value that does not introduce a security hole.

Functional Description

```
writeHint64Cache (rf[SrcA] & BYTE_64_ADDR_MASK);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

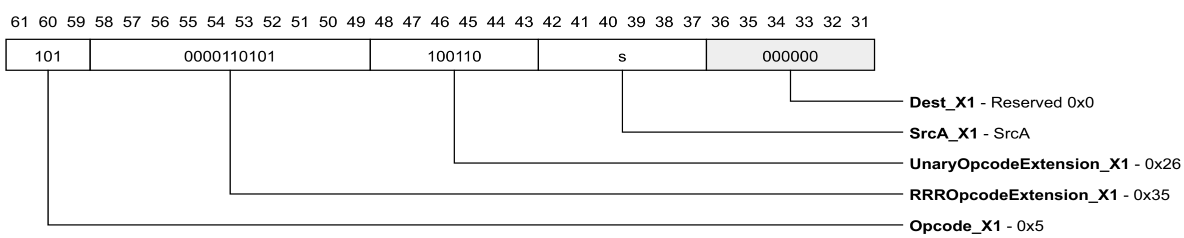

Figure 9-279: wh64 in X1 Bits Encoding

CHAPTER 10 MEMORY INSTRUCTIONS

10.1 Overview

The following sections provide detailed descriptions of memory instructions listed alphabetically.

ldna	Load No Alignment Trap
ldna_add	Load No Alignment Trap and Add
ldnt	Load Non-Temporal
ldnt1s	Load Non-Temporal One Byte Signed
ldnt1s_add	Load Non-Temporal One Byte Signed and Add
ldnt1u	Load Non-Temporal One Byte Unsigned
ldnt1u_add	Load Non-Temporal One Byte Unsigned and Add
ldnt2s	Load Non-Temporal Two Bytes Signed
ldnt2s_add	Load Non-Temporal Two Bytes Signed and Add
ldnt2u	Load Non-Temporal Two Bytes Unsigned
ldnt2u_add	Load Non-Temporal Two Bytes Unsigned and Add
ldnt4s	Load Non-Temporal Four Bytes Signed
ldnt4s_add	Load Non-Temporal Four Bytes Signed and Add
ldnt4u	Load Non-Temporal Four Bytes Unsigned
ldnt4u_add	Load Non-Temporal Four Bytes Unsigned and Add
ldnt_add	Load Non-Temporal and Add
st	Store
st1	Store Byte
st1_add	Store Byte and Add
st2	Store Two Bytes
st2_add	Store Two Bytes and Add
st4	Store Four Bytes
st4_add	Store Four Bytes and Add
st_add	Store and Add
stnt	Store Non-Temporal
stnt1	Store Non-Temporal Byte
stnt1_add	Store Non-Temporal Byte and Add
stnt2	Store Non-Temporal Two Bytes
stnt2_add	Store Non-Temporal Two Bytes and Add
stnt4	Store Non-Temporal Four Bytes
stnt4_add	Store Non-Temporal Four Bytes and Add
stnt_add	Store Non-Temporal and Add

10.2Instructions

Memory instructions are described in the sections that follow.

cmpexch

Compare and Exchange

Syntax

```
cmpexch Dest, SrcA, SrcB
```

Example

```
cmpexch r5, r6, r7
```

Description

Compare the 8-byte contents of the `CmpValue` SPR with the 8-byte value in memory at the address held in the first source register. If the values are not equal, then no memory operation is performed. If the values are equal, the 8-byte quantity from the second source register is written into memory at the address held in the first source register. In either case, the result of the instruction is the value read from memory. The compare and write to memory are atomic and thus can be used for synchronization purposes. This instruction only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = memoryReadDoubleWord (rf[SrcA]);
rf[Dest] = memVal;
if (memVal == SPR[CmpValueSPR])
  memoryWriteDoubleWord (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

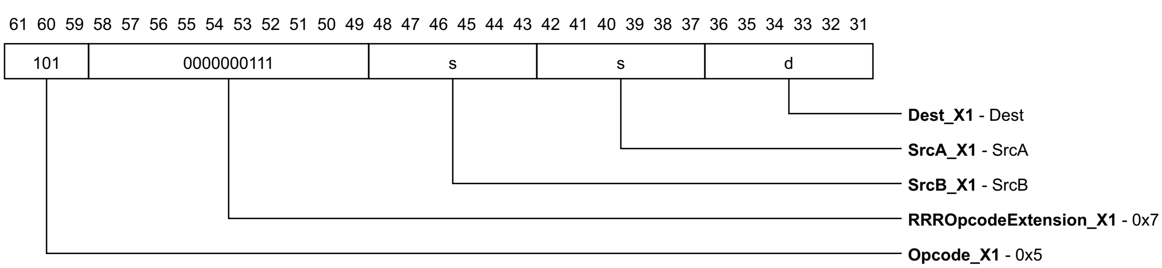

Figure 10-280: cmpexch in X1 Bits Encoding

cmpexch4

Compare and Exchange Four Bytes

Syntax

```
cmpexch4 Dest, SrcA, SrcB
```

Example

```
cmpexch4 r5, r6, r7
```

Description

Compare the 4-byte contents of the `CmpValue` SPR with the 4-byte value in memory at the address held in the first source register. If the values are not equal, then no memory operation is performed. If the values are equal, the 4-byte quantity from the second source register is written into memory at the address held in the first source register. In either case, the result of the instruction is the value read from memory. The compare and write to memory are atomic and thus can be used for synchronization purposes. This instruction only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = signExtend32 (memoryReadWord (rf[SrcA]));
rf[Dest] = memVal;
if (memVal == signExtend32 (SPR[CmpValueSPR]))
  memoryWriteWord (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

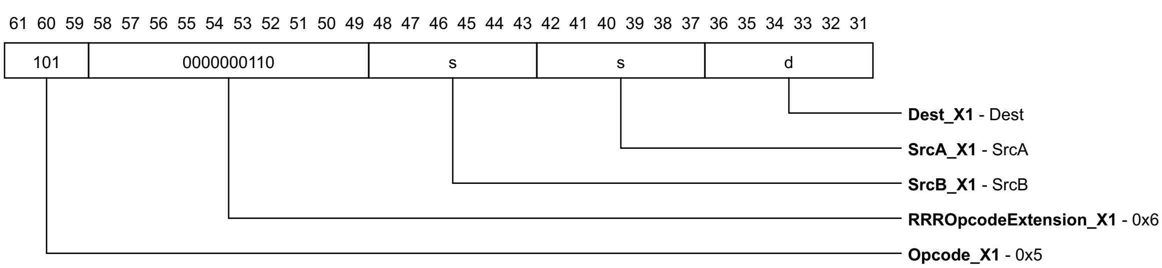

Figure 10-281: cmpexch4 in X1 Bits Encoding

exch

Exchange

Syntax

```
exch Dest, SrcA, SrcB
```

Example

```
exch r5, r6, r7
```

Description

Exchange the 8-byte quantity from the second source register with the value in memory at the address held in the first source register. The exchange in memory is atomic and thus can be used for synchronization purposes. This instruction only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
rf[Dest] = memoryReadDoubleWord (rf[SrcA]);
memoryWriteDoubleWord (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

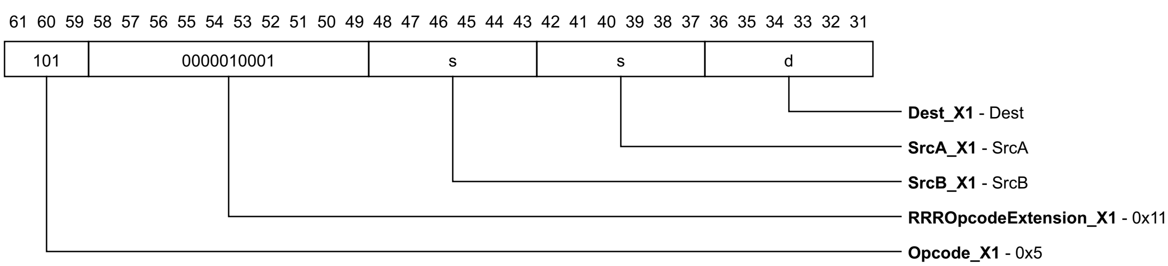

Figure 10-282: exch in X1 Bits Encoding

exch4

Exchange Four Bytes

Syntax

```
exch4 Dest, SrcA, SrcB
```

Example

```
exch4 r5, r6, r7
```

Description

Exchange the 4-byte quantity from the second source register with the value in memory at the address held in the first source register. The exchange in memory is atomic and thus can be used for synchronization purposes. This instruction only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
rf[Dest] = memoryReadWord (rf[SrcA]);
memoryWriteWord (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

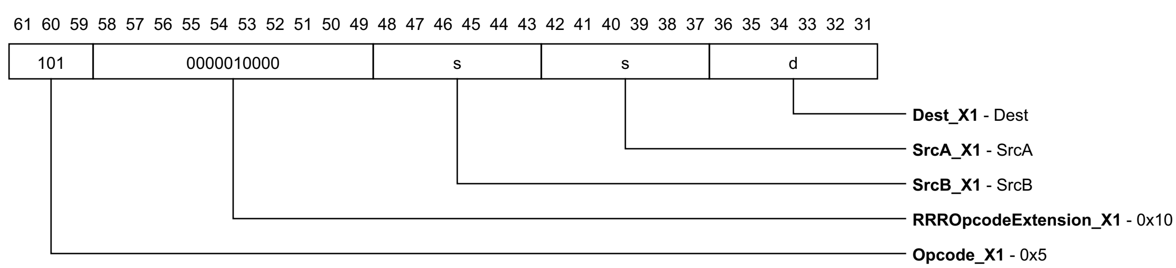

Figure 10-283: exch4 in X1 Bits Encoding

fetchadd

Fetch and Add

Syntax

```
fetchadd Dest, SrcA, SrcB
```

Example

```
fetchadd r5, r6, r7
```

Description

Read and return the 8-byte value in memory at the address held in the first source register. The value in memory is incremented by the 8-byte quantity from the second source register. The read and increment are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = memoryReadDoubleWord (rf[SrcA]);
rf[Dest] = memVal;
memoryWriteDoubleWord (rf[SrcA], memVal + rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

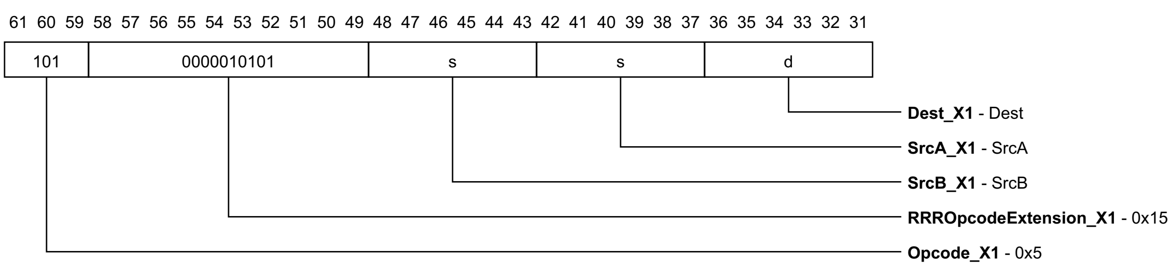

Figure 10-284: fetchadd in X1 Bits Encoding

fetchadd4

Fetch and Add Four Bytes

Syntax

```
fetchadd4 Dest, SrcA, SrcB
```

Example

```
fetchadd4 r5, r6, r7
```

Description

Read and return the 4-byte value in memory at the address held in the first source register. The value in memory is incremented by the 4-byte quantity from the second source register. The read and increment are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = signExtend32 (memoryReadWord (rf[SrcA]));
rf[Dest] = memVal;
memoryWriteWord (rf[SrcA], memVal + rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

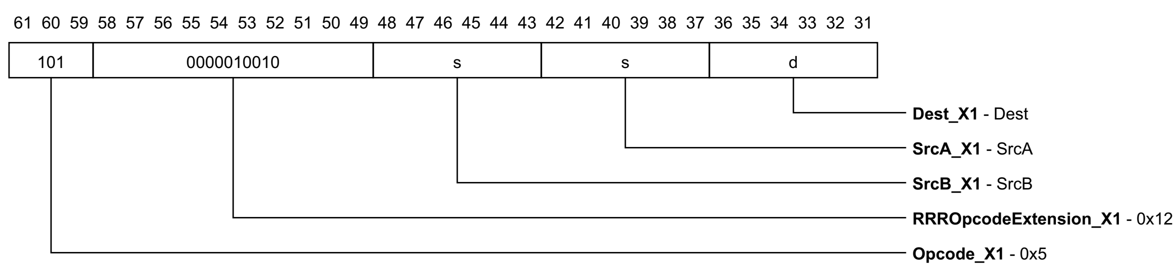

Figure 10-285: fetchadd4 in X1 Bits Encoding

fetchaddgez

Fetch and Add if Greater or Equal Zero

Syntax

```
fetchaddgez Dest, SrcA, SrcB
```

Example

```
fetchaddgez r5, r6, r7
```

Description

Read and return the 8-byte value in memory at the address held in the first source register. The value in memory is incremented by the 8-byte quantity from the second source register if the result would be greater or equal to zero. The read and increment are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
int64_t memVal = memoryReadDoubleWord (rf[SrcA]);
int64_t inc_result = memVal + rf[SrcB];
rf[Dest] = memVal;
if (inc_result >= 0)
  memoryWriteDoubleWord (rf[SrcA], inc_result);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

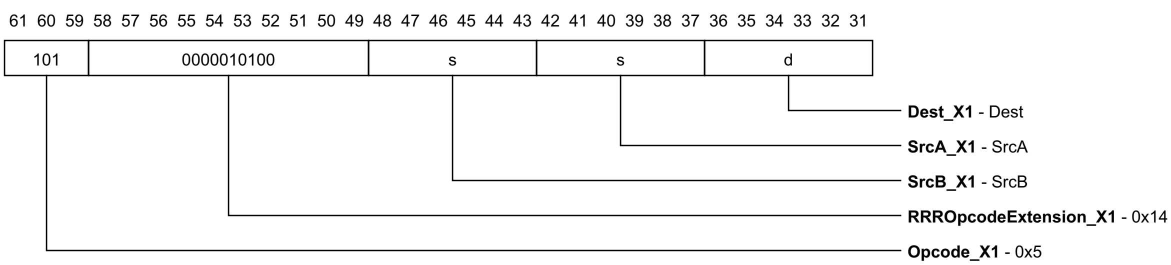

Figure 10-286: fetchaddgez in X1 Bits Encoding

fetchaddgez4

Fetch and Add if Greater or Equal Zero Four Bytes

Syntax

```
fetchaddgez4 Dest, SrcA, SrcB
```

Example

```
fetchaddgez4 r5, r6, r7
```

Description

Read and return the 4-byte value in memory at the address held in the first source register. The value in memory is incremented by the 4-byte quantity from the second source register if the result would be greater or equal to zero. The read and increment are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
int64_t memVal = signExtend32 (memoryReadWord (rf[SrcA]));
int32_t inc_result = memVal + rf[SrcB];
rf[Dest] = memVal;
if (inc_result >= 0)
  memoryWriteWord (rf[SrcA], inc_result);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

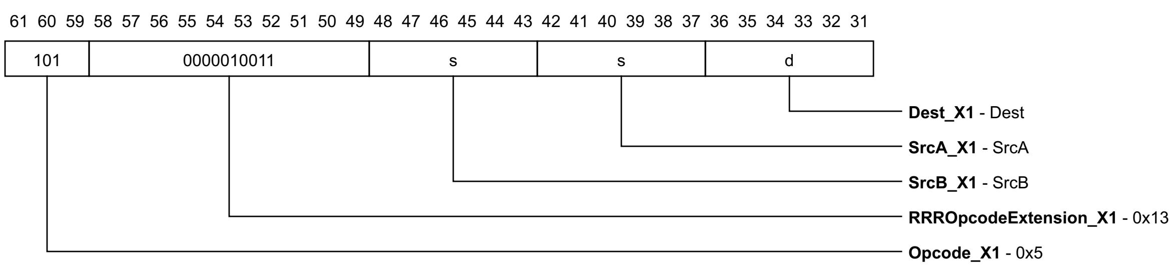

Figure 10-287: fetchaddgez4 in X1 Bits Encoding

fetchand

Fetch and And

Syntax

```
fetchand Dest, SrcA, SrcB
```

Example

```
fetchand r5, r6, r7
```

Description

Read and return the 8-byte value in memory at the address held in the first source register. The value in memory is logical ANDed with the 8-byte quantity from the second source register. The read and logical AND are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = memoryReadDoubleWord (rf[SrcA]);
rf[Dest] = memVal;
memoryWriteDoubleWord (rf[SrcA], memVal & rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

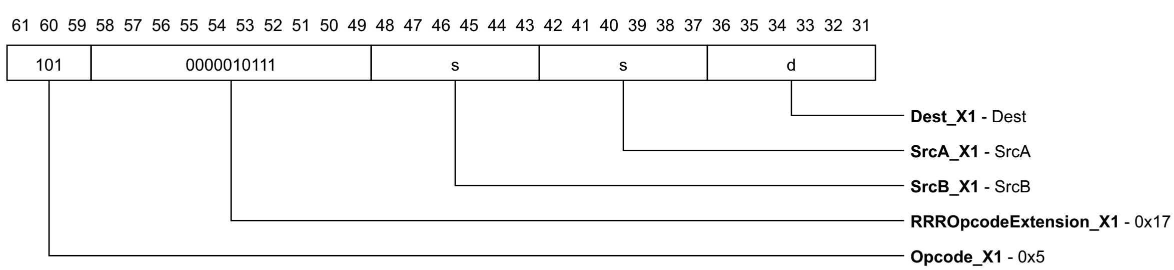

Figure 10-288: fetchand in X1 Bits Encoding

fetchand4

Fetch and And Four Bytes

Syntax

```
fetchand4 Dest, SrcA, SrcB
```

Example

```
fetchand4 r5, r6, r7
```

Description

Read and return the 4-byte value in memory at the address held in the first source register. The value in memory is logical ANDed with the 4-byte quantity from the second source register. The read and logical AND are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = signExtend32 (memoryReadWord (rf[SrcA]));
rf[Dest] = memVal;
memoryWriteWord (rf[SrcA], memVal & rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

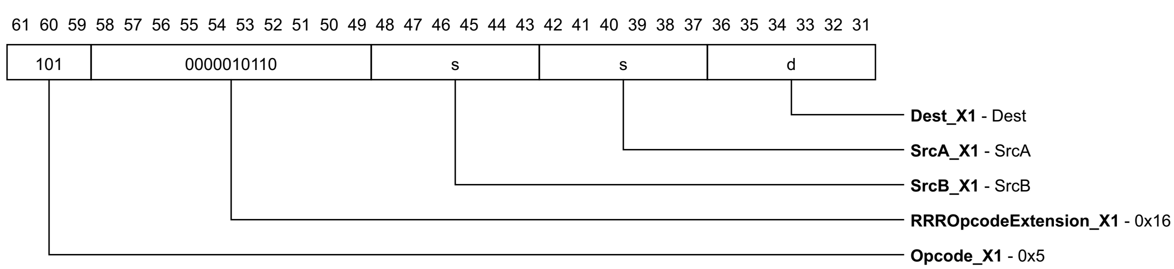

Figure 10-289: fetchand4 in X1 Bits Encoding

fetchor

Fetch and Or

Syntax

```
fetchor Dest, SrcA, SrcB
```

Example

```
fetchor r5, r6, r7
```

Description

Read and return the 8-byte value in memory at the address held in the first source register. The value in memory is logical ORed with the 8-byte quantity from the second source register. The read and logical OR are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = memoryReadDoubleWord (rf[SrcA]);
rf[Dest] = memVal;
memoryWriteDoubleWord (rf[SrcA], memVal | rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

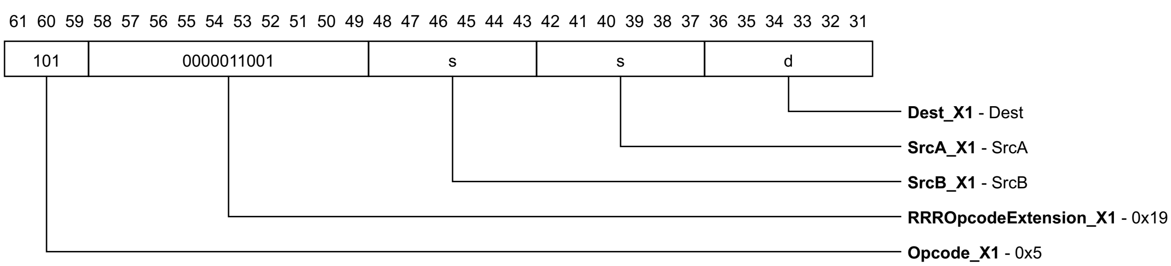

Figure 10-290: fetchor in X1 Bits Encoding

fetchor4

Fetch and Or Four Bytes

Syntax

```
fetchor4 Dest, SrcA, SrcB
```

Example

```
fetchor4 r5, r6, r7
```

Description

Read and return the 4-byte value in memory at the address held in the first source register. The value in memory is logical ORed with the 4-byte quantity from the second source register. The read and logical OR are atomic thus this instruction can be used for synchronization purposes. This instruction only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
uint64_t memVal = signExtend32 (memoryReadWord (rf[SrcA]));
rf[Dest] = memVal;
memoryWriteWord (rf[SrcA], memVal | rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

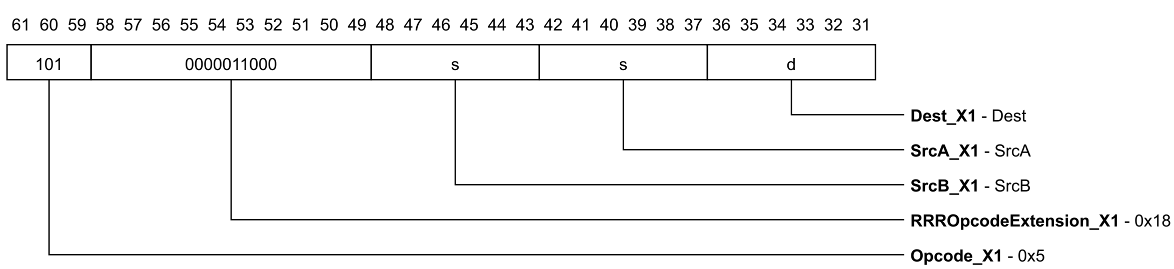

Figure 10-291: fetchor4 in X1 Bits Encoding

ld

Load

Syntax

```
ld Dest, Src
```

Example

```
ld r5, r6
```

Description

Load an 8-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to an 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
rf[Dest] = memoryReadDoubleWord (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

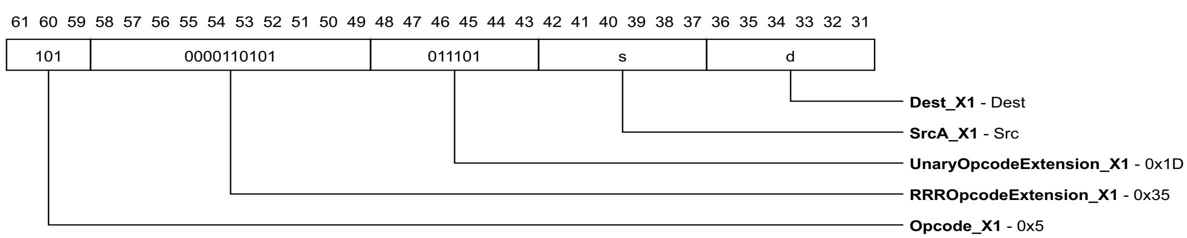

Figure 10-292: ld in X1 Bits Encoding

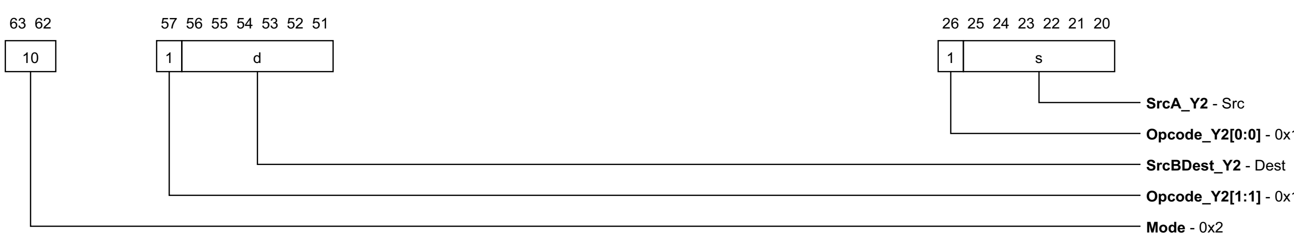

Figure 10-293: ld in Y2 Bits Encoding

ld1s

Load One Byte Signed

Syntax

```
ld1s Dest, Src
```

Example

```
ld1s r5, r6
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is sign-extended to a complete word.

Functional Description

```
rf[Dest] = signExtend8 (memoryReadByte (rf[Src]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

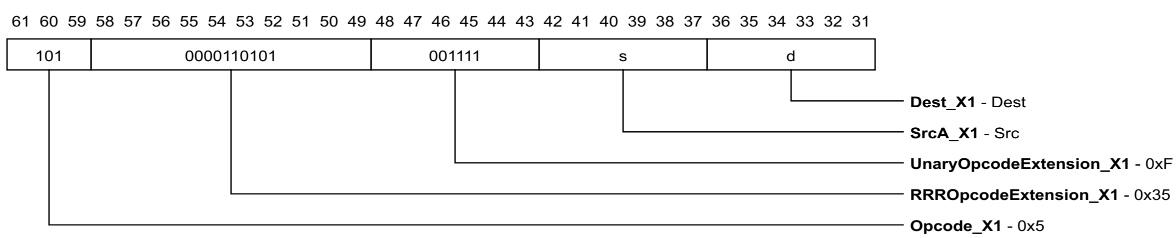

Figure 10-294: ld1s in X1 Bits Encoding

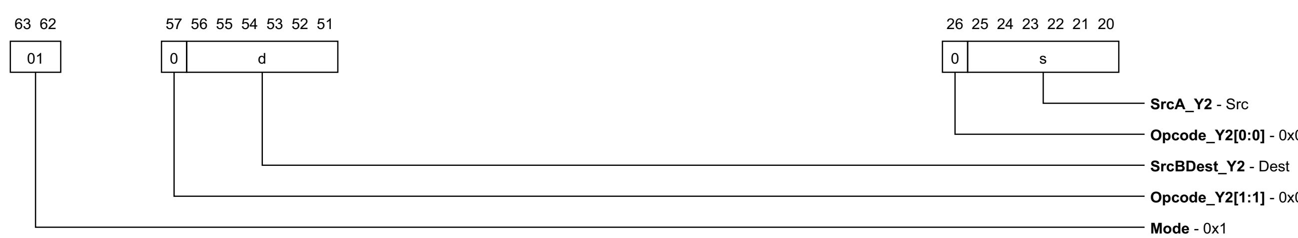

Figure 10-295: ld1s in Y2 Bits Encoding

ld1s_add

Load One Byte Signed and Add

Syntax

```
ld1s_add Dest, SrcA, Imm8
```

Example

```
ld1s_add r5, r6, 5
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is sign-extended to a complete word. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = signExtend8 (memoryReadByte (rf[SrcA]));
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

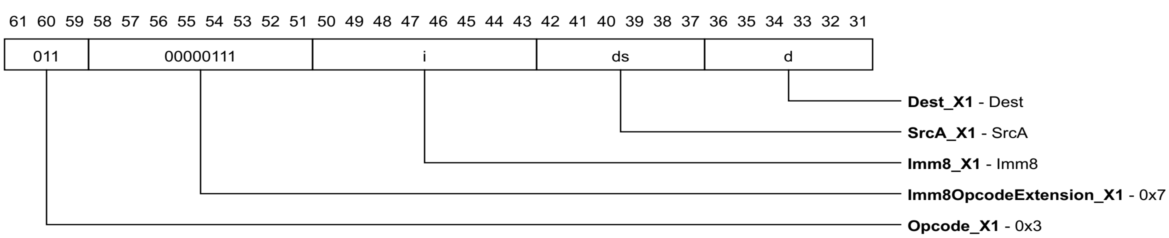

Figure 10-296: ld1s_add in X1 Bits Encoding

ld1u

Load One Byte Unsigned

Syntax

```
ld1u Dest, Src
```

Example

```
ld1u r5, r6
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is zero extended to a complete word.

Functional Description

```
rf[Dest] = memoryReadByte (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

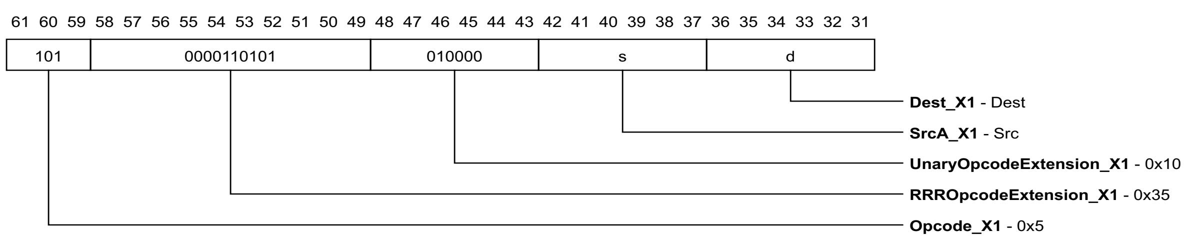

Figure 10-297: ld1u in X1 Bits Encoding

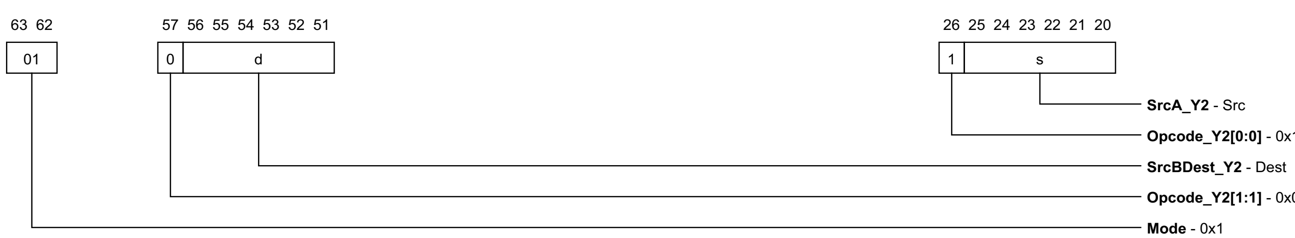

Figure 10-298: ld1u in Y2 Bits Encoding

ld1u_add

Load One Byte Unsigned and Add

Syntax

```
ld1u_add Dest, SrcA, Imm8
```

Example

```
ld1u_add r5, r6, 5
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is zero-extended to a complete word. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = memoryReadByte (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

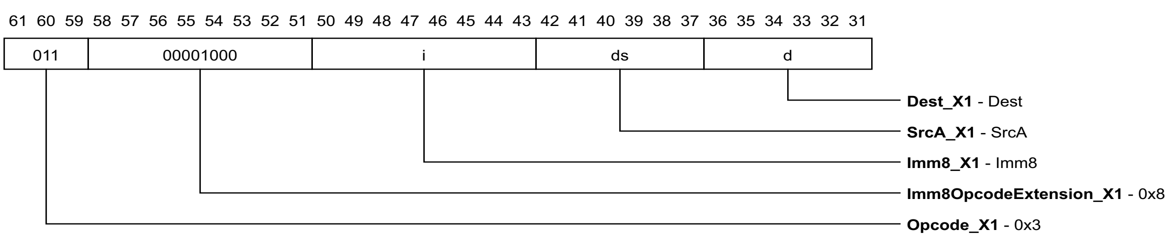

Figure 10-299: ld1u_add in X1 Bits Encoding

ld2s

Load Two Bytes Signed

Syntax

```
ld2s Dest, Src
```

Example

```
ld2s r5, r6
```

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is sign-extended to a complete word.

Functional Description

```
rf[Dest] = signExtend16 (memoryReadHalfWord (rf[Src]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

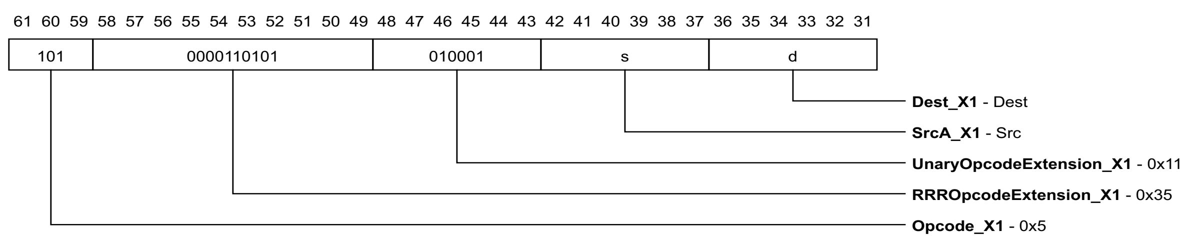

Figure 10-300: ld2s in X1 Bits Encoding

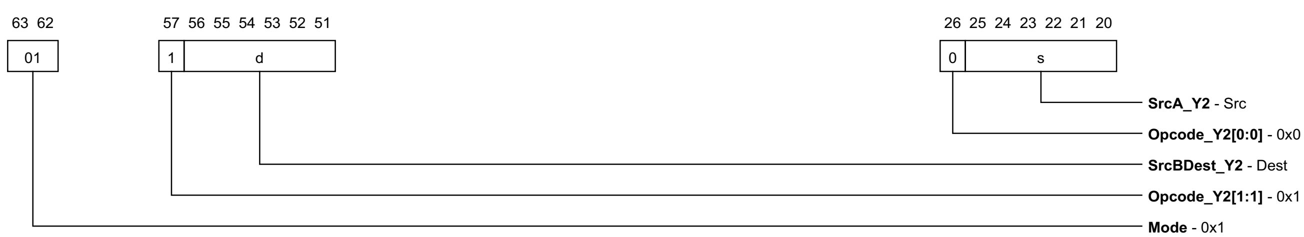

Figure 10-301: ld2s in Y2 Bits Encoding

ld2s_add

Load Two Bytes Signed and Add

Syntax

```
ld2s_add Dest, SrcA, Imm8
```

Example

```
ld2s_add r5, r6, 5
```

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is sign-extended to a complete word. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = signExtend16 (memoryReadHalfWord (rf[SrcA]));
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

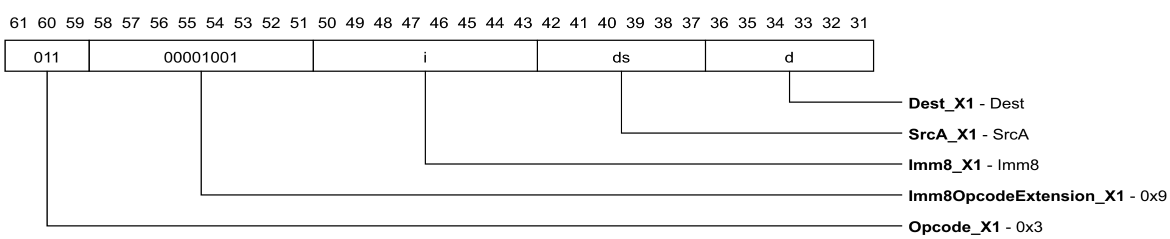

Figure 10-302: ld2s_add in X1 Bits Encoding

ld2u

Load Two Bytes Unsigned

Syntax

```
ld2u Dest, Src
```

Example

```
ld2u r5, r6
```

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is zero extended to a complete word.

Functional Description

```
rf[Dest] = memoryReadHalfWord (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

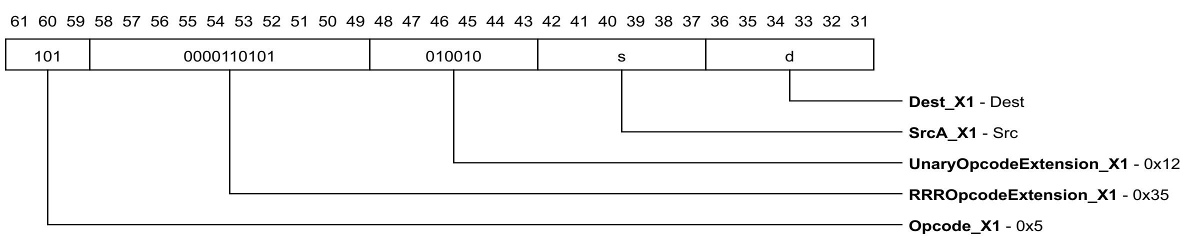

Figure 10-303: ld2u in X1 Bits Encoding

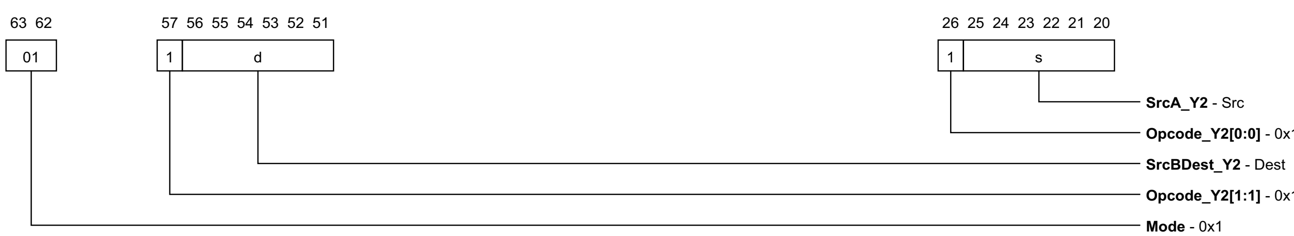

Figure 10-304: ld2u in Y2 Bits Encoding

ld2u_add

Load Two Bytes Unsigned and Add

Syntax

```
ld2u_add Dest, SrcA, Imm8
```

Example

```
ld2u_add r5, r6, 5
```

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is zero extended to a complete word. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = memoryReadHalfWord (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

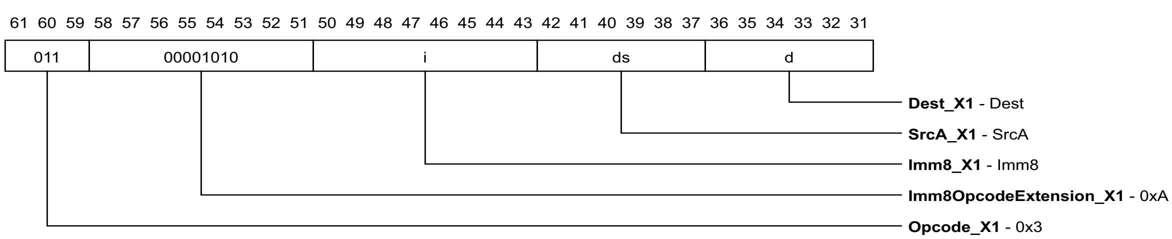

Figure 10-305: ld2u_add in X1 Bits Encoding

ld4s

Load Four Bytes Signed

Syntax

```
ld4s Dest, Src
```

Example

```
ld4s r5, r6
```

Description

Load a 4-byte signed quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
rf[Dest] = signExtend32 (memoryReadWord (rf[Src]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

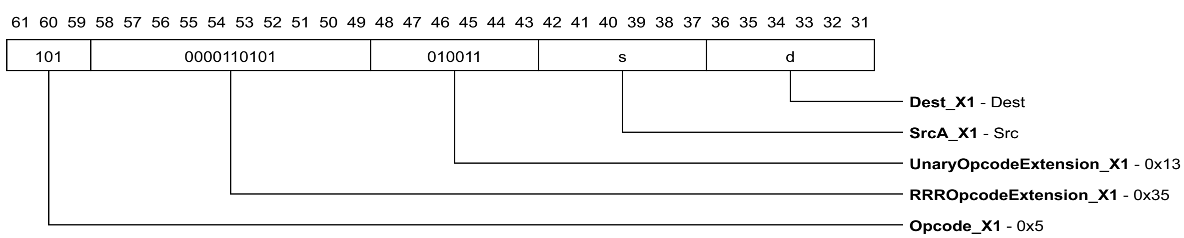

Figure 10-306: ld4s in X1 Bits Encoding

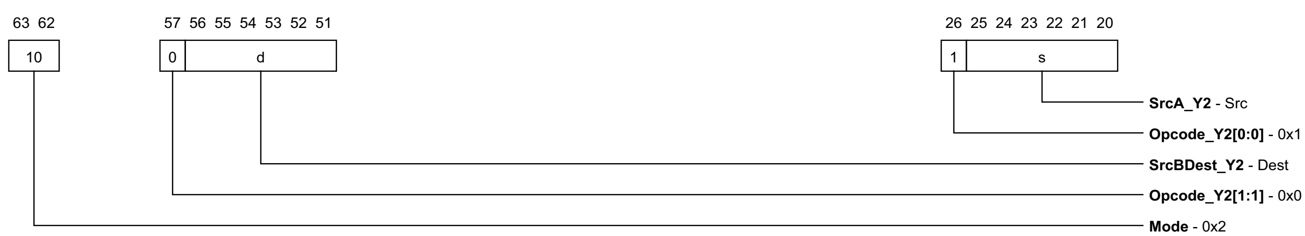

Figure 10-307: ld4s in Y2 Bits Encoding

ld4s_add

Load Four Bytes Signed and Add

Syntax

```
ld4s_add Dest, SrcA, Imm8
```

Example

```
ld4s_add r5, r6, 5
```

Description

Load a 4-byte signed quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = signExtend32 (memoryReadWord (rf[SrcA]));
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

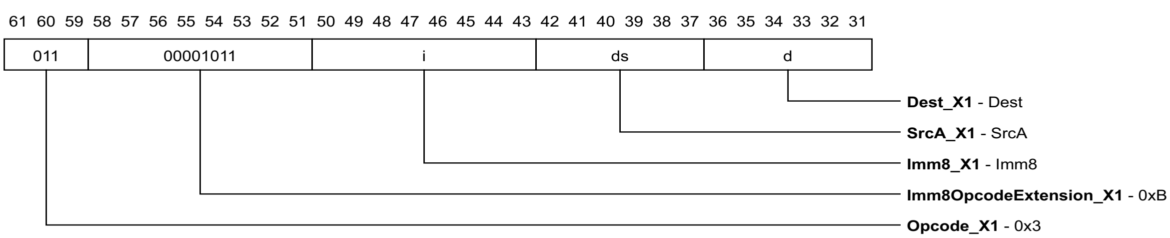

Figure 10-308: ld4s_add in X1 Bits Encoding

ld4u

Load Four Bytes Unsigned

Syntax

```
ld4u Dest, Src
```

Example

```
ld4u r5, r6
```

Description

Load a 4-byte unsigned quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
rf[Dest] = memoryReadWord (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

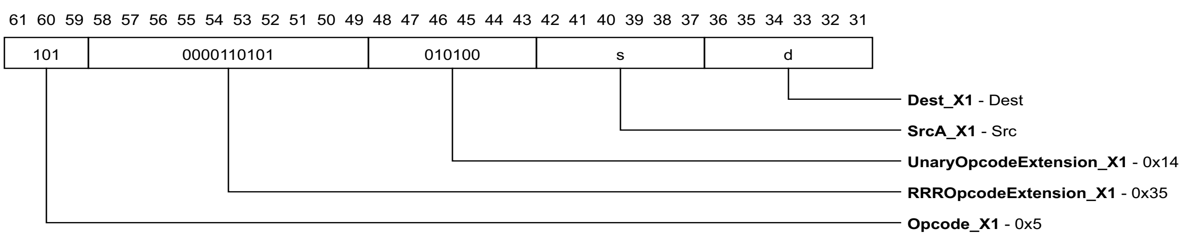

Figure 10-309: ld4u in X1 Bits Encoding

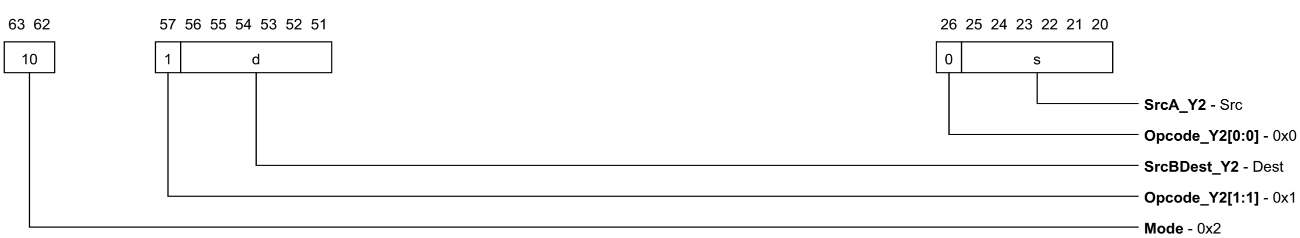

Figure 10-310: ld4u in Y2 Bits Encoding

ld4u_add

Load Four Bytes Unsigned and Add

Syntax

```
ld4u_add Dest, SrcA, Imm8
```

Example

```
ld4u_add r5, r6, 5
```

Description

Load a 4-byte unsigned quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = memoryReadWord (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

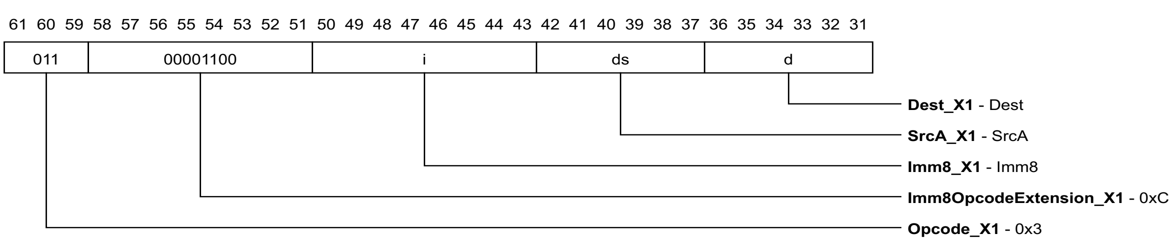

Figure 10-311: ld4u_add in X1 Bits Encoding

ld_add

Load and Add

Syntax

```
ld_add Dest, SrcA, Imm8
```

Example

```
ld_add r5, r6, 5
```

Description

Load an 8-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to an 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = memoryReadDoubleWord (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

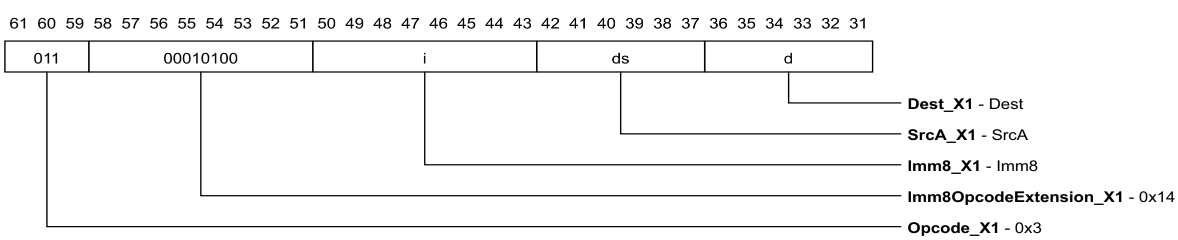

Figure 10-312: ld_add in X1 Bits Encoding

ldna

Load No Alignment Trap

Syntax

```
ldna Dest, Src
```

Example

```
ldna r5, r6
```

Description

Load an 8-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand and the bottom three bits are set to zero. No Unaligned Data Reference interrupts are caused by this instruction.

Functional Description

```
rf[Dest] = memoryReadDoubleWordNA (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

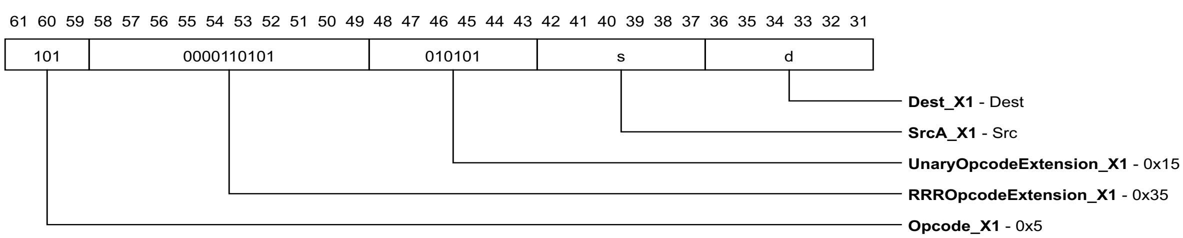

Figure 10-313: ldna in X1 Bits Encoding

ldna_add

Load No Alignment Trap and Add

Syntax

```
ldna_add Dest, SrcA, Imm8
```

Example

```
ldna_add r5, r6, 5
```

Description

Load an 8-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand and the bottom three bits are set to zero. No Unaligned Data Reference interrupts are caused by this instruction. Add the signed immediate argument to the address register.

Functional Description

```
rf[Dest] = memoryReadDoubleWordNA (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

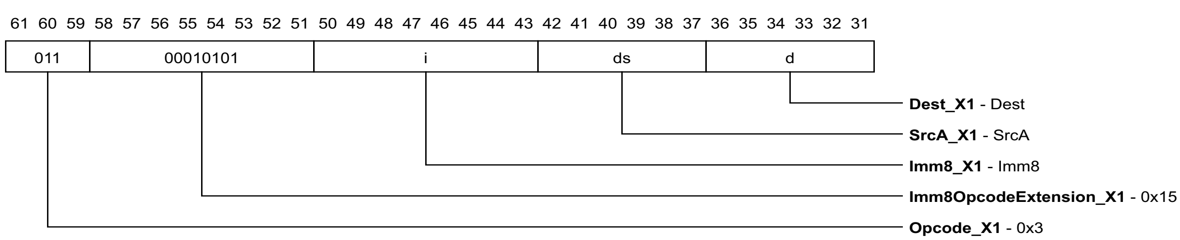

Figure 10-314: ldna_add in X1 Bits Encoding

ldnt

Load Non-Temporal

Syntax

```
ldnt Dest, Src
```

Example

```
ldnt r5, r6
```

Description

Load an 8-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to an 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadDoubleWordNonTemporal (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

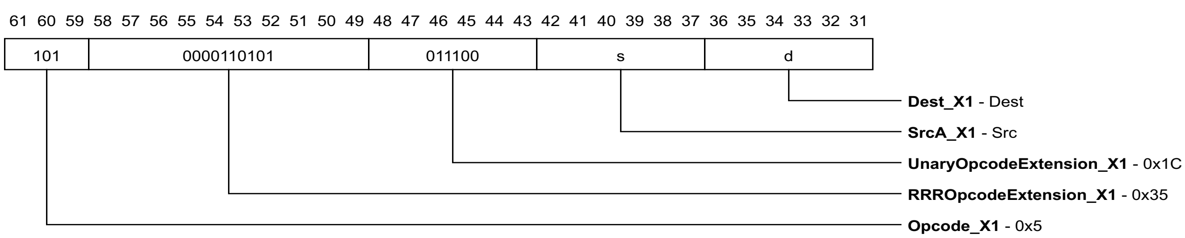

Figure 10-315: ldnt in X1 Bits Encoding

ldnt1s

Load Non-Temporal One Byte Signed

Syntax

```
ldnt1s Dest, Src
```

Example

```
ldnt1s r5, r6
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is sign-extended to a complete word. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = signExtend8 (memoryReadByteNonTemporal (rf[Src]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

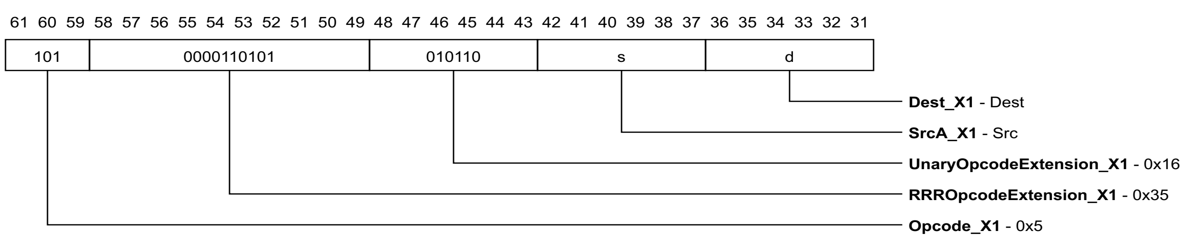

Figure 10-316: ldnt1s in X1 Bits Encoding

ldnt1s_add

Load Non-Temporal One Byte Signed and Add

Syntax

```
ldnt1s_add Dest, SrcA, Imm8
```

Example

```
ldnt1s_add r5, r6, 5
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is sign-extended to a complete word. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = signExtend8 (memoryReadByteNonTemporal (rf[SrcA]));
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

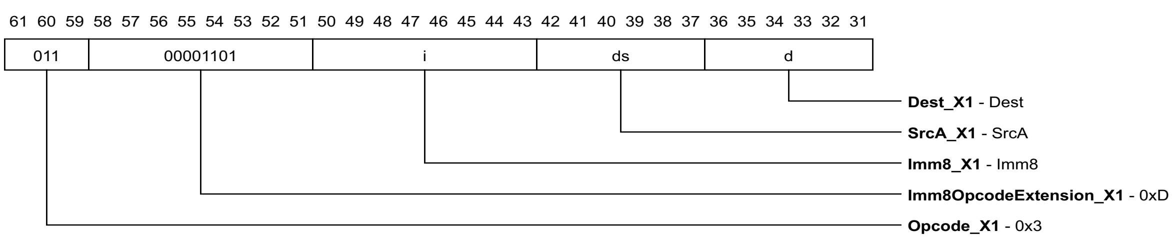

Figure 10-317: ldnt1s_add in X1 Bits Encoding

ldnt1u

Load Non-Temporal One Byte Unsigned

Syntax

```
ldnt1u Dest, Src
```

Example

```
ldnt1u r5, r6
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is zero extended to a complete word. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadByteNonTemporal (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

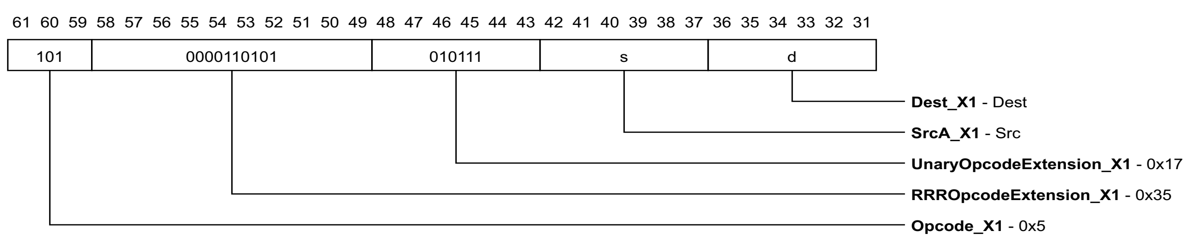

Figure 10-318: ldnt1u in X1 Bits Encoding

ldnt1u_add

Load Non-Temporal One Byte Unsigned and Add

Syntax

```
ldnt1u_add Dest, SrcA, Imm8
```

Example

```
ldnt1u_add r5, r6, 5
```

Description

Load a byte from memory into the destination register. The address of the value to be loaded is read from the source operand. The value read from memory is zero-extended to a complete word. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadByteNonTemporal (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

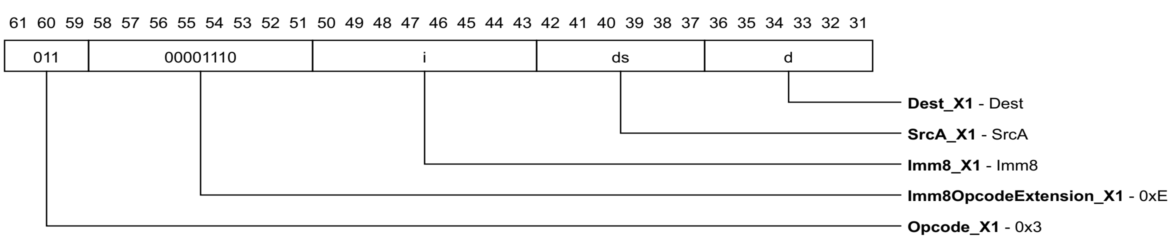

Figure 10-319: ldnt1u_add in X1 Bits Encoding

ldnt2s

Load Non-Temporal Two Bytes Signed

Syntax

`ldnt2s Dest, Src`

Example

`ldnt2s r5, r6`

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is sign-extended to a complete word. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = signExtend16 (memoryReadHalfWordNonTemporal (rf[Src]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

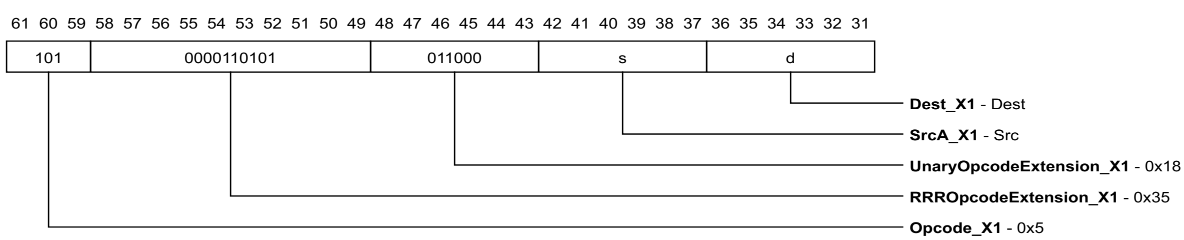

Figure 10-320: ldnt2s in X1 Bits Encoding

ldnt2s_add

Load Non-Temporal Two Bytes Signed and Add

Syntax

```
ldnt2s_add Dest, SrcA, Imm8
```

Example

```
ldnt2s_add r5, r6, 5
```

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is sign-extended to a complete word. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = signExtend16 (memoryReadHalfWordNonTemporal (rf[SrcA]));
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

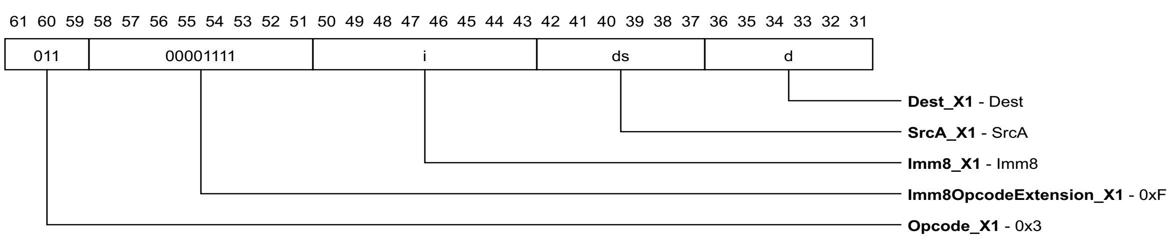

Figure 10-321: ldnt2s_add in X1 Bits Encoding

ldnt2u

Load Non-Temporal Two Bytes Unsigned

Syntax

```
ldnt2u Dest, Src
```

Example

```
ldnt2u r5, r6
```

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is zero extended to a complete word. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadHalfWordNonTemporal (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

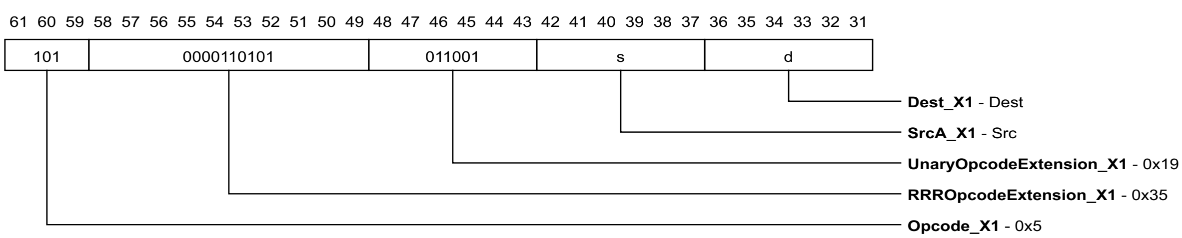

Figure 10-322: ldnt2u in X1 Bits Encoding

ldnt2u_add

Load Non-Temporal Two Bytes Unsigned and Add

Syntax

```
ldnt2u_add Dest, SrcA, Imm8
```

Example

```
ldnt2u_add r5, r6, 5
```

Description

Load a 2-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is zero extended to a complete word. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadHalfWordNonTemporal (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

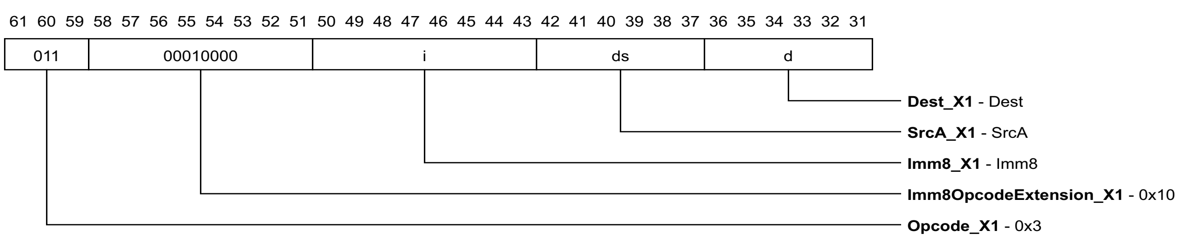

Figure 10-323: ldnt2u_add in X1 Bits Encoding

ldnt4s

Load Non-Temporal Four Bytes Signed

Syntax

```
ldnt4s Dest, Src
```

Example

```
ldnt4s r5, r6
```

Description

Load a 4-byte signed quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = signExtend32 (memoryReadWordNonTemporal (rf[Src]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

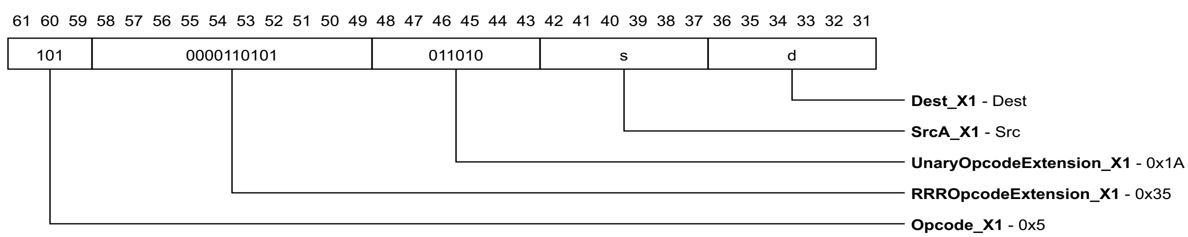

Figure 10-324: ldnt4s in X1 Bits Encoding

ldnt4s_add

Load Non-Temporal Four Bytes Signed and Add

Syntax

```
ldnt4s_add Dest, SrcA, Imm8
```

Example

```
ldnt4s_add r5, r6, 5
```

Description

Load a 4-byte signed quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = signExtend32 (memoryReadWordNonTemporal (rf[SrcA]));
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

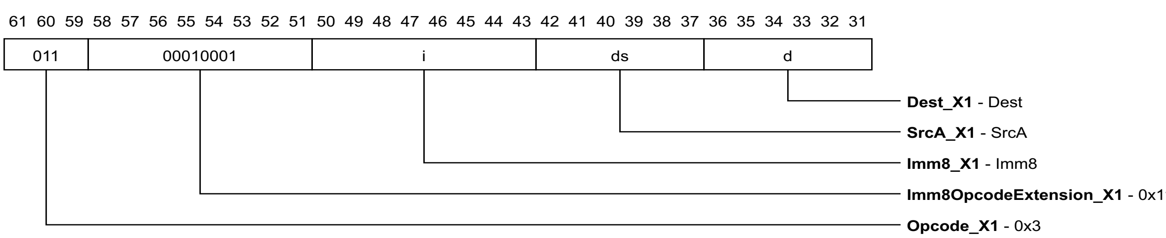

Figure 10-325: ldnt4s_add in X1 Bits Encoding

ldnt4u

Load Non-Temporal Four Bytes Unsigned

Syntax

```
ldnt4u Dest, Src
```

Example

```
ldnt4u r5, r6
```

Description

Load a 4-byte unsigned quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadWordNonTemporal (rf[Src]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

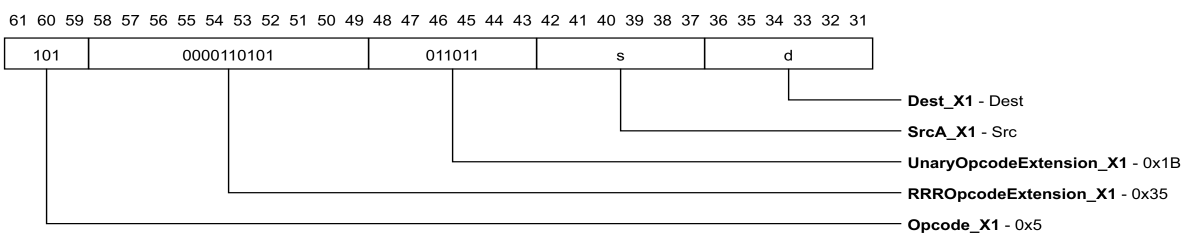

Figure 10-326: ldnt4u in X1 Bits Encoding

ldnt4u_add

Load Non-Temporal Four Bytes Unsigned and Add

Syntax

```
ldnt4u_add Dest, SrcA, Imm8
```

Example

```
ldnt4u_add r5, r6, 5
```

Description

Load a 4-byte unsigned quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadWordNonTemporal (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

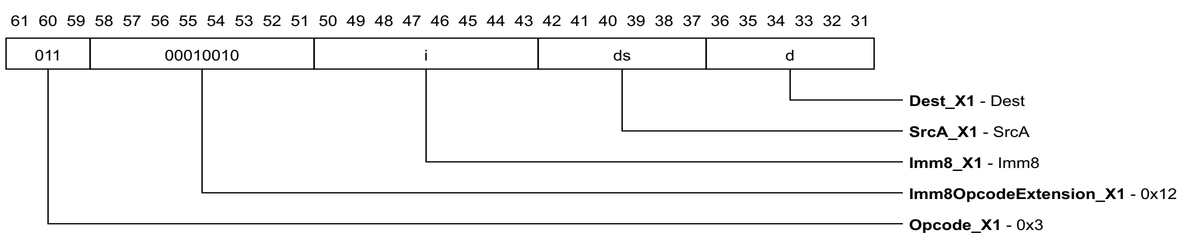

Figure 10-327: ldnt4u_add in X1 Bits Encoding

ldnt_add

Load Non-Temporal and Add

Syntax

```
ldnt_add Dest, SrcA, Imm8
```

Example

```
ldnt_add r5, r6, 5
```

Description

Load an 8-byte quantity from memory into the destination register. The address of the value to be loaded is read from the source operand. This load only operates for addresses aligned to an 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
rf[Dest] = memoryReadDoubleWordNonTemporal (rf[SrcA]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

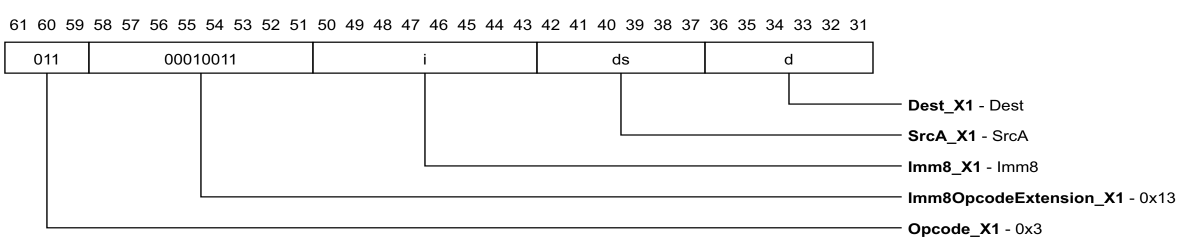

Figure 10-328: ldnt_add in X1 Bits Encoding

st

Store

Syntax

```
st SrcA, SrcB
```

Example

```
st r5, r6
```

Description

Store an 8-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to an 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
memoryWriteDoubleWord (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

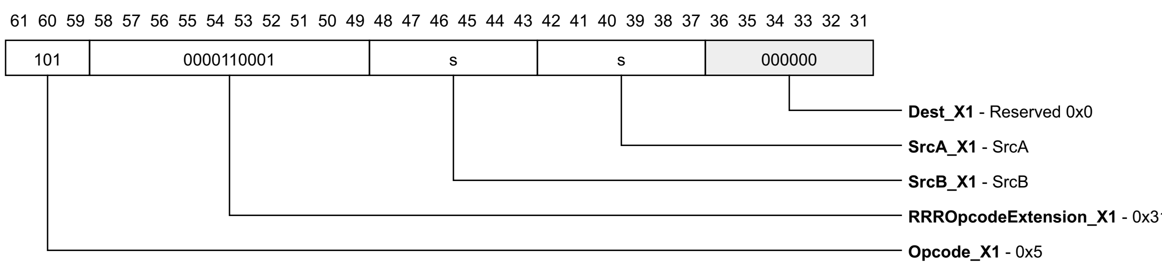

Figure 10-329: st in X1 Bits Encoding

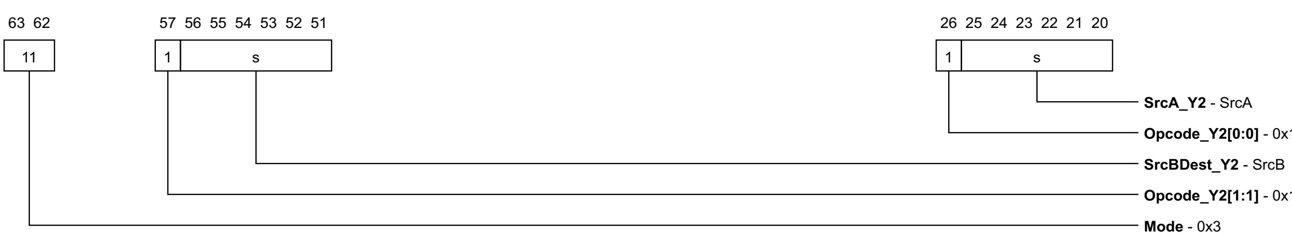

Figure 10-330: st in Y2 Bits Encoding

st1

Store Byte

Syntax

```
st1 SrcA, SrcB
```

Example

```
st1 r5, r6
```

Description

Store a byte from the second source register into memory at the address held in the first source register.

Functional Description

```
memoryWriteByte (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

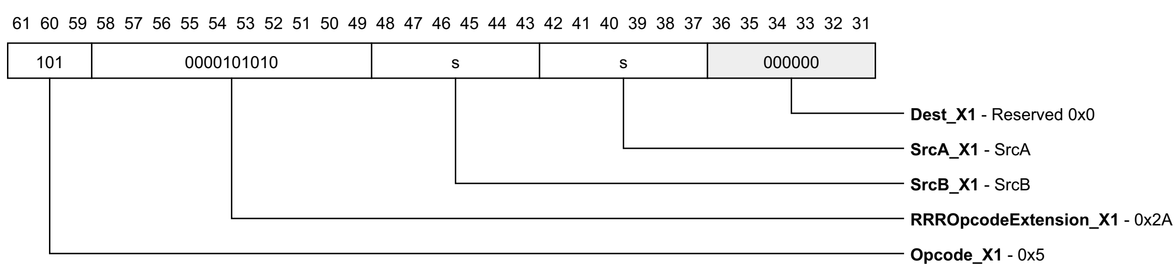

Figure 10-331: st1 in X1 Bits Encoding

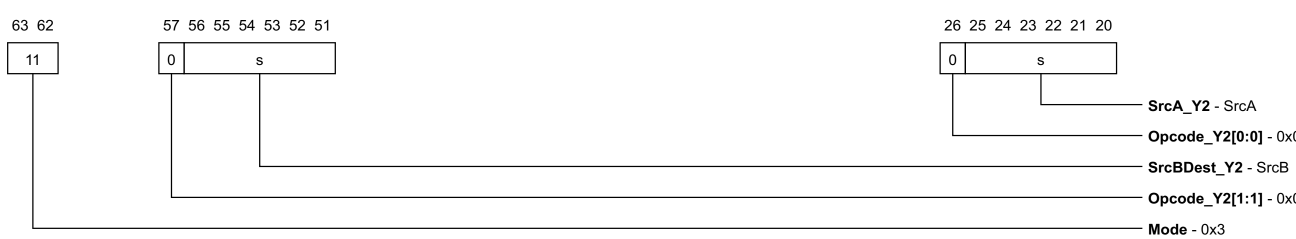

Figure 10-332: st1 in Y2 Bits Encoding

st1_add

Store Byte and Add

Syntax

```
st1_add SrcA, SrcB, Imm8
```

Example

```
st1_add r5, r6, 5
```

Description

Store a byte from the second source register into memory at the address held in the first source register. Add the signed immediate argument to the address register.

Functional Description

```
memoryWriteByte (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

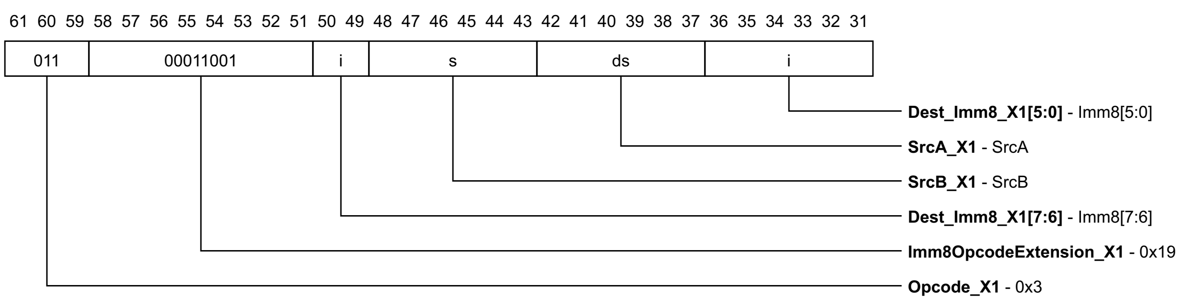

Figure 10-333: st1_add in X1 Bits Encoding

st2

Store Two Bytes

Syntax

```
st2 SrcA, SrcB
```

Example

```
st2 r5, r6
```

Description

Store a 2-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
memoryWriteHalfWord (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

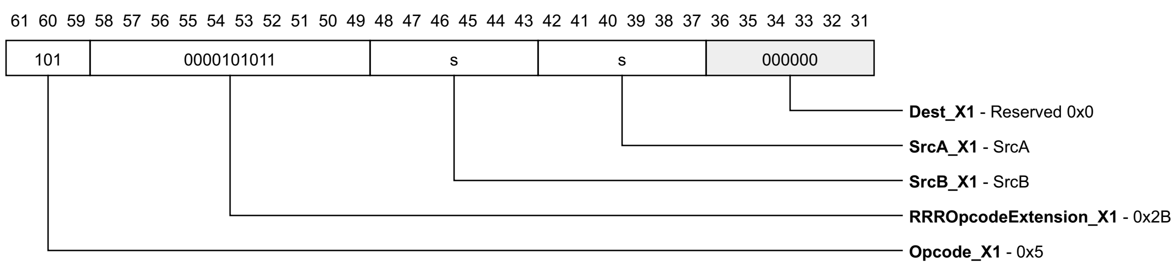

Figure 10-334: st2 in X1 Bits Encoding

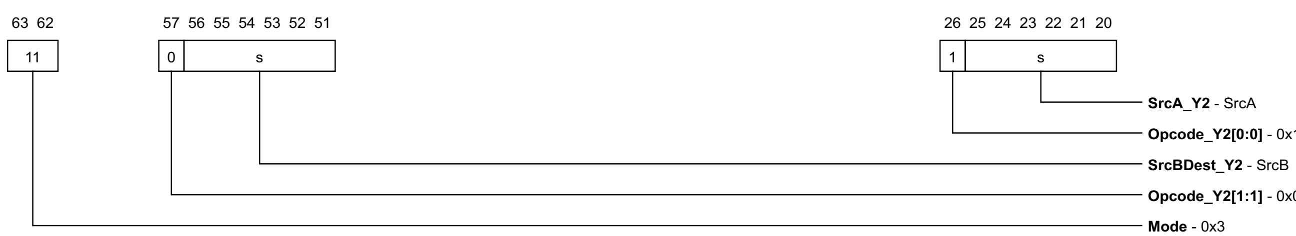

Figure 10-335: st2 in Y2 Bits Encoding

st2_add

Store Two Bytes and Add

Syntax

```
st2_add SrcA, SrcB, Imm8
```

Example

```
st2_add r5, r6, 5
```

Description

Store a 2-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

Functional Description

```
memoryWriteHalfWord (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

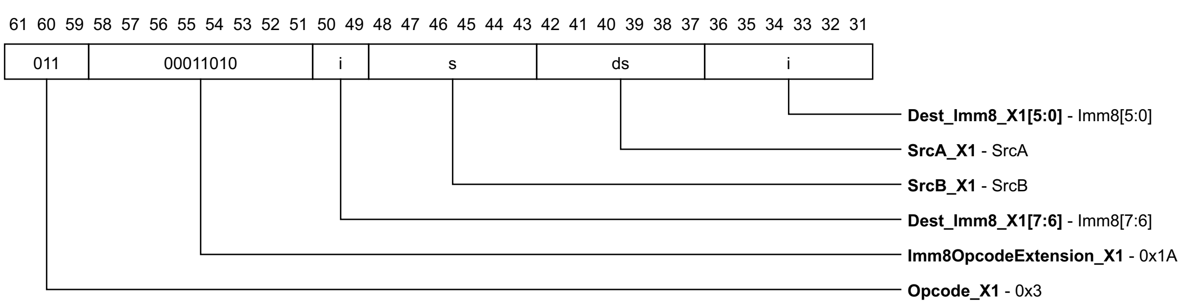

Figure 10-336: st2_add in X1 Bits Encoding

st4

Store Four Bytes

Syntax

```
st4 SrcA, SrcB
```

Example

```
st4 r5, r6
```

Description

Store a 4-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
memoryWriteWord (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

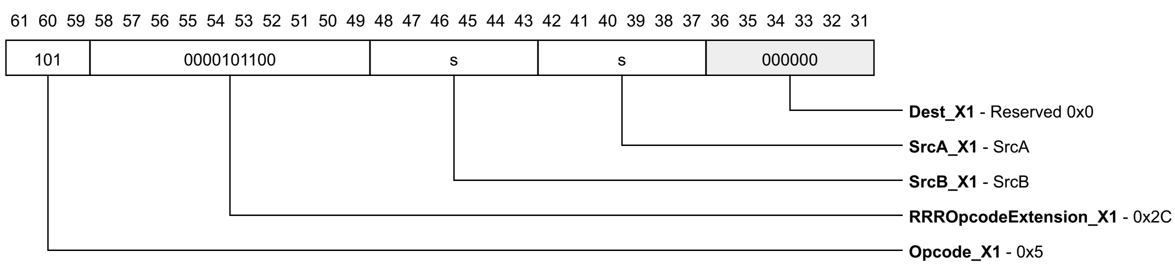

Figure 10-337: st4 in X1 Bits Encoding

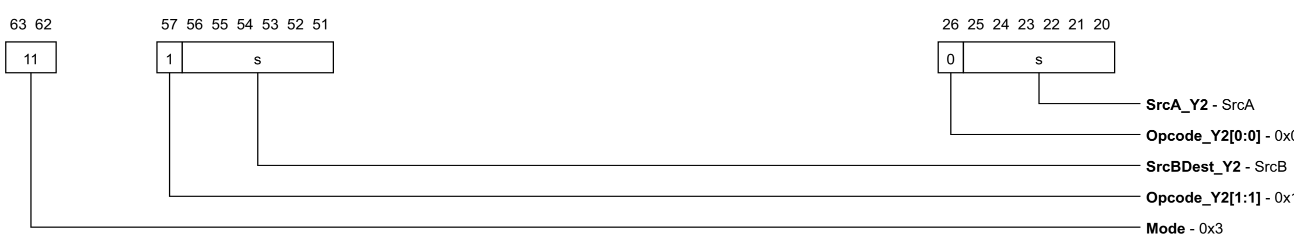

Figure 10-338: st4 in Y2 Bits Encoding

st4_add

Store Four Bytes and Add

Syntax

```
st4_add SrcA, SrcB, Imm8
```

Example

```
st4_add r5, r6, 5
```

Description

Store a 4-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

Functional Description

```
memoryWriteWord (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

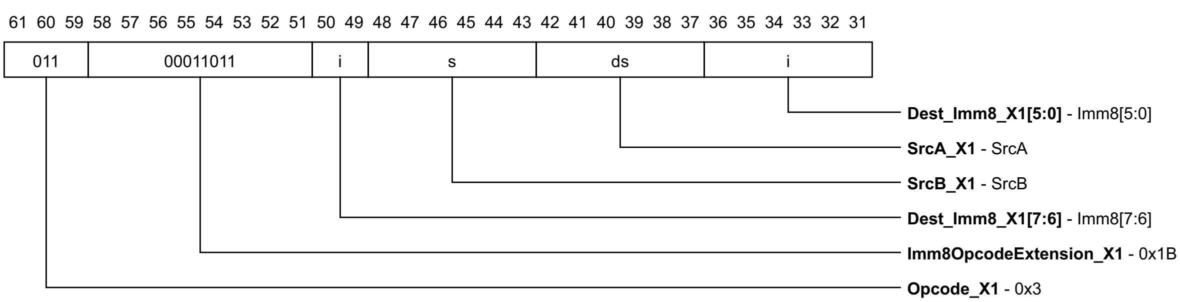

Figure 10-339: st4_add in X1 Bits Encoding

st_add

Store and Add

Syntax

```
st_add SrcA, SrcB, Imm8
```

Example

```
st_add r5, r6, 5
```

Description

Store an 8-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

Functional Description

```
memoryWriteDoubleWord (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

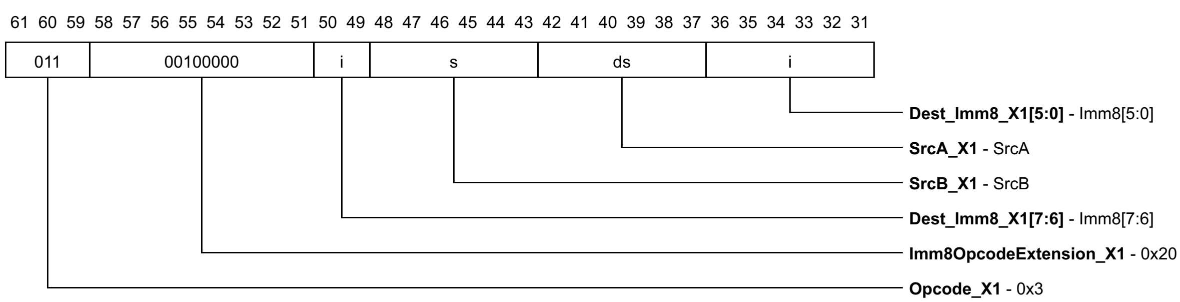

Figure 10-340: st_add in X1 Bits Encoding

stnt

Store Non-Temporal

Syntax
```
stnt SrcA, SrcB
```

Example
```
stnt r5, r6
```

Description

Store an 8-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to an 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description
```
memoryWriteDoubleWordNonTemporal (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

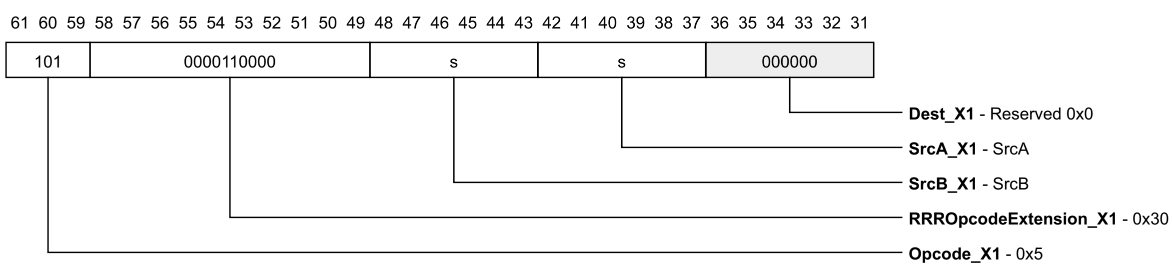

Figure 10-341: stnt in X1 Bits Encoding

stnt1

Store Non-Temporal Byte

Syntax

```
stnt1 SrcA, SrcB
```

Example

```
stnt1 r5, r6
```

Description

Store a byte from the second source register into memory at the address held in the first source register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
memoryWriteByteNonTemporal (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

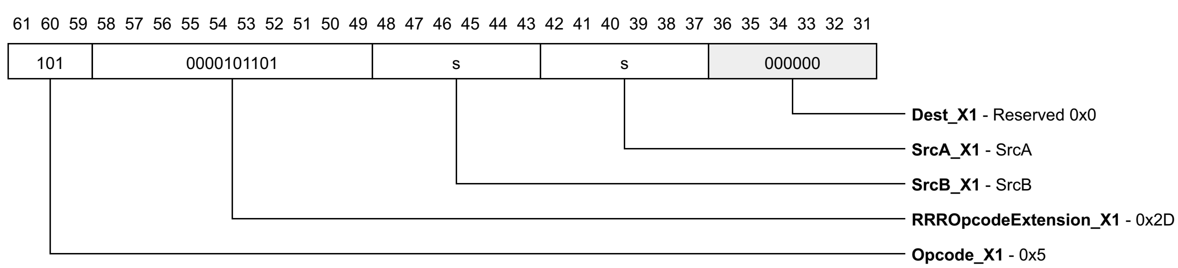

Figure 10-342: stnt1 in X1 Bits Encoding

stnt1_add

Store Non-Temporal Byte and Add

Syntax

```
stnt1_add SrcA, SrcB, Imm8
```

Example

```
stnt1_add r5, r6, 5
```

Description

Store a byte from the second source register into memory at the address held in the first source register. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
memoryWriteByteNonTemporal (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

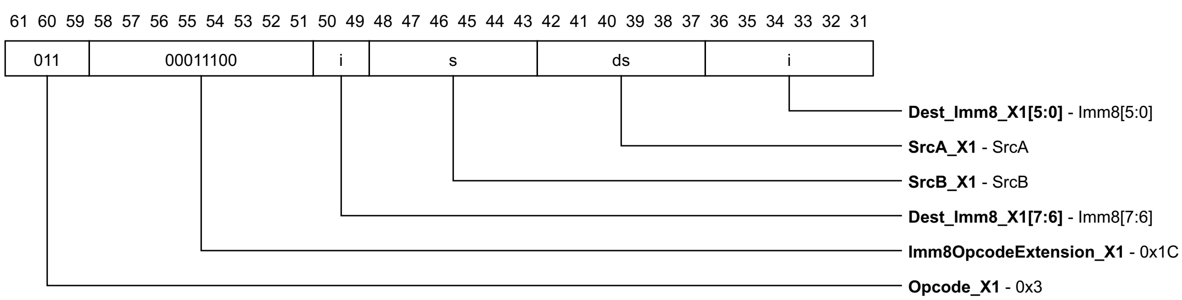

Figure 10-343: stnt1_add in X1 Bits Encoding

stnt2

Store Non-Temporal Two Bytes

Syntax

```
stnt2 SrcA, SrcB
```

Example

```
stnt2 r5, r6
```

Description

Store a 2-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
memoryWriteHalfWordNonTemporal (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

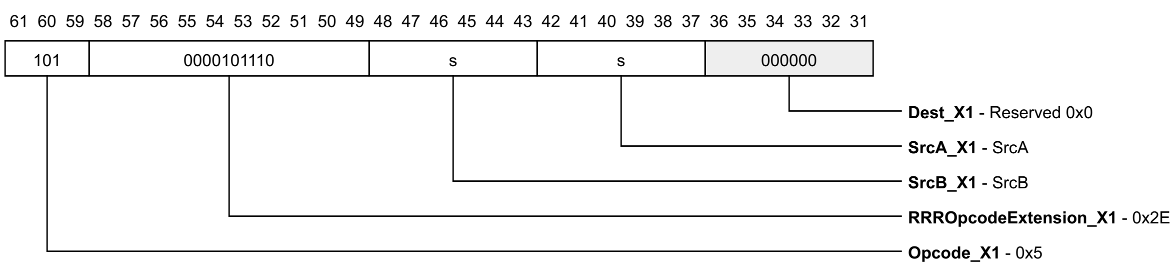

Figure 10-344: stnt2 in X1 Bits Encoding

stnt2_add

Store Non-Temporal Two Bytes and Add

Syntax

```
stnt2_add SrcA, SrcB, Imm8
```

Example

```
stnt2_add r5, r6, 5
```

Description

Store a 2-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 2-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
memoryWriteHalfWordNonTemporal (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

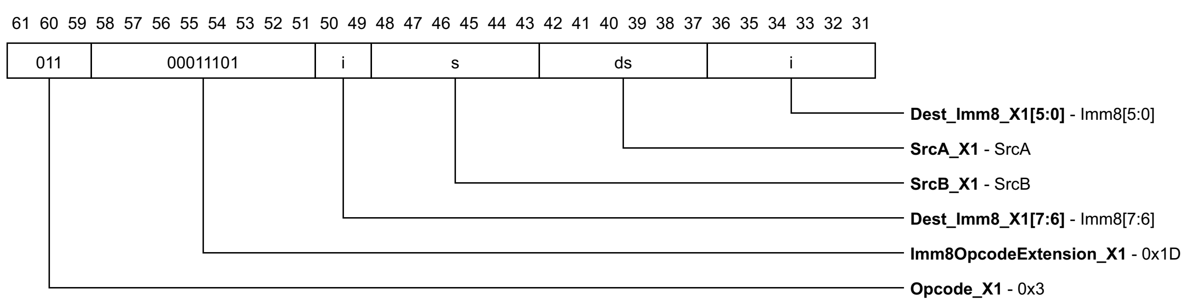

Figure 10-345: stnt2_add in X1 Bits Encoding

stnt4

Store Non-Temporal Four Bytes

Syntax

```
stnt4 SrcA, SrcB
```

Example

```
stnt4 r5, r6
```

Description

Store a 4-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
memoryWriteWordNonTemporal (rf[SrcA], rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

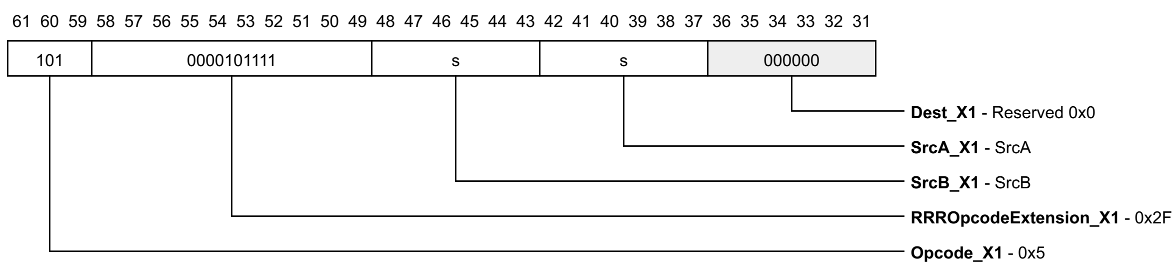

Figure 10-346: stnt4 in X1 Bits Encoding

stnt4_add

Store Non-Temporal Four Bytes and Add

Syntax

```
stnt4_add SrcA, SrcB, Imm8
```

Example

```
stnt4_add r5, r6, 5
```

Description

Store a 4-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 4-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
memoryWriteWordNonTemporal (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

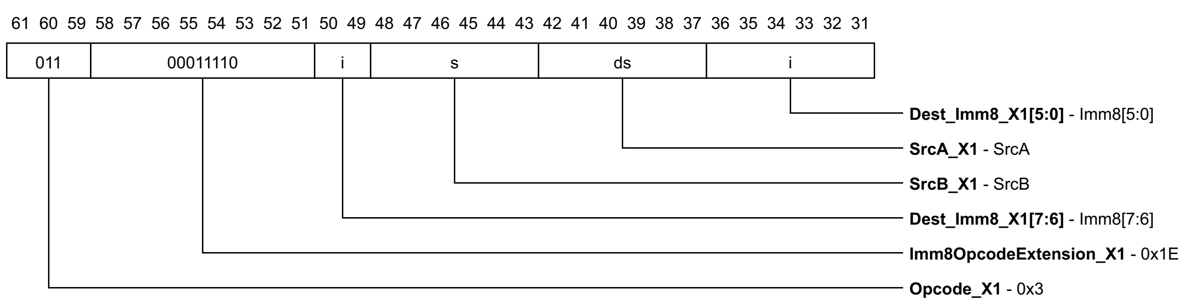

Figure 10-347: stnt4_add in X1 Bits Encoding

stnt_add

Store Non-Temporal and Add

Syntax

```
stnt_add SrcA, SrcB, Imm8
```

Example

```
stnt_add r5, r6, 5
```

Description

Store an 8-byte quantity from the second source register into memory at the address held in the first source register. This store only operates for addresses aligned to a 8-byte boundary. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register. The cache system is given an indication that the data will not be re-accessed in the near future.

Functional Description

```
memoryWriteDoubleWordNonTemporal (rf[SrcA], rf[SrcB]);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

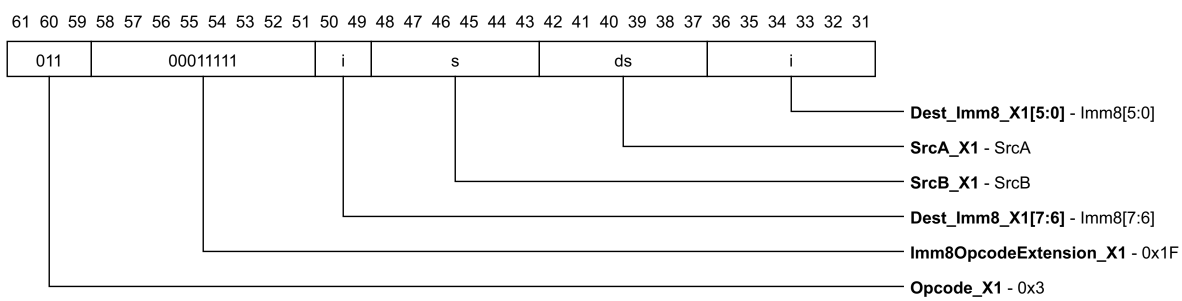

Figure 10-348: stnt_add in X1 Bits Encoding

CHAPTER 11 MULTIPLY INSTRUCTIONS

11.1 Overview

The following sections provide detailed descriptions of multiply instructions listed alphabetically.

cmul	Complex Multiply
cmula	Complex Multiply Accumulate
cmulaf	Complex Multiply Accumulate Fixed Point
cmulf	Complex Multiply Fixed Point
cmulfr	Complex Multiply Fixed Point Round
cmulh	Complex Multiply High Result
cmulhr	Complex Multiply High Result Round
mul_hs_hs	Multiply High Signed High Signed
mul_hs_hu	Multiply High Signed High Unsigned
mul_hs_ls	Multiply High Signed Low Signed
mul_hs_lu	Multiply High Signed Low Unsigned
mul_hu_hu	Multiply High Unsigned High Unsigned
mul_hu_ls	Multiply High Unsigned Low Signed
mul_hu_lu	Multiply High Unsigned Low Unsigned
mul_ls_ls	Multiply Low Signed Low Signed
mul_ls_lu	Multiply Low Signed Low Unsigned
mul_lu_lu	Multiply Low Unsigned Low Unsigned
mula_hs_hs	Multiply Accumulate High Signed High Signed
mula_hs_hu	Multiply Accumulate High Signed High Unsigned
mula_hs_ls	Multiply Accumulate High Signed Low Signed
mula_hs_lu	Multiply Accumulate High Signed Low Unsigned
mula_hu_hu	Multiply Accumulate High Unsigned High Unsigned
mula_hu_ls	Multiply Accumulate High Unsigned Low Signed
mula_hu_lu	Multiply Accumulate High Unsigned Low Unsigned
mula_ls_ls	Multiply Accumulate Low Signed Low Signed
mula_ls_lu	Multiply Accumulate Low Signed Low Unsigned

11.2 Instructions

Multiply instructions are described in the sections that follow.

cmul

Complex Multiply

Syntax

```
cmul Dest, SrcA, SrcB
```

Example

```
cmul r5, r6, r7
```

Description

Multiply the 32-bit complex number in the low half of the first operand by the 32-bit complex number in the low half of the second operand, producing a 64-bit complex result. The 64-bit complex number is represented as a 32-bit signed real value in the lowest-order 32-bits and a 32-bit signed imaginary value in the high-order 32-bits.

Functional Description

```
uint64_t output = 0;
int32_t realA = signExtend16 (get2Byte (rf[SrcA], 0));
int32_t imagA = signExtend16 (get2Byte (rf[SrcA], 1));
int32_t realB = signExtend16 (get2Byte (rf[SrcB], 0));
int32_t imagB = signExtend16 (get2Byte (rf[SrcB], 1));
int32_t realRes = realA * realB - imagA * imagB;
int32_t imagRes = realA * imagB + imagA * realB;
output = set4Byte (output, 0, realRes);
output = set4Byte (output, 1, imagRes);
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

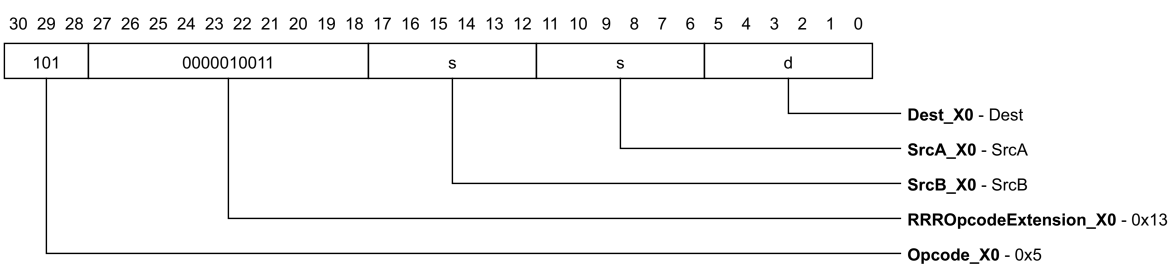

Figure 11-349: cmul in X0 Bits Encoding

cmula

Complex Multiply Accumulate

Syntax

```
cmula Dest, SrcA, SrcB
```

Example

```
cmula r5, r6, r7
```

Description

Multiply the 32-bit complex number in the low half of the first operand by the 32-bit complex number in the low half of the second operand, producing a 64-bit complex result. The 64-bit complex number is represented as a 32-bit signed real value in the lowest-order 32-bits and a 32-bit signed imaginary value in the high-order 32-bits. The complex multiply result is accumulated into the 64-bit complex number in the destination operand.

Functional Description

```
uint64_t output = 0;
int32_t realA = signExtend16 (get2Byte (rf[SrcA], 0));
int32_t imagA = signExtend16 (get2Byte (rf[SrcA], 1));
int32_t realB = signExtend16 (get2Byte (rf[SrcB], 0));
int32_t imagB = signExtend16 (get2Byte (rf[SrcB], 1));
int32_t realRes = realA * realB - imagA * imagB;
int32_t imagRes = realA * imagB + imagA * realB;
output = set4Byte (output, 0, realRes + get4Byte (rf[Dest], 0));
output = set4Byte (output, 1, imagRes + get4Byte (rf[Dest], 1));
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

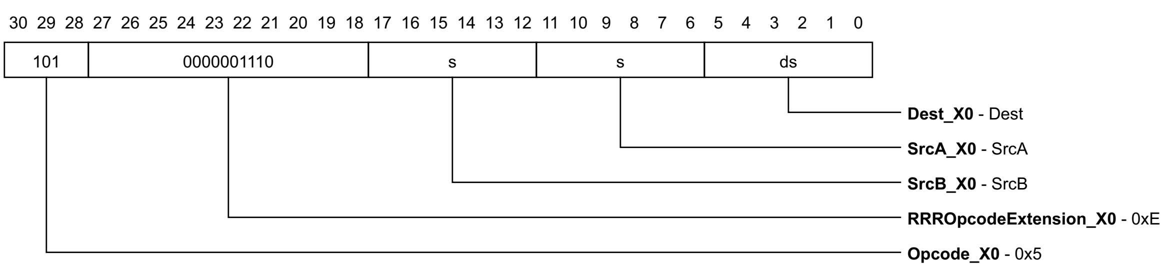

Figure 11-350: cmula in X0 Bits Encoding

cmulaf

Complex Multiply Accumulate Fixed Point

Syntax

```
cmulaf Dest, SrcA, SrcB
```

Example

```
cmulaf r5, r6, r7
```

Description

Multiply the 32-bit complex number in the low half of the first operand by the 32-bit complex number in the low half of the second operand, producing a 32-bit complex result. The 32-bit complex number is represented as a 16-bit signed real value in the lowest-order 16-bits and a 16-bit signed imaginary value in the high-order 16-bits. The complex multiply result is accumulated into the 32-bit complex number in the destination operand. The operands are treated as 16-bit signed fractions with the decimal point below the sign bit.

Functional Description

```
uint64_t output = 0;
int32_t realA = signExtend16 (get2Byte (rf[SrcA], 0));
int32_t imagA = signExtend16 (get2Byte (rf[SrcA], 1));
int32_t realB = signExtend16 (get2Byte (rf[SrcB], 0));
int32_t imagB = signExtend16 (get2Byte (rf[SrcB], 1));
int32_t realRes = realA * realB - imagA * imagB;
int32_t imagRes = realA * imagB + imagA * realB;
output = set2Byte (output, 0, (realRes >> 15) + get2Byte (rf[Dest], 0));
output = set2Byte (output, 1, (imagRes >> 15) + get2Byte (rf[Dest], 1));
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

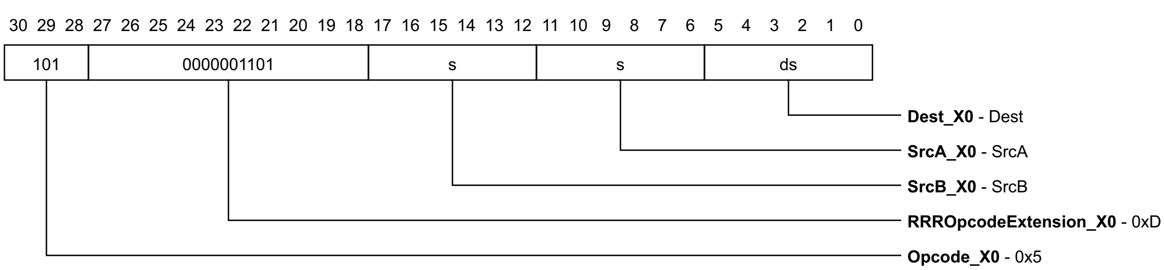

Figure 11-351: cmulaf in X0 Bits Encoding

cmulf

Complex Multiply Fixed Point

Syntax

```
cmulf Dest, SrcA, SrcB
```

Example

```
cmulf r5, r6, r7
```

Description

Multiply the 32-bit complex number in the low half of the first operand by the 32-bit complex number in the low half of the second operand, producing a 32-bit complex result. The 32-bit complex number is represented as a 16-bit signed real value in the lowest-order 16-bits and a 16-bit signed imaginary value in the high-order 16-bits. The operands are treated as 16-bit signed fractions with the decimal point below the sign bit.

Functional Description

```
uint64_t output = 0;
int32_t realA = signExtend16 (get2Byte (rf[SrcA], 0));
int32_t imagA = signExtend16 (get2Byte (rf[SrcA], 1));
int32_t realB = signExtend16 (get2Byte (rf[SrcB], 0));
int32_t imagB = signExtend16 (get2Byte (rf[SrcB], 1));
int32_t realRes = realA * realB - imagA * imagB;
int32_t imagRes = realA * imagB + imagA * realB;
output = set2Byte (output, 0, (realRes >> 15));
output = set2Byte (output, 1, (imagRes >> 15));
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

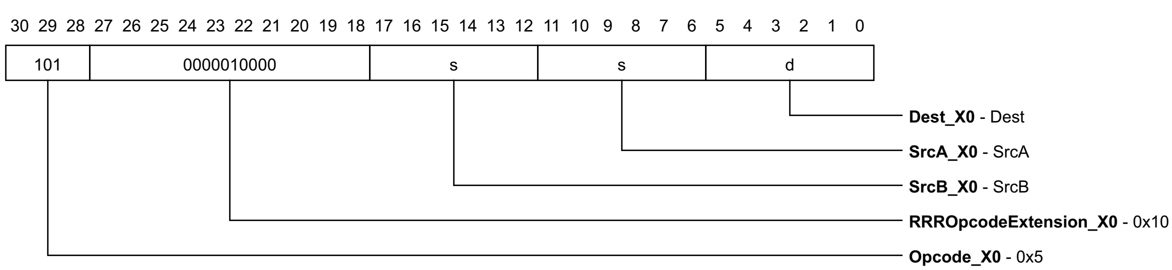

Figure 11-352: cmulf in X0 Bits Encoding

cmulfr

Complex Multiply Fixed Point Round

Syntax

```
cmulfr Dest, SrcA, SrcB
```

Example

```
cmulfr r5, r6, r7
```

Description

Multiply the 32-bit complex number in the low half of the first operand by the 32-bit complex number in the low half of the second operand, producing a rounded 32-bit complex result. The 32-bit complex number is represented as a 16-bit signed real value in the lowest-order 16-bits and a 16-bit signed imaginary value in the high-order 16-bits. The operands are treated as 16-bit signed fractions with the decimal point below the sign bit. The full-precision real and imaginary components of the multiplication are both rounded up while being reduced to 16-bit wide results.

Functional Description

```
uint64_t output = 0;
int32_t realA = signExtend16 (get2Byte (rf[SrcA], 0));
int32_t imagA = signExtend16 (get2Byte (rf[SrcA], 1));
int32_t realB = signExtend16 (get2Byte (rf[SrcB], 0));
int32_t imagB = signExtend16 (get2Byte (rf[SrcB], 1));
int32_t realRes = realA * realB - imagA * imagB + (1 << 14);
int32_t imagRes = realA * imagB + imagA * realB + (1 << 14);
output = set2Byte (output, 0, (realRes >> 15));
output = set2Byte (output, 1, (imagRes >> 15));
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

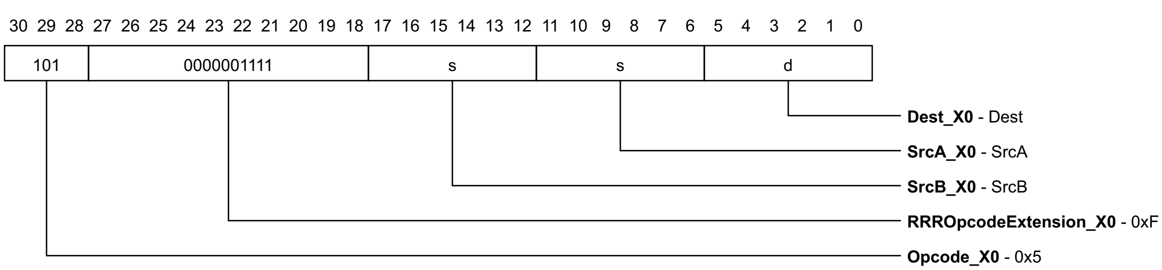

Figure 11-353: cmulfr in X0 Bits Encoding

cmulh

Complex Multiply High Result

Syntax

```
cmulh Dest, SrcA, SrcB
```

Example

```
cmulh r5, r6, r7
```

Description

Multiply the 32-bit complex number in the low half of the first operand by the 32-bit complex number in the low half of the second operand, producing a 32-bit complex result. The 32-bit complex number is represented as a 16-bit signed real value in the lowest-order 16-bits and a 16-bit signed imaginary value in the high-order 16-bits. The 16-bit real and imaginary components of the result are the high-order 16 bits of the full precision multiply result.

Functional Description

```
uint64_t output = 0;
int32_t realA = signExtend16 (get2Byte (rf[SrcA], 0));
int32_t imagA = signExtend16 (get2Byte (rf[SrcA], 1));
int32_t realB = signExtend16 (get2Byte (rf[SrcB], 0));
int32_t imagB = signExtend16 (get2Byte (rf[SrcB], 1));
int32_t realRes = realA * realB - imagA * imagB;
int32_t imagRes = realA * imagB + imagA * realB;
output = set2Byte (output, 0, (realRes >> 16));
output = set2Byte (output, 1, (imagRes >> 16));
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

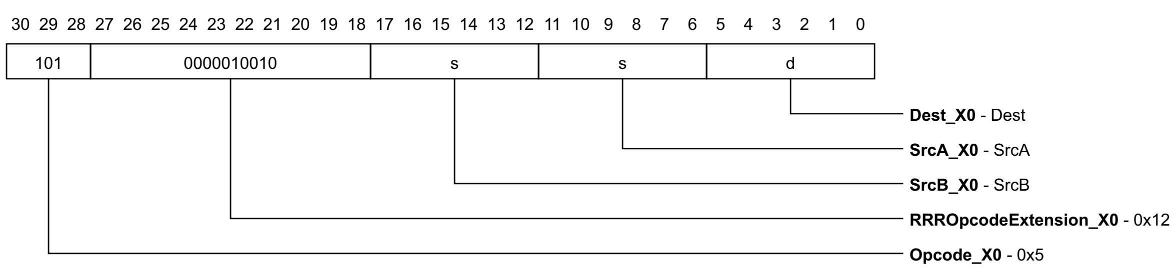

Figure 11-354: cmulh in X0 Bits Encoding

cmulhr

Complex Multiply High Result Round

Syntax

```
cmulhr Dest, SrcA, SrcB
```

Example

```
cmulhr r5, r6, r7
```

Description

Multiply the 32-bit complex number in the low half of the first operand by the 32-bit complex number in the low half of the second operand, producing a rounded 32-bit complex result. The 32-bit complex number is represented as a 16-bit signed real value in the lowest-order 16-bits and a 16-bit signed imaginary value in the high-order 16-bits. The operands are treated as 16-bit signed fractions with the decimal point below the sign bit. The 16-bit real and imaginary components of the result are the high-order 16 bits of the rounded-up full precision multiply result.

Functional Description

```
uint64_t output = 0;
int32_t realA = signExtend16 (get2Byte (rf[SrcA], 0));
int32_t imagA = signExtend16 (get2Byte (rf[SrcA], 1));
int32_t realB = signExtend16 (get2Byte (rf[SrcB], 0));
int32_t imagB = signExtend16 (get2Byte (rf[SrcB], 1));
int32_t realRes = realA * realB - imagA * imagB + (1 << 15);
int32_t imagRes = realA * imagB + imagA * realB + (1 << 15);
output = set2Byte (output, 0, (realRes >> 16));
output = set2Byte (output, 1, (imagRes >> 16));
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

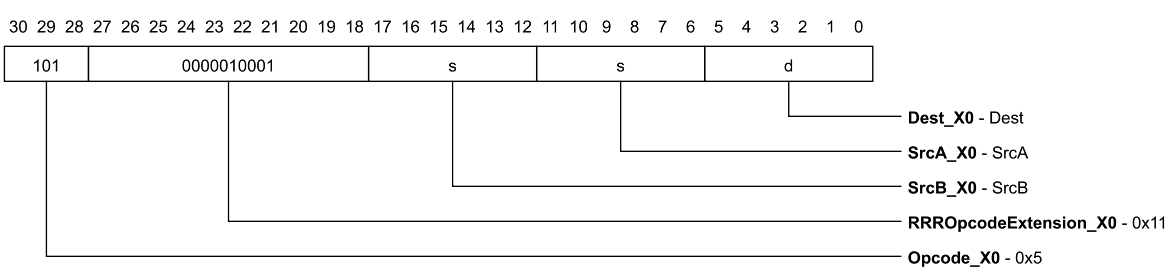

Figure 11-355: cmulhr in X0 Bits Encoding

mul_hs_hs

Multiply High Signed High Signed

Syntax

```
mul_hs_hs Dest, SrcA, SrcB
```

Example

```
mul_hs_hs r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the high 32-bits of the second operand, producing a 64-bit result. The input operands are interpreted as signed values.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] >> 32) * signExtend32 (rf[SrcB] >> 32);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

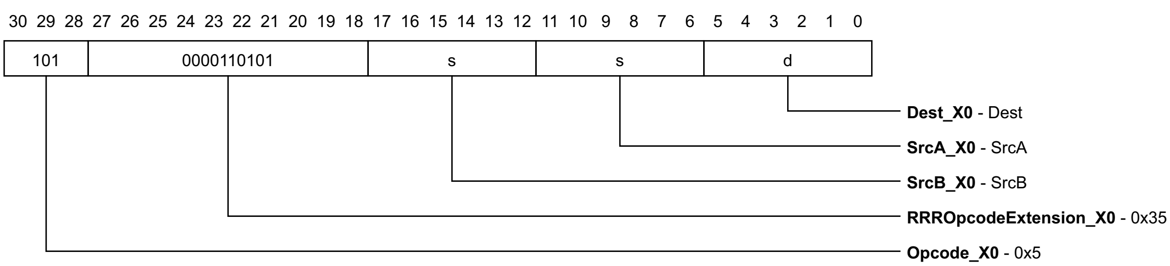

Figure 11-356: mul_hs_hs in X0 Bits Encoding

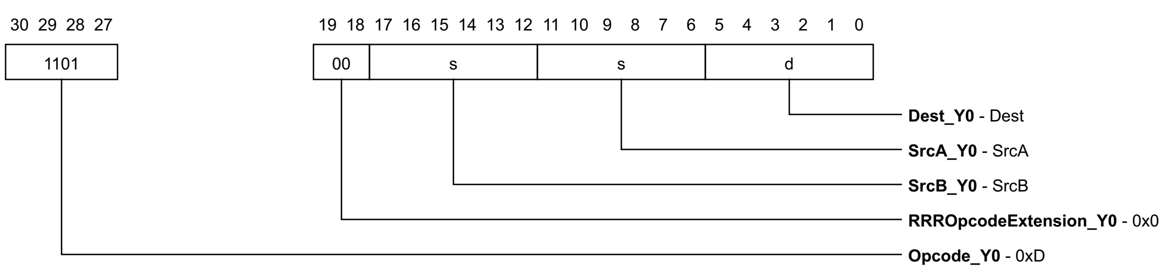

Figure 11-357: mul_hs_hs in Y0 Bits Encoding

mul_hs_hu

Multiply High Signed High Unsigned

Syntax

```
mul_hs_hu Dest, SrcA, SrcB
```

Example

```
mul_hs_hu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the high 32-bits of the second operand, producing a 64-bit result. The first operand is interpreted as a signed value and the second operand is interpreted as an unsigned value.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] >> 32) * ((uint64_t) rf[SrcB] >> 32);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

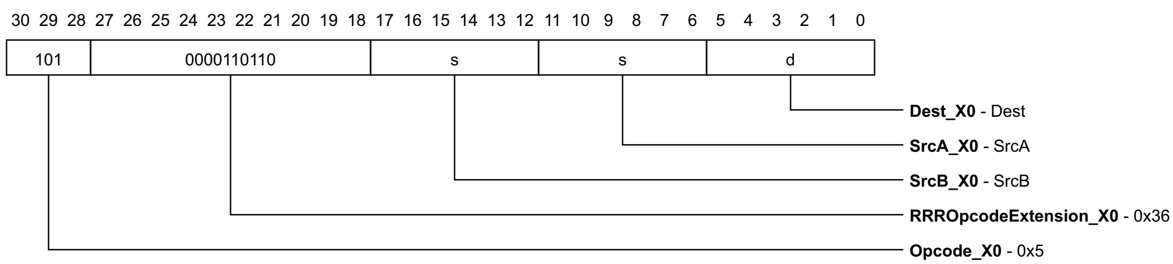

Figure 11-358: mul_hs_hu in X0 Bits Encoding

mul_hs_ls

Multiply High Signed Low Signed

Syntax

```
mul_hs_ls Dest, SrcA, SrcB
```

Example

```
mul_hs_ls r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand, producing a 64-bit result. The input operands are interpreted as signed values.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] >> 32) * signExtend32 (rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

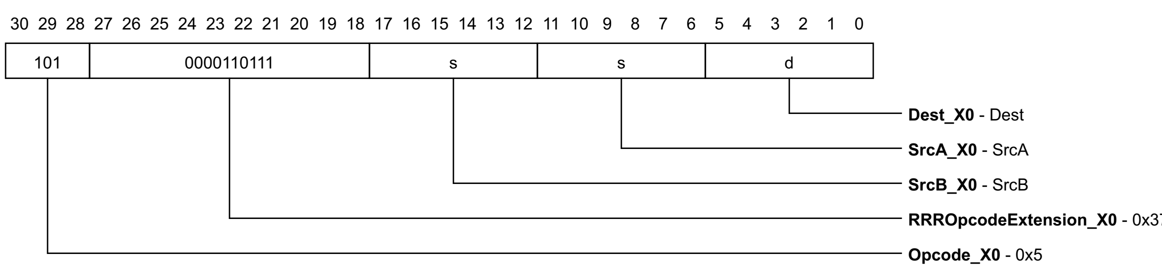

Figure 11-359: mul_hs_ls in X0 Bits Encoding

mul_hs_lu

Multiply High Signed Low Unsigned

Syntax

```
mul_hs_lu Dest, SrcA, SrcB
```

Example

```
mul_hs_lu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand, producing a 64-bit result. The first operand is interpreted as a signed value and the second operand is interpreted as an unsigned value.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA] >> 32) * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

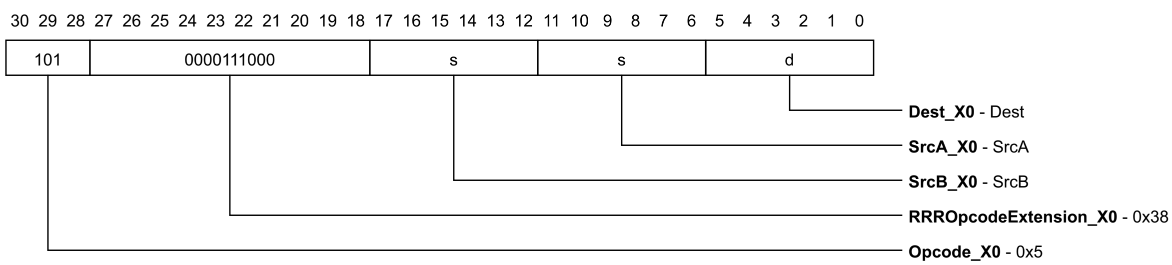

Figure 11-360: mul_hs_lu in X0 Bits Encoding

mul_hu_hu

Multiply High Unsigned High Unsigned

Syntax

```
mul_hu_hu Dest, SrcA, SrcB
```

Example

```
mul_hu_hu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the high 32-bits of the second operand, producing a 64-bit result. The input operands are interpreted as unsigned values.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] >> 32) * ((uint64_t) rf[SrcB] >> 32);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

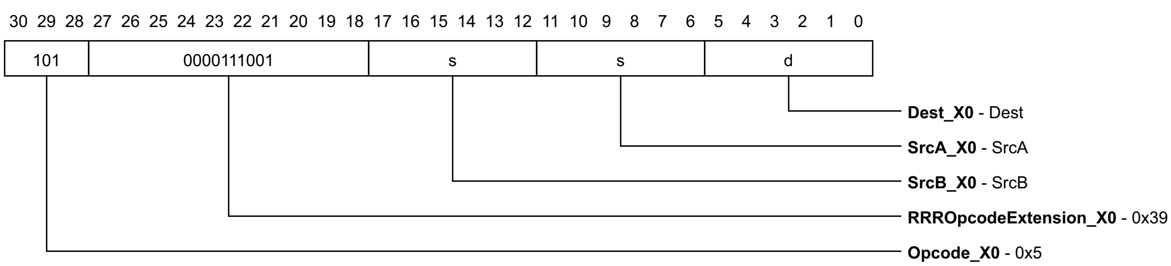

Figure 11-361: mul_hu_hu in X0 Bits Encoding

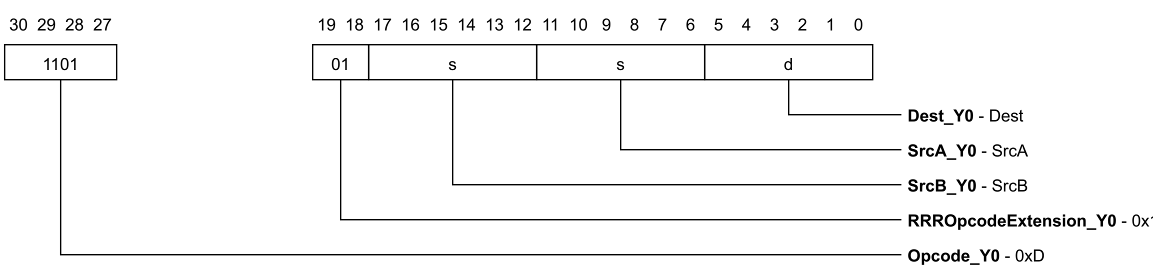

Figure 11-362: mul_hu_hu in Y0 Bits Encoding

mul_hu_ls

Multiply High Unsigned Low Signed

Syntax

```
mul_hu_ls Dest, SrcA, SrcB
```

Example

```
mul_hu_ls r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand, producing a 64-bit result. The first operand is interpreted as an unsigned value and the second operand is interpreted as a signed value.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] >> 32) * signExtend32 (rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

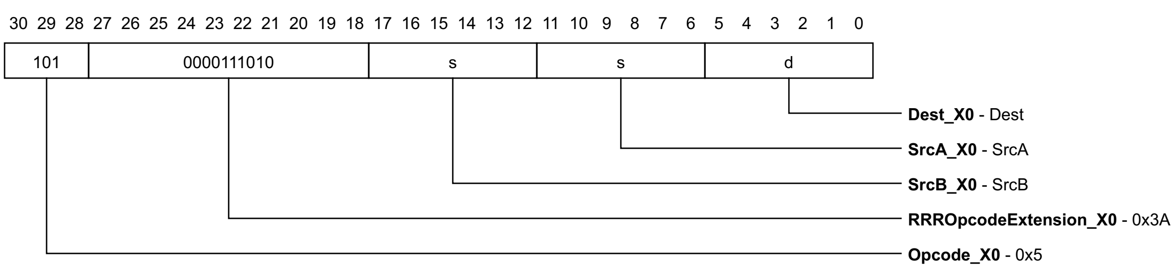

Figure 11-363: mul_hu_ls in X0 Bits Encoding

mul_hu_lu

Multiply High Unsigned Low Unsigned

Syntax

```
mul_hu_lu Dest, SrcA, SrcB
```

Example

```
mul_hu_lu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand, producing a 64-bit result. The input operands are interpreted as unsigned values.

Functional Description

```
rf[Dest] = ((uint64_t) rf[SrcA] >> 32) * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

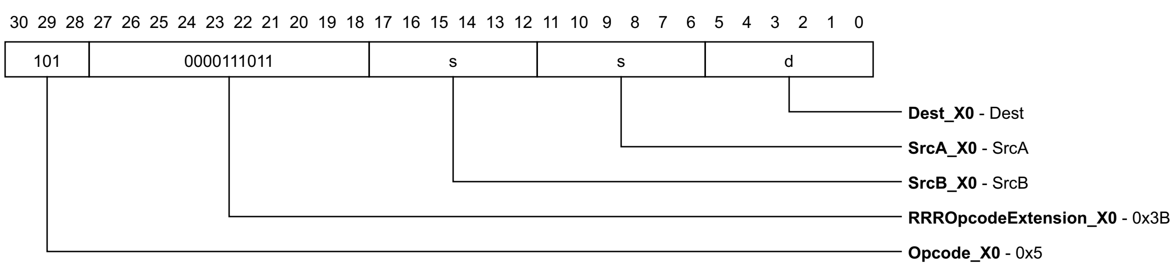

Figure 11-364: mul_hu_lu in X0 Bits Encoding

mul_ls_ls

Multiply Low Signed Low Signed

Syntax

```
mul_ls_ls Dest, SrcA, SrcB
```

Example

```
mul_ls_ls r5, r6, r7
```

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand, producing a 64-bit result. The input operands are interpreted as signed values.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA]) * signExtend32 (rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

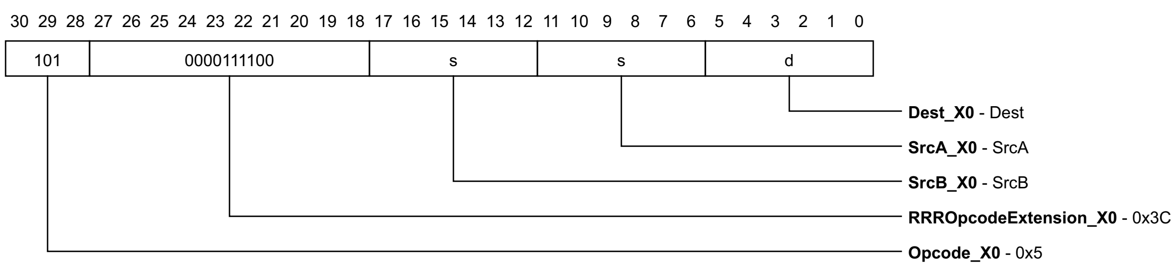

Figure 11-365: mul_ls_ls in X0 Bits Encoding

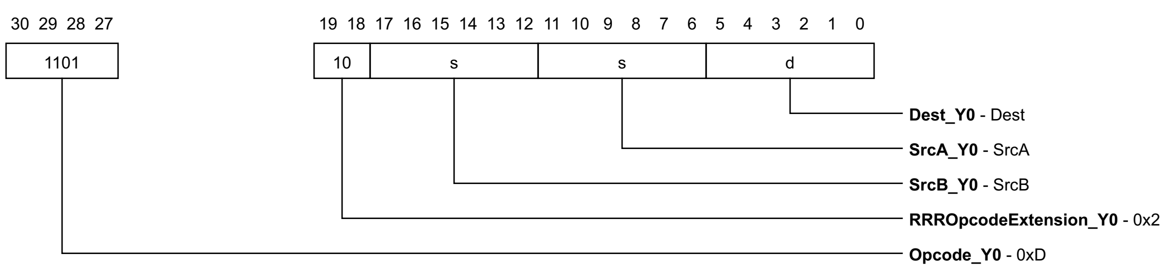

Figure 11-366: mul_ls_ls in Y0 Bits Encoding

mul_ls_lu

Multiply Low Signed Low Unsigned

Syntax

`mul_ls_lu Dest, SrcA, SrcB`

Example

`mul_ls_lu r5, r6, r7`

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand, producing a 64-bit result. The first operand is interpreted as a signed value and the second operand is interpreted as an unsigned value.

Functional Description

```
rf[Dest] = signExtend32 (rf[SrcA]) * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

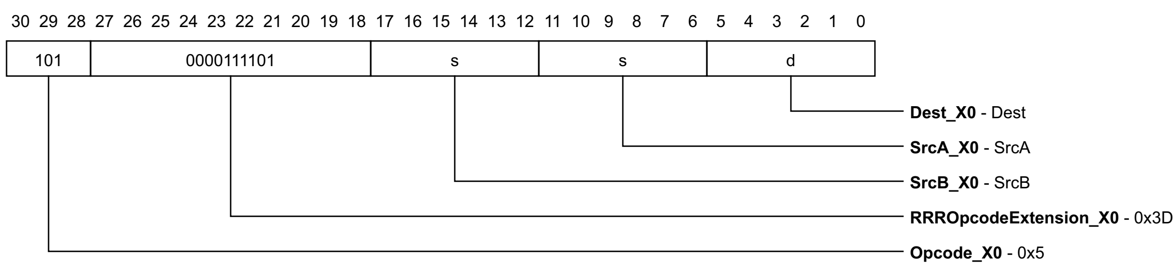

Figure 11-367: mul_ls_lu in X0 Bits Encoding

mul_lu_lu

Multiply Low Unsigned Low Unsigned

Syntax

```
mul_lu_lu Dest, SrcA, SrcB
```

Example

```
mul_lu_lu r5, r6, r7
```

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand, producing a 64-bit result. The input operands are interpreted as unsigned values.

Functional Description

```
rf[Dest] = (uint64_t) (uint32_t) rf[SrcA] * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

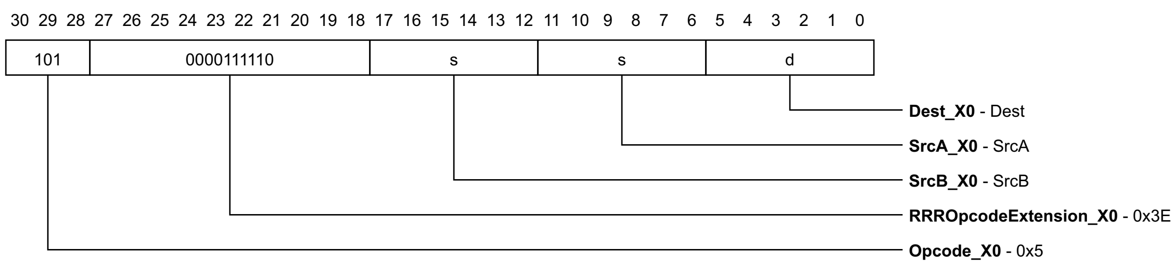

Figure 11-368: mul_lu_lu in X0 Bits Encoding

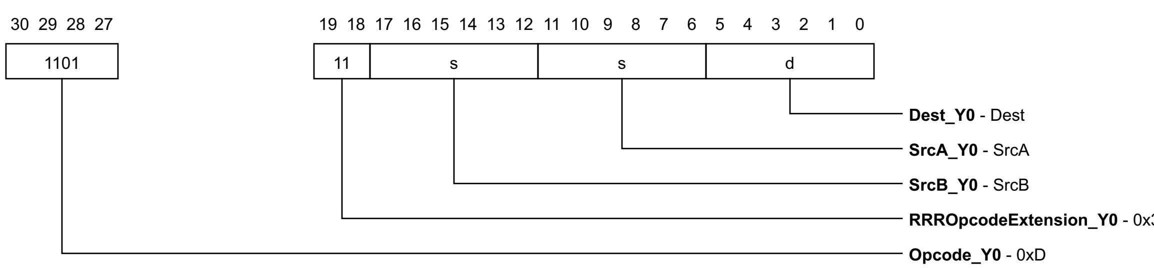

Figure 11-369: mul_lu_lu in Y0 Bits Encoding

mula_hs_hs

Multiply Accumulate High Signed High Signed

Syntax

```
mula_hs_hs Dest, SrcA, SrcB
```

Example

```
mula_hs_hs r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the high 32-bits of the second operand and accumulate the result into the destination operand. The input operands are interpreted as signed values.

Functional Description

```
rf[Dest] =
  rf[Dest] + signExtend32 (rf[SrcA] >> 32) * signExtend32 (rf[SrcB] >> 32);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

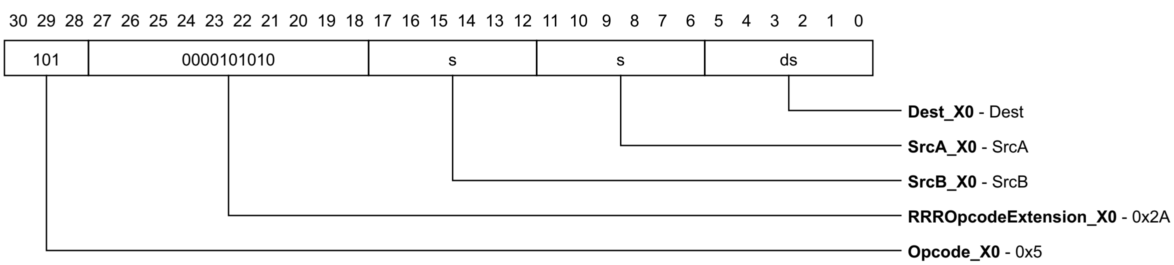

Figure 11-370: mula_hs_hs in X0 Bits Encoding

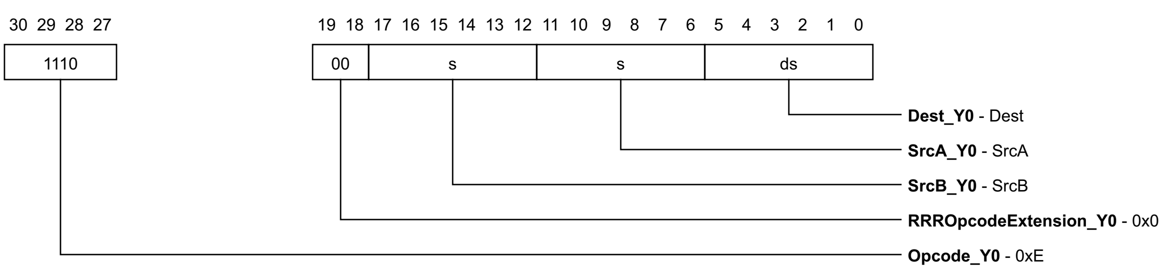

Figure 11-371: mula_hs_hs in Y0 Bits Encoding

mula_hs_hu

Multiply Accumulate High Signed High Unsigned

Syntax

```
mula_hs_hu Dest, SrcA, SrcB
```

Example

```
mula_hs_hu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the high 32-bits of the second operand and accumulate the result into the destination operand. The first operand is interpreted as a signed value and the second operand is interpreted as an unsigned value.

Functional Description

```
rf[Dest] =
  rf[Dest] + signExtend32 (rf[SrcA] >> 32) * ((uint64_t) rf[SrcB] >> 32);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

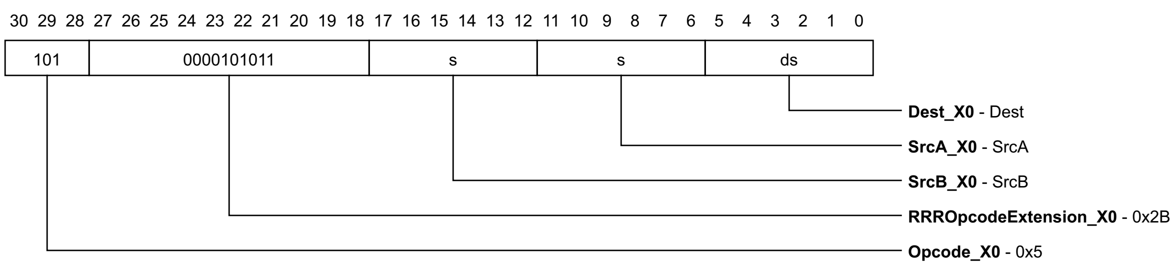

Figure 11-372: mula_hs_hu in X0 Bits Encoding

mula_hs_ls

Multiply Accumulate High Signed Low Signed

Syntax

```
mula_hs_ls Dest, SrcA, SrcB
```

Example

```
mula_hs_ls r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand and accumulate the result into the destination operand. The input operands are interpreted as signed values.

Functional Description

```
rf[Dest] = rf[Dest] + signExtend32 (rf[SrcA] >> 32) * signExtend32 (rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

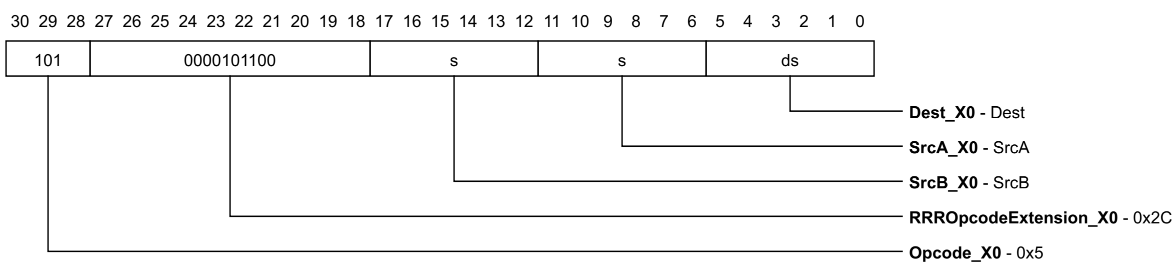

Figure 11-373: mula_hs_ls in X0 Bits Encoding

mula_hs_lu

Multiply Accumulate High Signed Low Unsigned

Syntax

```
mula_hs_lu Dest, SrcA, SrcB
```

Example

```
mula_hs_lu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand and accumulate the result into the destination operand. The first operand is interpreted as a signed value and the second operand is interpreted as an unsigned value.

Functional Description

```
rf[Dest] =
  rf[Dest] + signExtend32 (rf[SrcA] >> 32) * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

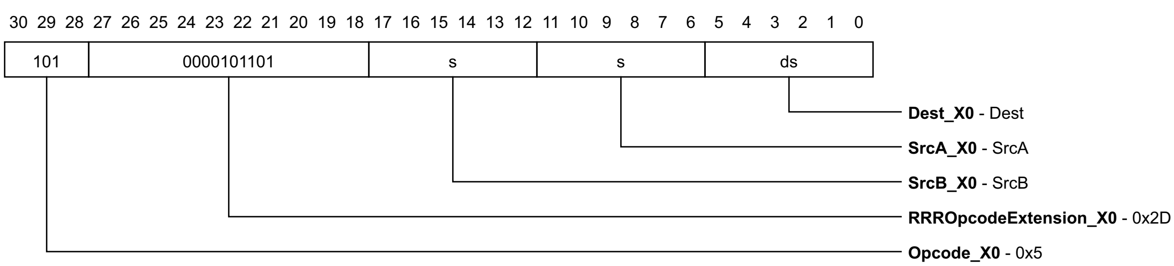

Figure 11-374: mula_hs_lu in X0 Bits Encoding

mula_hu_hu

Multiply Accumulate High Unsigned High Unsigned

Syntax

```
mula_hu_hu Dest, SrcA, SrcB
```

Example

```
mula_hu_hu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the high 32-bits of the second operand and accumulate the result into the destination operand. The input operands are interpreted as unsigned values.

Functional Description

```
rf[Dest] =
  rf[Dest] + ((uint64_t) rf[SrcA] >> 32) * ((uint64_t) rf[SrcB] >> 32);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

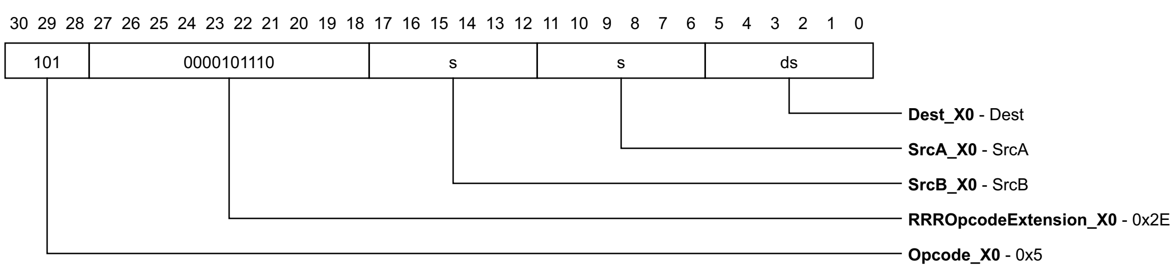

Figure 11-375: mula_hu_hu in X0 Bits Encoding

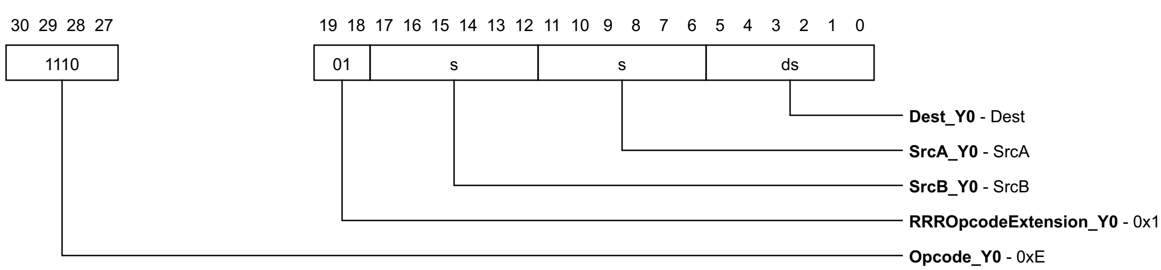

Figure 11-376: mula_hu_hu in Y0 Bits Encoding

mula_hu_ls

Multiply Accumulate High Unsigned Low Signed

Syntax

```
mula_hu_ls Dest, SrcA, SrcB
```

Example

```
mula_hu_ls r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand and accumulate the result into the destination operand. The first operand is interpreted as an unsigned value and the second operand is interpreted as a signed value.

Functional Description

```
rf[Dest] = rf[Dest] + ((uint64_t) rf[SrcA] >> 32) * signExtend32 (rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

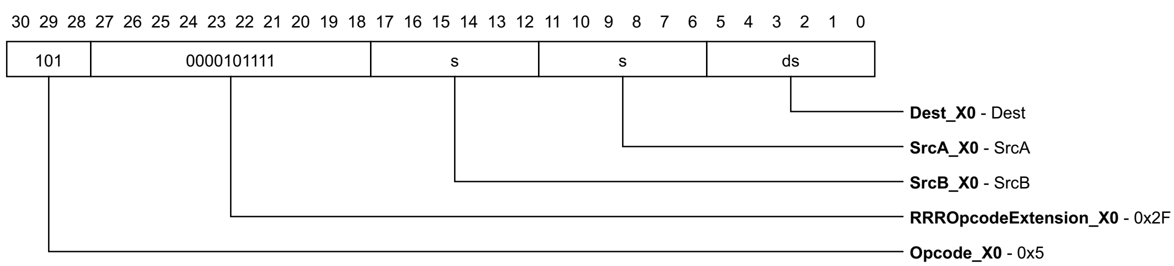

Figure 11-377: mula_hu_ls in X0 Bits Encoding

mula_hu_lu

Multiply Accumulate High Unsigned Low Unsigned

Syntax

```
mula_hu_lu Dest, SrcA, SrcB
```

Example

```
mula_hu_lu r5, r6, r7
```

Description

Multiply the high 32 bits of the first operand by the low 32-bits of the second operand and accumulate the result into the destination operand. The input operands are interpreted as unsigned values.

Functional Description

```
rf[Dest] =
  rf[Dest] + ((uint64_t) rf[SrcA] >> 32) * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

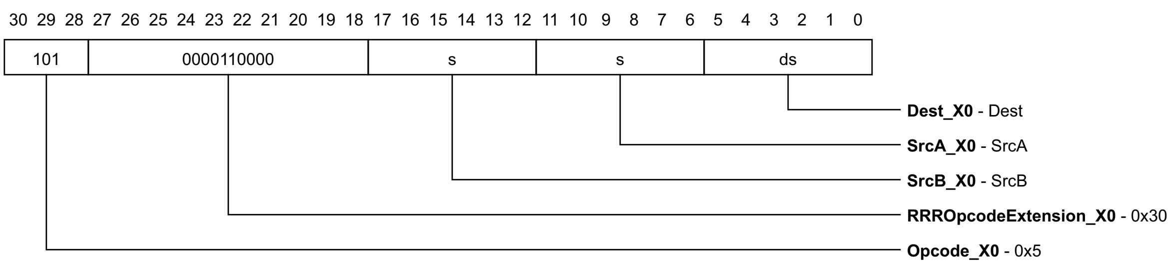

Figure 11-378: mula_hu_lu in X0 Bits Encoding

mula_ls_ls

Multiply Accumulate Low Signed Low Signed

Syntax

```
mula_ls_ls Dest, SrcA, SrcB
```

Example

```
mula_ls_ls r5, r6, r7
```

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand and accumulate the result into the destination operand. The input operands are interpreted as signed values.

Functional Description

```
rf[Dest] = rf[Dest] + signExtend32 (rf[SrcA]) * signExtend32 (rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

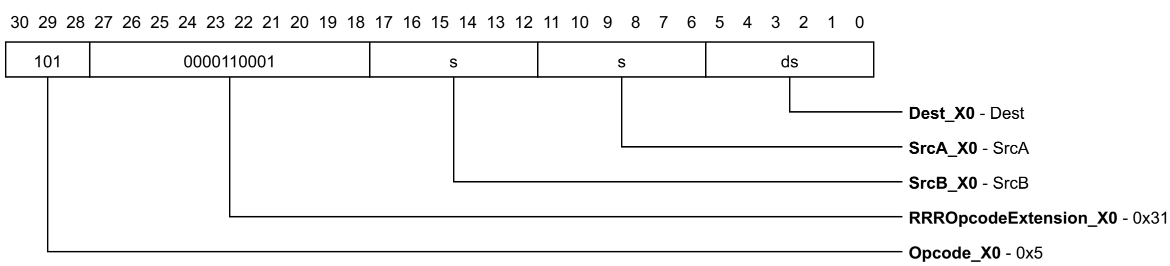

Figure 11-379: mula_ls_ls in X0 Bits Encoding

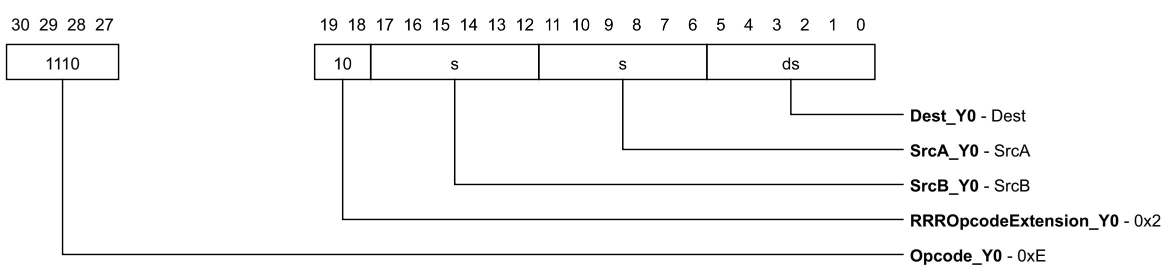

Figure 11-380: mula_ls_ls in Y0 Bits Encoding

mula_ls_lu

Multiply Accumulate Low Signed Low Unsigned

Syntax

```
mula_ls_lu Dest, SrcA, SrcB
```

Example

```
mula_ls_lu r5, r6, r7
```

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand and accumulate the result into the destination operand. The first operand is interpreted as a signed value and the second operand is interpreted as an unsigned value.

Functional Description

```
rf[Dest] =
  rf[Dest] + signExtend32 (rf[SrcA]) * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

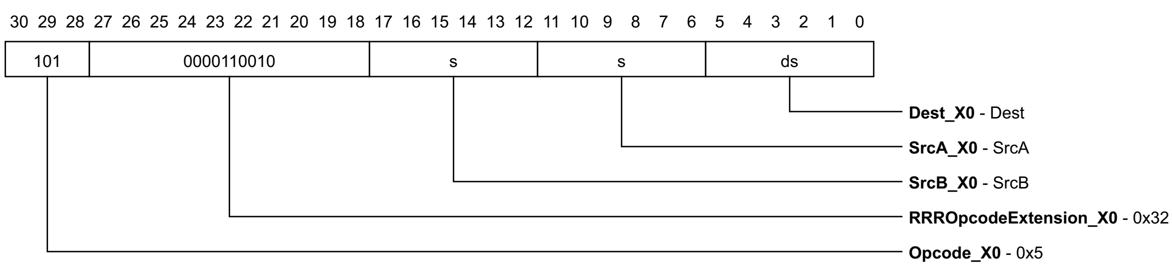

Figure 11-381: mula_ls_lu in X0 Bits Encoding

mula_lu_lu

Multiply Accumulate Low Unsigned Low Unsigned

Syntax

```
mula_lu_lu Dest, SrcA, SrcB
```

Example

```
mula_lu_lu r5, r6, r7
```

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand and accumulate the result into the destination operand. The input operands are interpreted as unsigned values.

Functional Description

```
rf[Dest] =
  rf[Dest] + (uint64_t) (uint32_t) rf[SrcA] * (uint64_t) (uint32_t) rf[SrcB];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

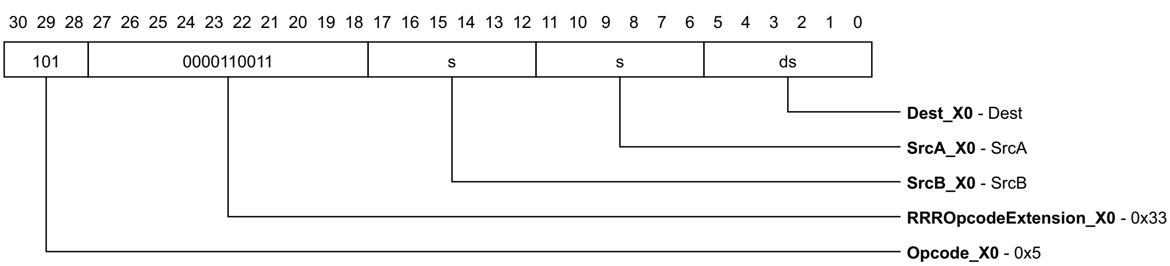

Figure 11-382: mula_lu_lu in X0 Bits Encoding

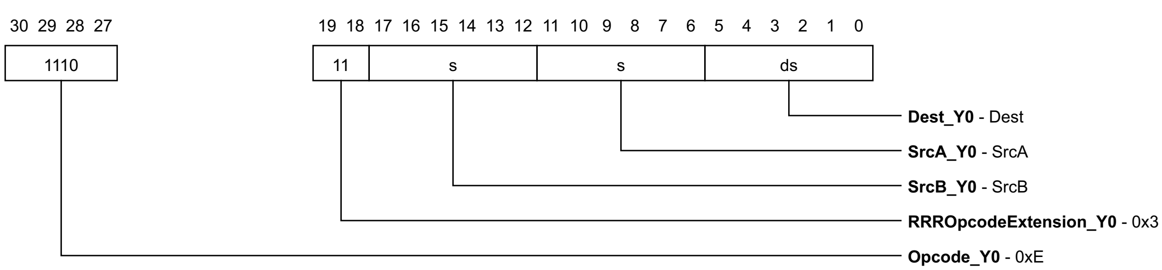

Figure 11-383: mula_lu_lu in Y0 Bits Encoding

mulax

Multiply Accumulate and Extend

Syntax

```
mulax Dest, SrcA, SrcB
```

Example

```
mulax r5, r6, r7
```

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand, and add to the destination operand producing a 32-bit result that is sign-extended.

Functional Description

```
rf[Dest] =
signExtend32 ((int32_t) rf[Dest] + (int32_t) rf[SrcA] * (int32_t) rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

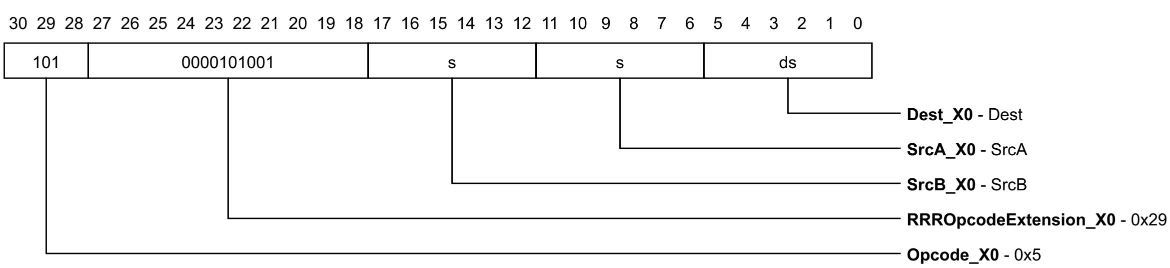

Figure 11-384: mulax in X0 Bits Encoding

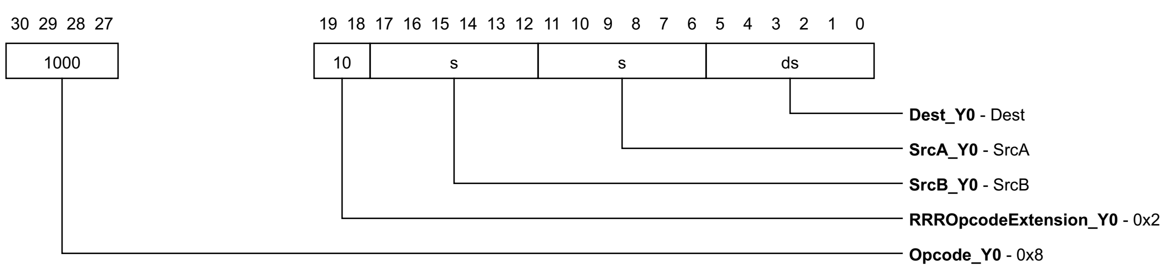

Figure 11-385: mulax in Y0 Bits Encoding

mulx

Multiply and Extend

Syntax

```
mulx Dest, SrcA, SrcB
```

Example

```
mulx r5, r6, r7
```

Description

Multiply the low 32 bits of the first operand by the low 32-bits of the second operand, producing a 32-bit result, which is sign-extended.

Functional Description

```
rf[Dest] = signExtend32 ((int32_t) rf[SrcA] * (int32_t) rf[SrcB]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

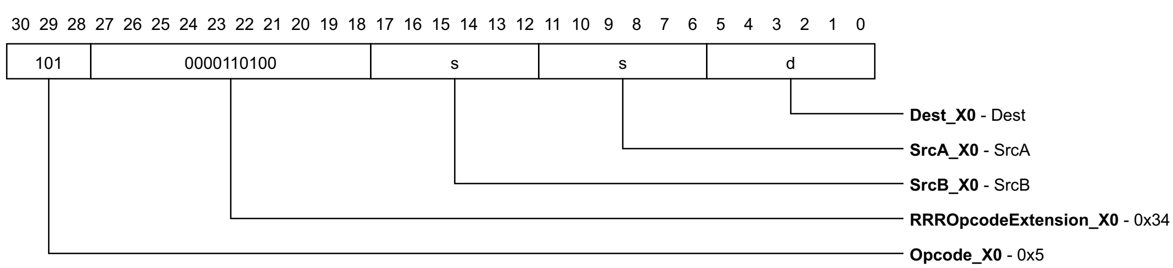

Figure 11-386: mulx in X0 Bits Encoding

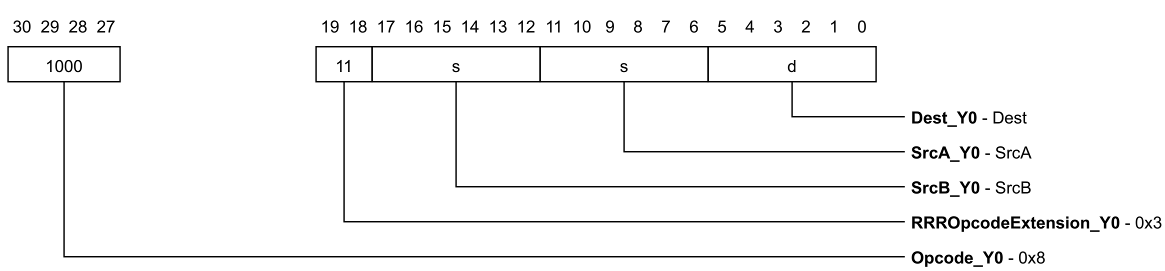

Figure 11-387: mulx in Y0 Bits Encoding

CHAPTER 12 NOP INSTRUCTIONS

12.1 Overview

The following sections provide detailed descriptions of `nop` instructions listed alphabetically.

12.2 Instructions

NOP instructions are described in the sections that follow.

fnop

Filler No Operation

Syntax

`fnop`

Example

`fnop`

Description

Indicate that the programmer, compiler, or tool was not able to fill this operation slot with a suitable operation. This operation has no outcome. The `fnop` instruction should be used to signal that the no operation is inserted because nothing else could be packed into the instruction bundle, not because an architectural `nop` is needed for correct operation or for timing delay. Typically, `fnop`s can be removed at any point in the tool flow.

Functional Description

```
fnop ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

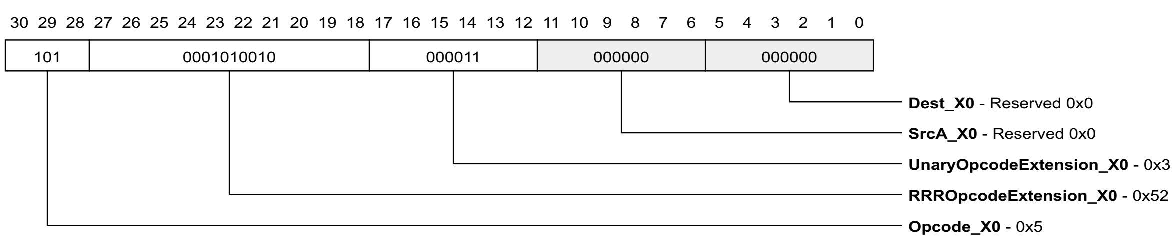

Figure 12-388: fnop in X0 Bits Encoding

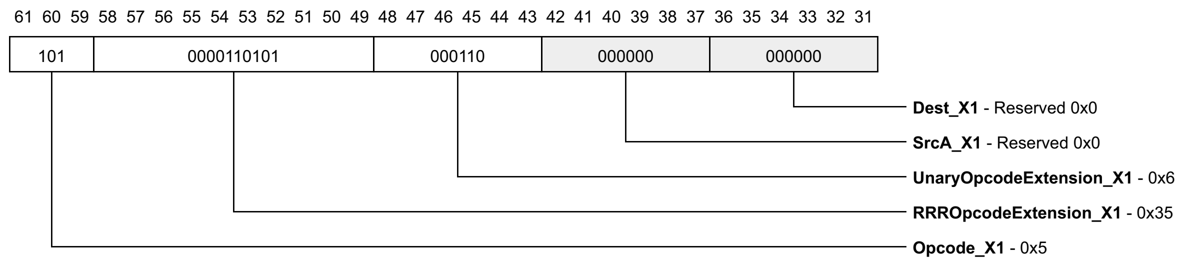

Figure 12-389: fnop in X1 Bits Encoding

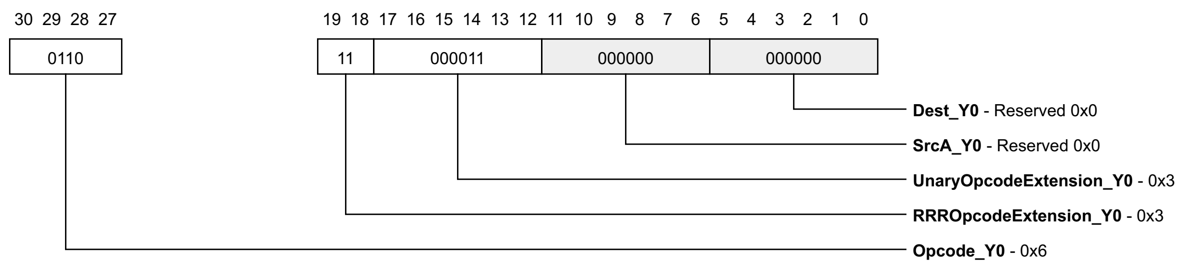

Figure 12-390: fnop in Y0 Bits Encoding

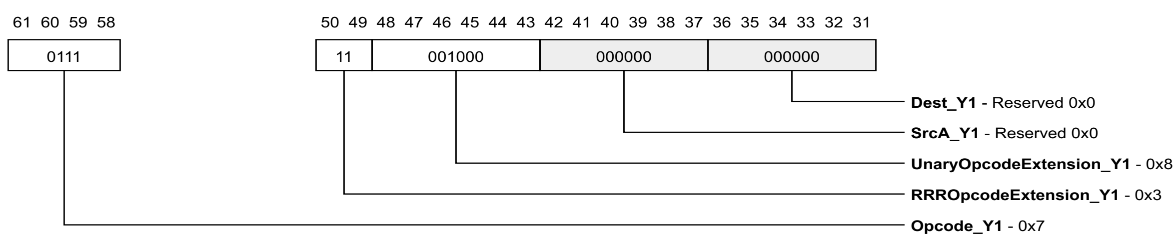

Figure 12-391: fnop in Y1 Bits Encoding

nop

Architectural No Operation

Syntax

```
nop
```

Example

```
nop
```

Description

Indicate to the hardware architecture that the machine should not issue an instruction with a side effect in this slot.

Functional Description

```
nop ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

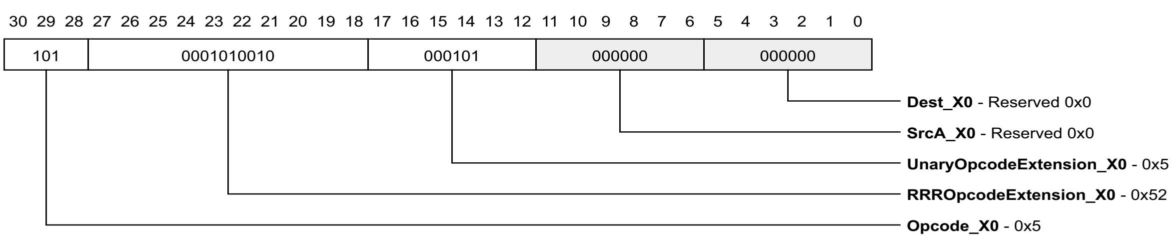

Figure 12-392: nop in X0 Bits Encoding

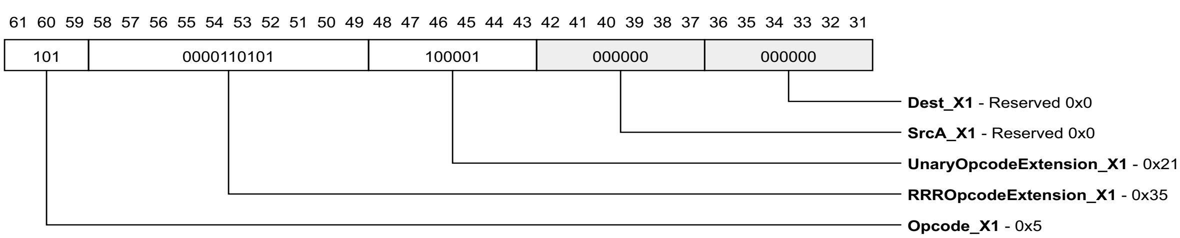

Figure 12-393: nop in X1 Bits Encoding

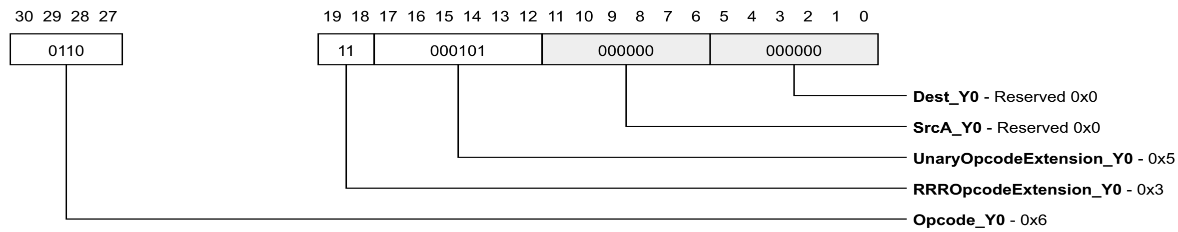

Figure 12-394: nop in Y0 Bits Encoding

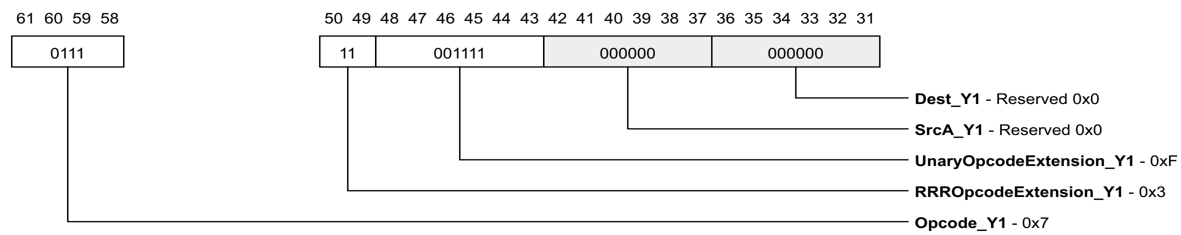

Figure 12-395: nop in Y1 Bits Encoding

CHAPTER 13 PSEUDO INSTRUCTIONS

13.1 Overview

The following sections provide detailed descriptions of pseudo instructions listed alphabetically.

13.2 Instructions

The following sections provide detailed descriptions of pseudo instructions listed alphabetically.

bpt

Breakpoint

Syntax

`bpt`

Example

`bpt`

Description

Causes an illegal instruction interrupt to occur. The Illegal Instruction is guaranteed to always cause an illegal instruction interrupt for all current and future derivations of the architecture. The runtime can use the `BreakpointDistinguisher` fields to characterize the fault as due to a breakpoint.

Functional Description

```
illegalInstruction ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

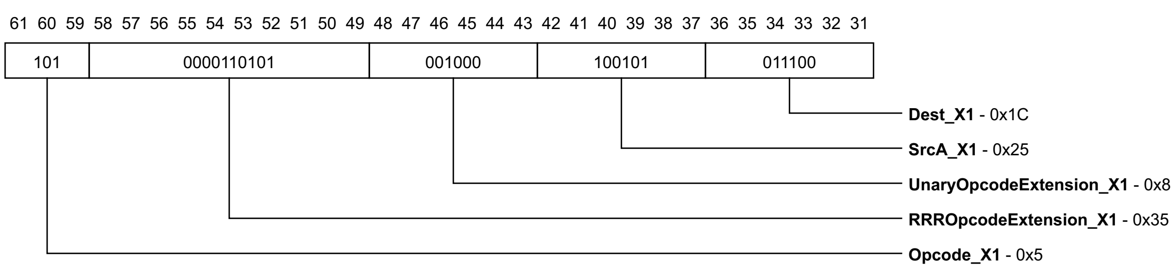

Figure 13-396: bpt in X1 Bits Encoding

info

Informational Note

Syntax

```
info Imm8
```

Example

```
info 19
```

Description

A no-op whose presence provides information to the backtracer and other tools.

Functional Description

```
nop ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

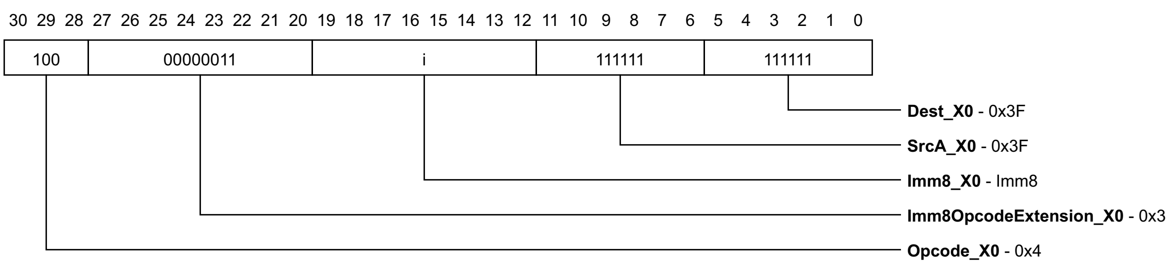

Figure 13-397: info in X0 Bits Encoding

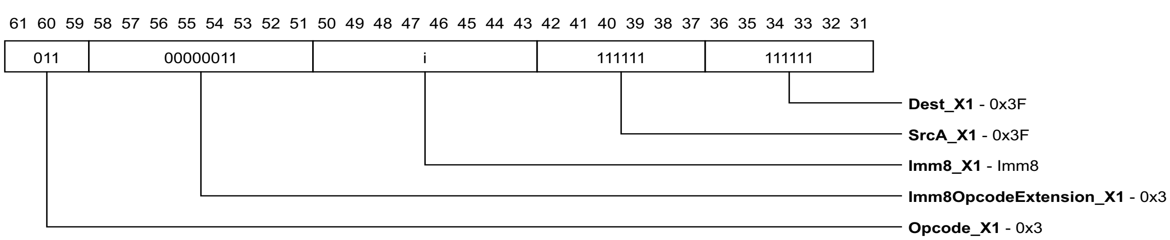

Figure 13-398: info in X1 Bits Encoding

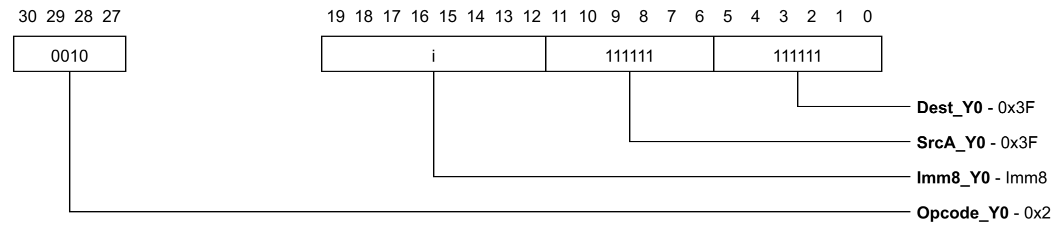

Figure 13-399: info in Y0 Bits Encoding

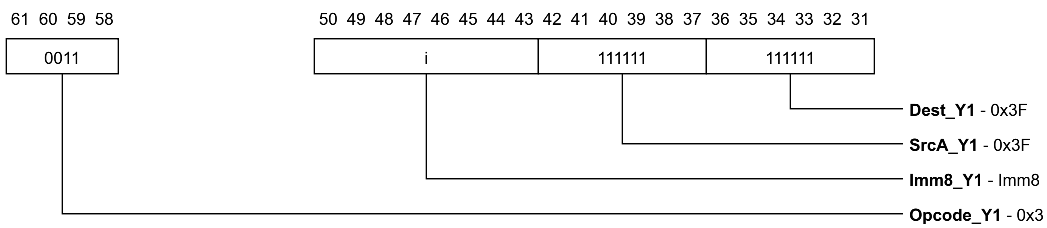

Figure 13-400: info in Y1 Bits Encoding

infol

Long Informational Note

Syntax

```
infol Imm16
```

Example

```
infol 0x1234
```

Description

A no-op whose presence provides information to the backtracer and other tools.

Functional Description

```
nop ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

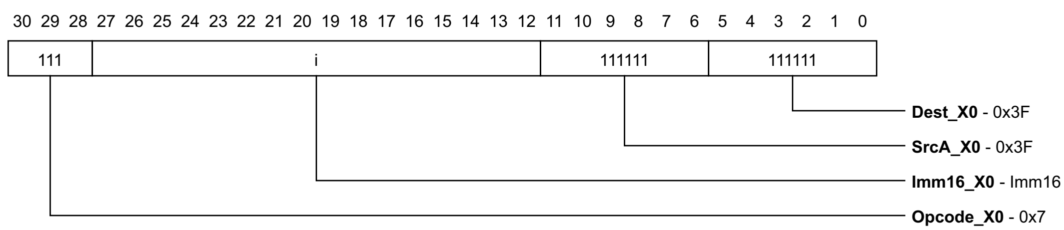

Figure 13-401: infol in X0 Bits Encoding

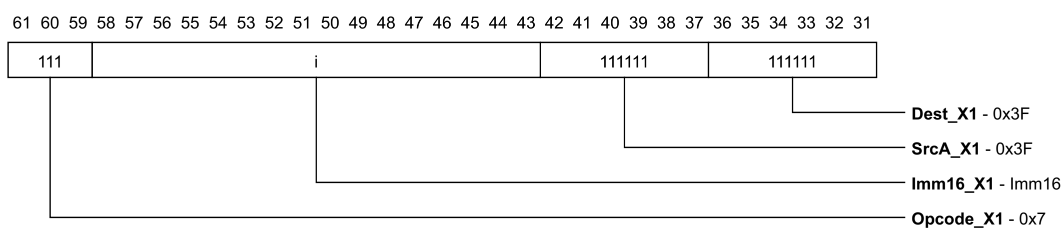

Figure 13-402: infol in X1 Bits Encoding

ld4s_tls

Load Four Bytes Signed TLS

Syntax

```
ld4s_tls Dest, SrcA, Imm8
```

Example

```
ld4s_tls r5, r6, 0
```

Description

Do a four-byte signed TLS IE load, or the corresponding LE action.

Functional Description

```
rf[Dest] = signExtend32 (memoryReadWord (rf[SrcA]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

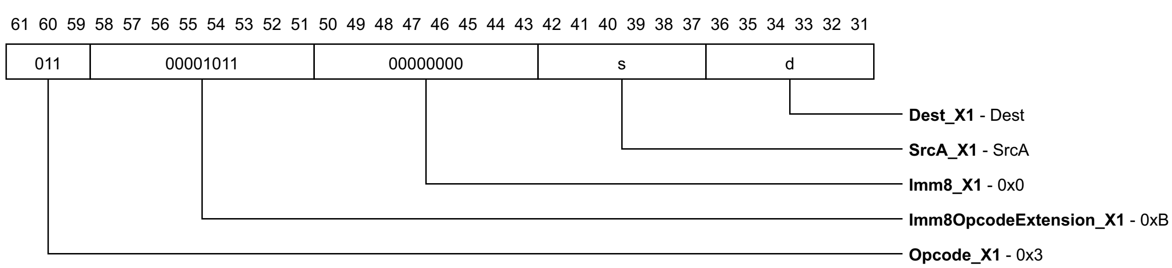

Figure 13-403: ld4s_tls in X1 Bits Encoding

ld_tls

Load TLS

Syntax

```
ld_tls Dest, SrcA, Imm8
```

Example

```
ld_tls r5, r6, 0
```

Description

Do a TLS IE load, or the corresponding LE action.

Functional Description

```
rf[Dest] = memoryReadDoubleWord (rf[SrcA]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

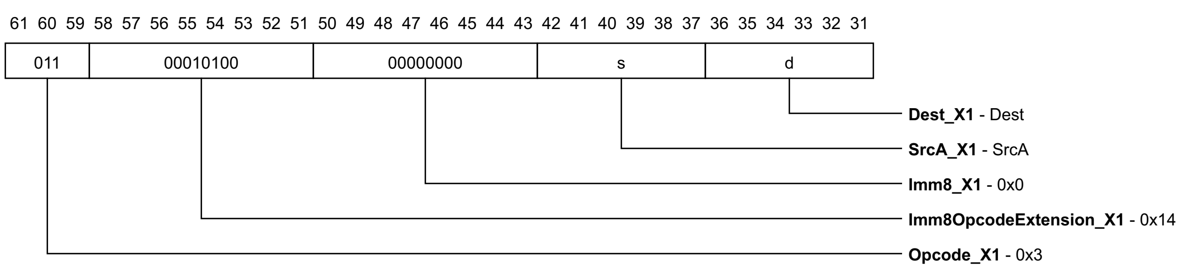

Figure 13-404: ld_tls in X1 Bits Encoding

move

Move

Syntax

```
move Dest, SrcA
```

Example

```
move r5, r6
```

Description

Moves one operand to another operand.

Functional Description

```
rf[Dest] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

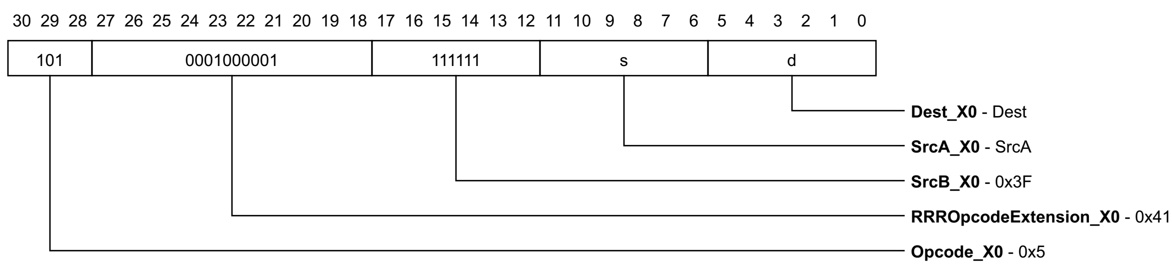

Figure 13-405: move in X0 Bits Encoding

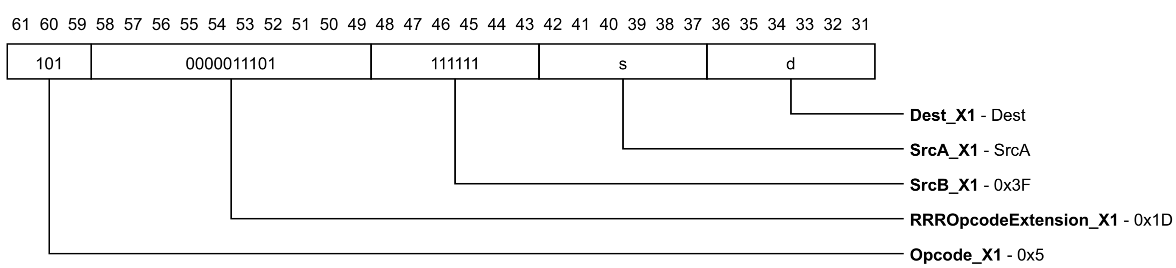

Figure 13-406: move in X1 Bits Encoding

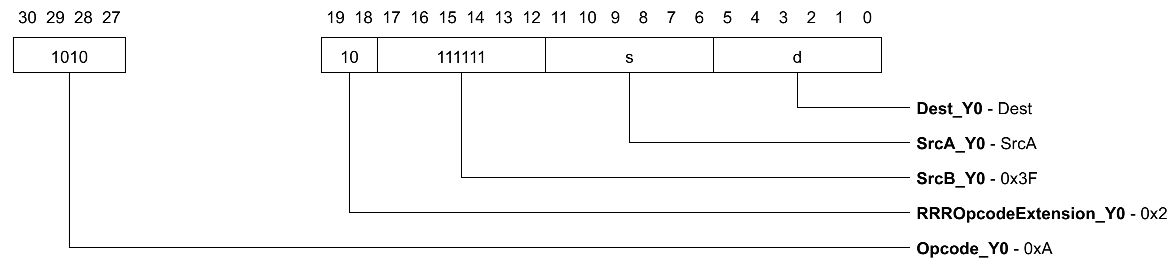

Figure 13-407: move in Y0 Bits Encoding

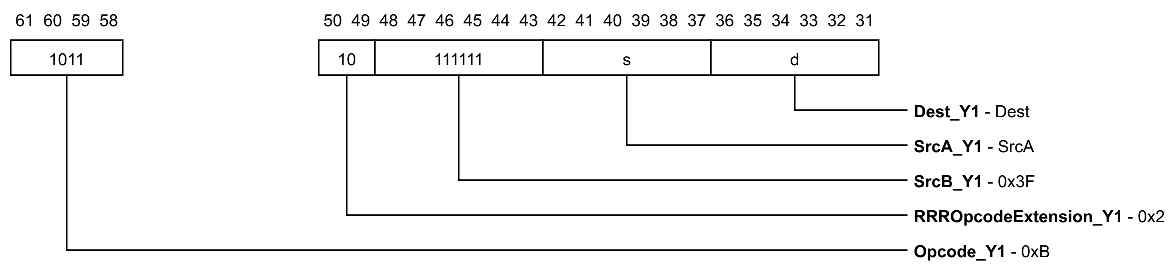

Figure 13-408: move in Y1 Bits Encoding

movei

Move Immediate Word

Syntax

```
movei Dest, Imm8
```

Example

```
movei r5, 5
```

Description

Moves a sign extended 8-bit immediate into a register.

Functional Description

```
rf[Dest] = signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

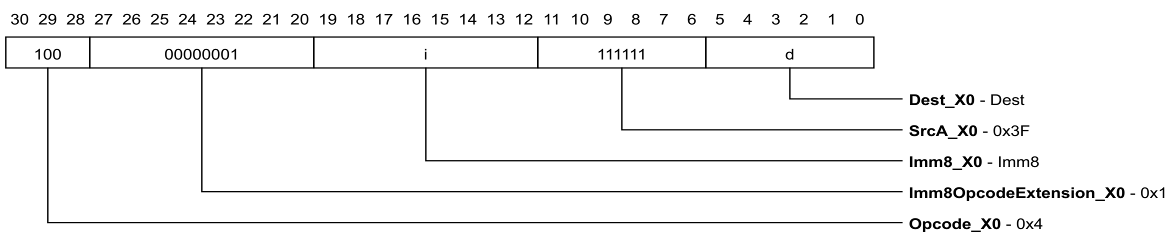

Figure 13-409: movei in X0 Bits Encoding

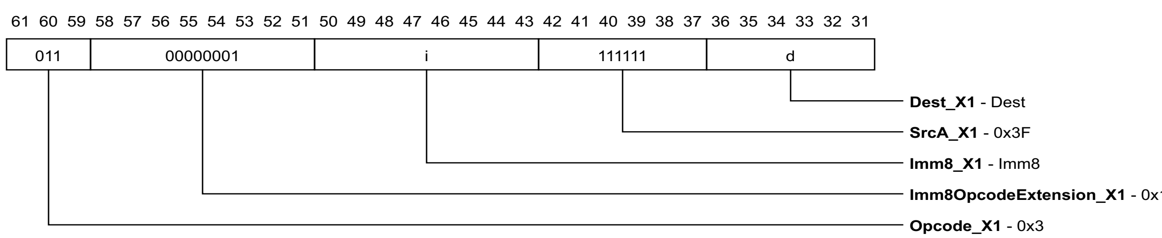

Figure 13-410: movei in X1 Bits Encoding

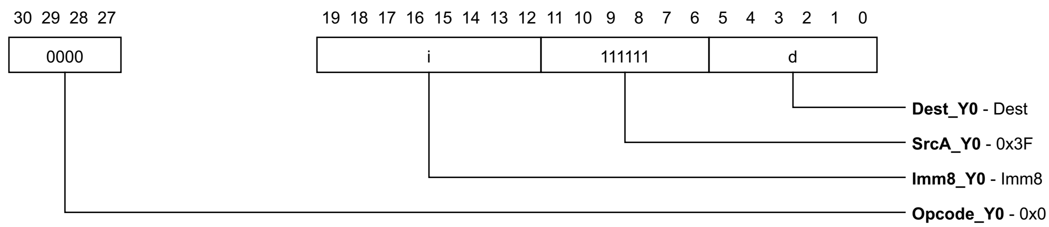

Figure 13-411: movei in Y0 Bits Encoding

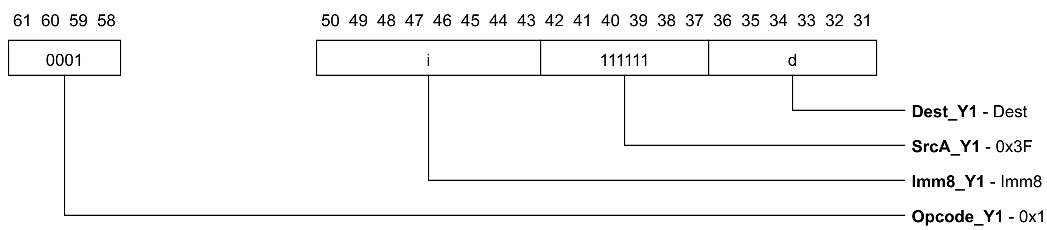

Figure 13-412: movei in Y1 Bits Encoding

moveli

Move Long Immediate Word

Syntax

```
moveli Dest, Imm16
```

Example

```
moveli r5, 0x1234
```

Description

Moves a sign extended long immediate into a register.

Functional Description

```
rf[Dest] = signExtend16 (Imm16);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

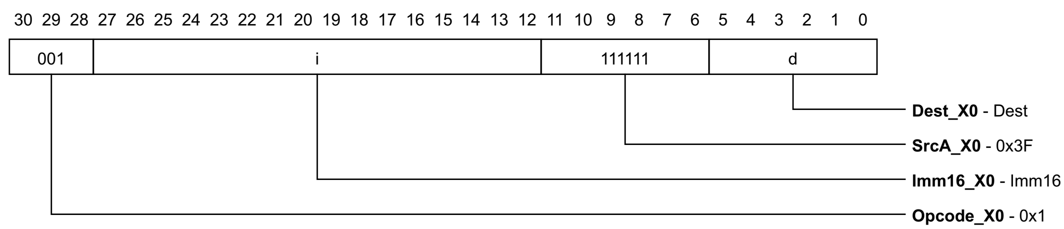

Figure 13-413: moveli in X0 Bits Encoding

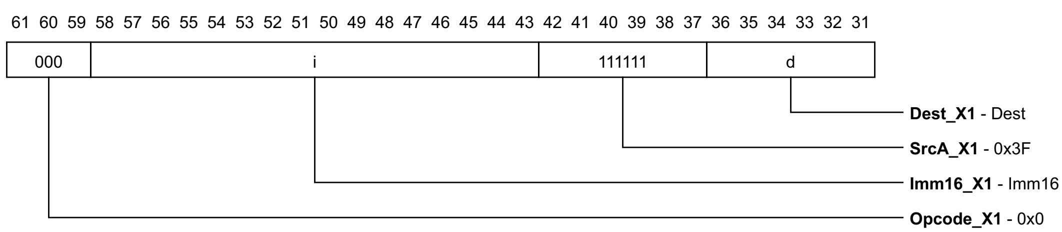

Figure 13-414: moveli in X1 Bits Encoding

prefetch

Prefetch to L1 with No Faults

Syntax

```
prefetch Src
```

Example

```
prefetch r5
```

Description

Prefetch a cache line from memory into all levels of cache without signalling any address or access interrupts. The address to be prefetched is read from the source operand.

Functional Description

```
memoryPrefetch (rf[Src], 1, false);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

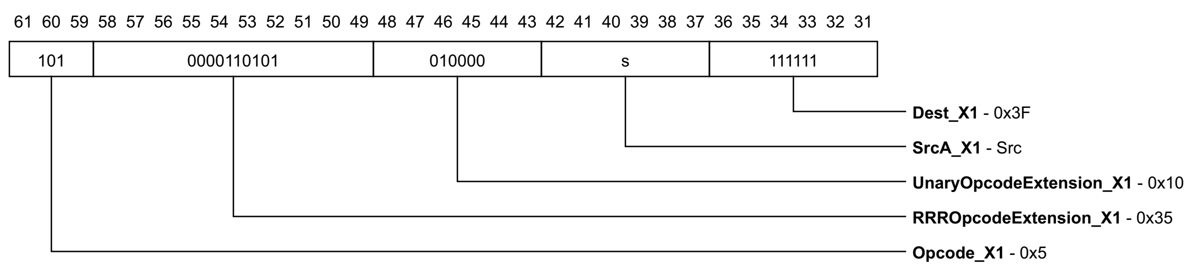

Figure 13-415: prefetch in X1 Bits Encoding

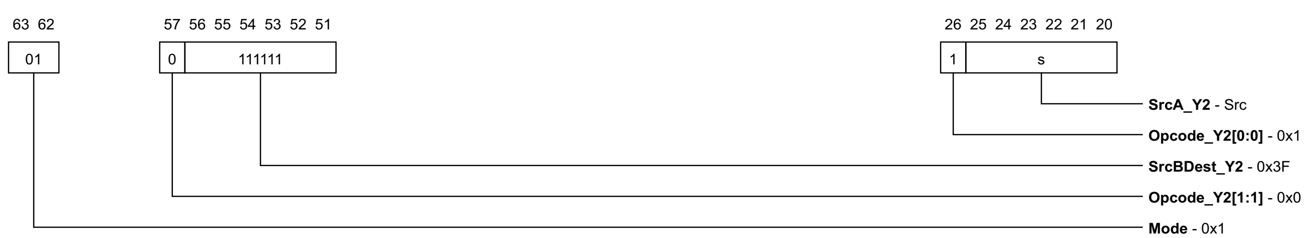

Figure 13-416: prefetch in Y2 Bits Encoding

prefetch_add_l1

Prefetch to L1 and Add with No Faults

Syntax

```
prefetch_add_l1 SrcA, Imm8
```

Example

```
prefetch_add_l1 r5, 5
```

Description

Prefetch a cache line from memory into all levels of cache without signalling of address or access interrupts. The address to be prefetched is read from the source operand. Add the signed immediate argument to the address register.

Functional Description

```
memoryPrefetch (rf[SrcA], 1, false);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

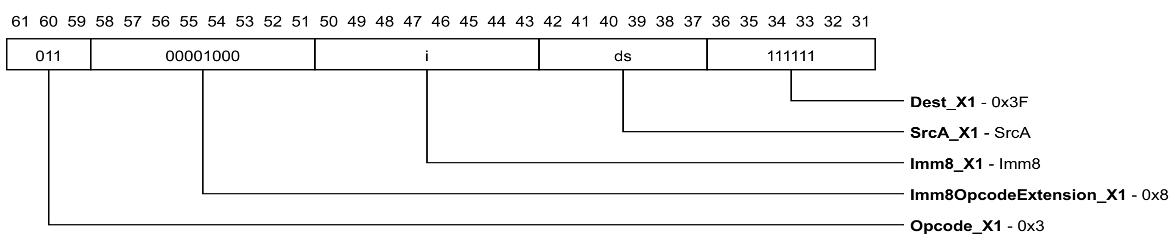

Figure 13-417: prefetch_add_l1 in X1 Bits Encoding

prefetch_add_l1_fault

Prefetch to L1 and Add with Faults

Syntax

```
prefetch_add_l1_fault SrcA, Imm8
```

Example

```
prefetch_add_l1_fault r5, 5
```

Description

Prefetch a cache line from memory into all levels of cache with signalling of address or access interrupts. The address to be prefetched is read from the source operand. Add the signed immediate argument to the address register even if an interrupt is signalled.

Functional Description

```
memoryPrefetch (rf[SrcA], 1, true);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

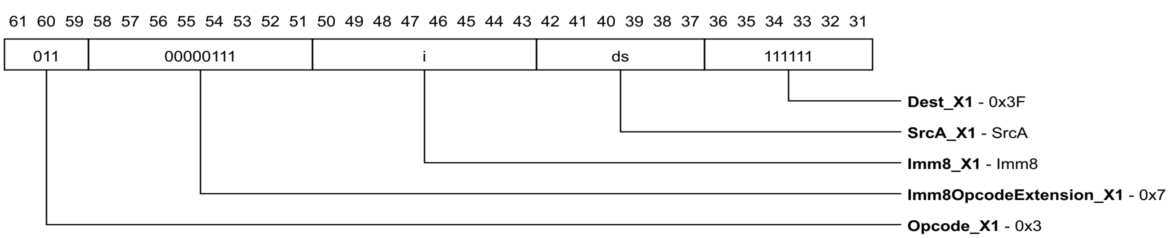

Figure 13-418: prefetch_add_l1_fault in X1 Bits Encoding

prefetch_add_l2

Prefetch to L2 and Add with No Faults

Syntax

```
prefetch_add_l2 SrcA, Imm8
```

Example

```
prefetch_add_l2 r5, 5
```

Description

Prefetch a cache line from memory into the level 2 and higher levels of cache without signalling of address or access interrupts. The address to be prefetched is read from the source operand. Add the signed immediate argument to the address register.

Functional Description

```
memoryPrefetch (rf[SrcA], 2, false);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

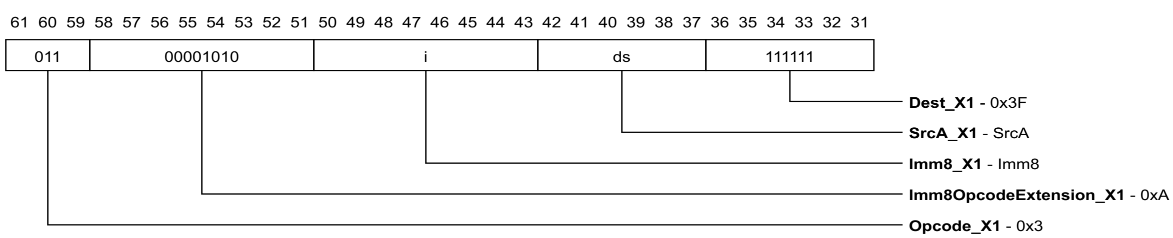

Figure 13-419: prefetch_add_l2 in X1 Bits Encoding

prefetch_add_l2_fault

Prefetch to L2 and Add with Faults

Syntax

```
prefetch_add_l2_fault SrcA, Imm8
```

Example

```
prefetch_add_l2_fault r5, 5
```

Description

Prefetch a cache line from memory into the level 2 and higher levels of cache with signalling of address or access interrupts. The address to be prefetched is read from the source operand. Add the signed immediate argument to the address register even if an interrupt is signalled.

Functional Description

```
memoryPrefetch (rf[SrcA], 2, true);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

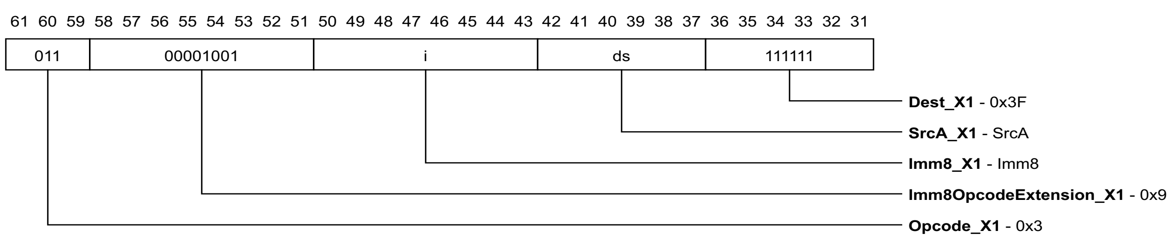

Figure 13-420: prefetch_add_l2_fault in X1 Bits Encoding

prefetch_add_l3

Prefetch to L3 and Add with No Faults

Syntax

```
prefetch_add_l3 SrcA, Imm8
```

Example

```
prefetch_add_l3 r5, 5
```

Description

Prefetch a cache line from memory into the level 3 cache, that is only at the home tile of the cache line, without signalling of address or access interrupts. The address to be prefetched is read from the source operand. Add the signed immediate argument to the address register.

Functional Description

```
memoryPrefetch (rf[SrcA], 4, false);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

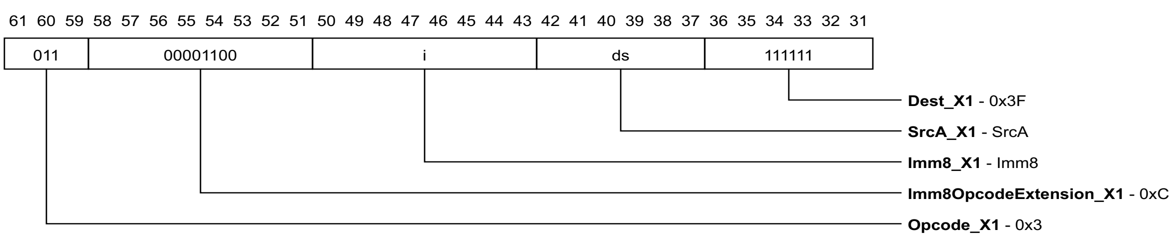

Figure 13-421: prefetch_add_l3 in X1 Bits Encoding

prefetch_add_l3_fault

Prefetch to L3 and Add with Faults

Syntax

```
prefetch_add_l3_fault SrcA, Imm8
```

Example

```
prefetch_add_l3_fault r5, 5
```

Description

Prefetch a cache line from memory into the level 3 cache, that is only at the home tile of the cache line, with signalling of address or access interrupts. The address to be prefetched is read from the source operand. Add the signed immediate argument to the address register even if an interrupt is signalled.

Functional Description

```
memoryPrefetch (rf[SrcA], 4, true);
rf[SrcA] = rf[SrcA] + signExtend8 (Imm8);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

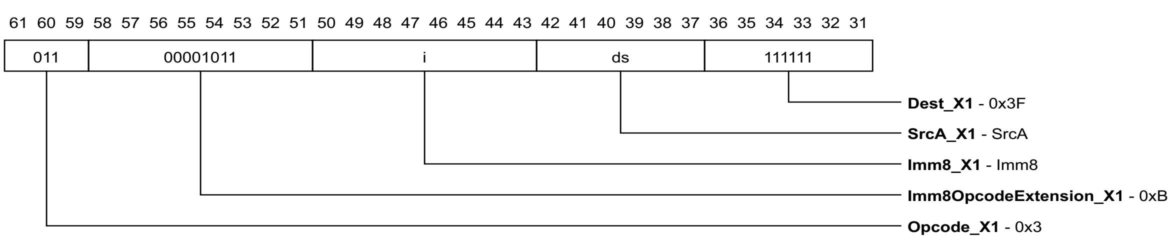

Figure 13-422: prefetch_add_l3_fault in X1 Bits Encoding

prefetch_l1

Prefetch to L1 with No Faults

Syntax

```
prefetch_l1 Src
```

Example

```
prefetch_l1 r5
```

Description

Prefetch a cache line from memory into all levels of cache without signalling any address or access interrupts. The address to be prefetched is read from the source operand.

Functional Description

```
memoryPrefetch (rf[Src], 1, false);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

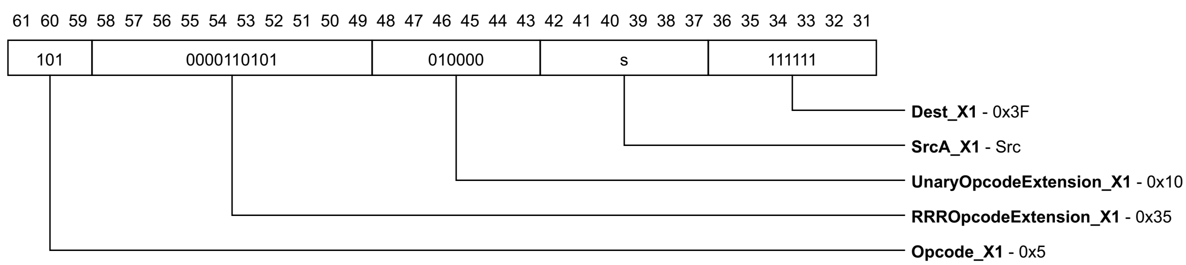

Figure 13-423: prefetch_l1 in X1 Bits Encoding

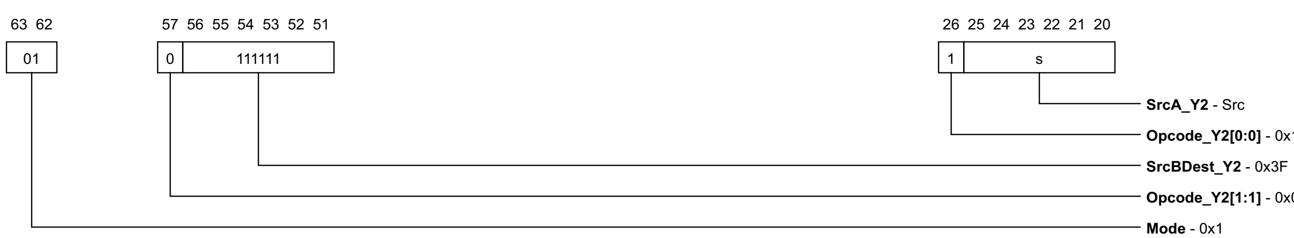

Figure 13-424: prefetch_l1 in Y2 Bits Encoding

prefetch_l1_fault

Prefetch to L1 with Faults

Syntax

```
prefetch_l1_fault Src
```

Example

```
prefetch_l1_fault r5
```

Description

Prefetch a cache line from memory into all levels of cache with signalling of address or access interrupts. The address to be prefetched is read from the source operand.

Functional Description

```
memoryPrefetch (rf[Src], 1, true);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

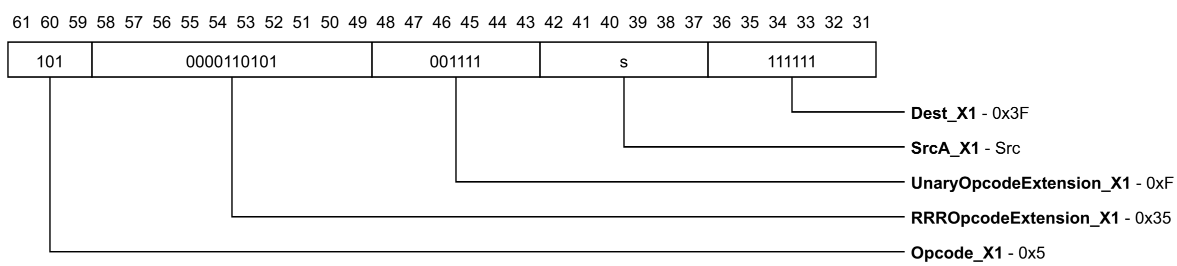

Figure 13-425: prefetch_l1_fault in X1 Bits Encoding

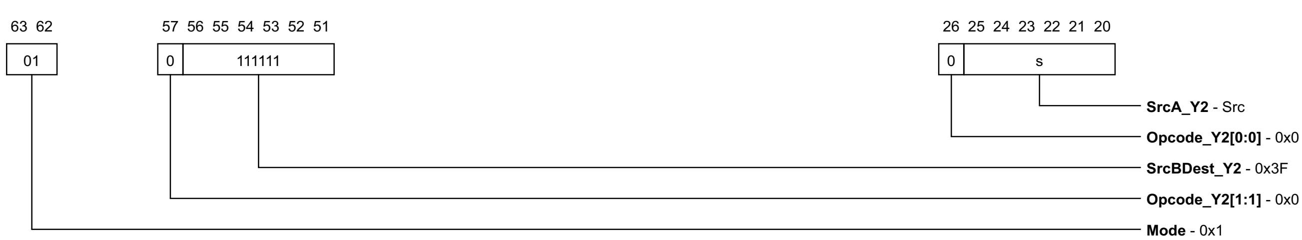

Figure 13-426: prefetch_l1_fault in Y2 Bits Encoding

prefetch_l2

Prefetch to L2 with No Faults

Syntax

```
prefetch_l2 Src
```

Example

```
prefetch_l2 r5
```

Description

Prefetch a cache line from memory into the level 2 and higher levels of cache without signalling any address or access interrupts. The address to be prefetched is read from the source operand.

Functional Description

```
memoryPrefetch (rf[Src], 2, false);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

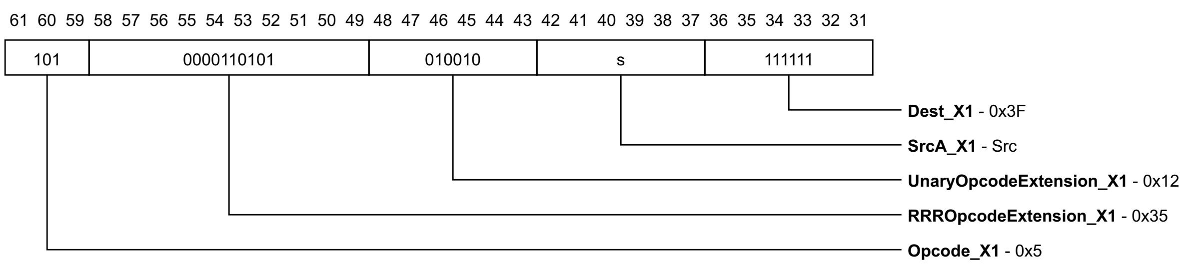

Figure 13-427: prefetch_l2 in X1 Bits Encoding

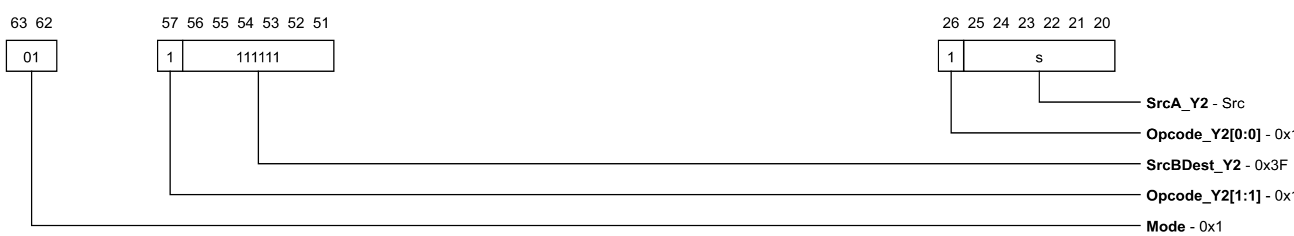

Figure 13-428: prefetch_l2 in Y2 Bits Encoding

prefetch_l2_fault

Prefetch to L2 with Faults

Syntax

```
prefetch_l2_fault Src
```

Example

```
prefetch_l2_fault r5
```

Description

Prefetch a cache line from memory into the level 2 and higher levels of cache with signalling of address or access interrupts. The address to be prefetched is read from the source operand.

Functional Description

```
memoryPrefetch (rf[Src], 2, true);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

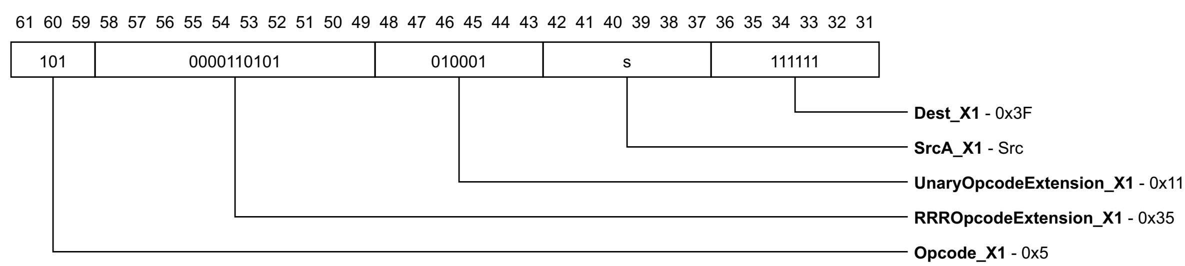

Figure 13-429: prefetch_l2_fault in X1 Bits Encoding

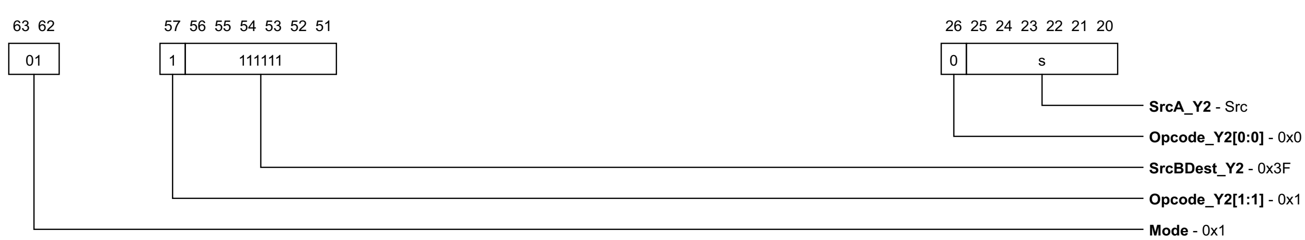

Figure 13-430: prefetch_l2_fault in Y2 Bits Encoding

prefetch_l3

Prefetch to L3 with No Faults

Syntax

```
prefetch_l3 Src
```

Example

```
prefetch_l3 r5
```

Description

Prefetch a cache line from memory into the level 3 cache, that is only at the home tile of the cache line, without signalling any address or access interrupts. The address to be prefetched is read from the source operand.

Functional Description

```
memoryPrefetch (rf[Src], 4, false);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

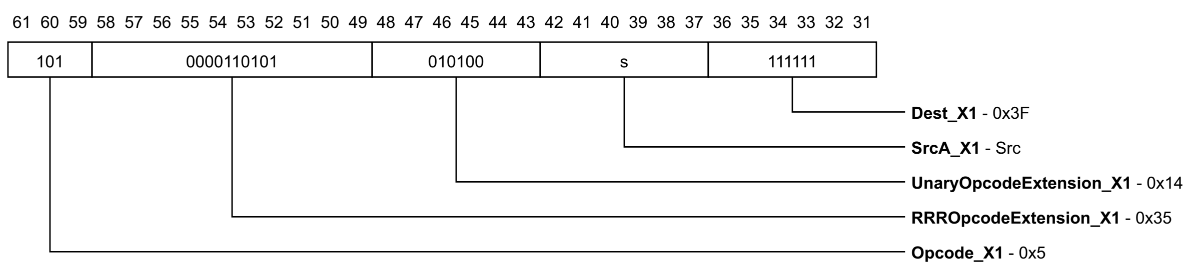

Figure 13-431: prefetch_l3 in X1 Bits Encoding

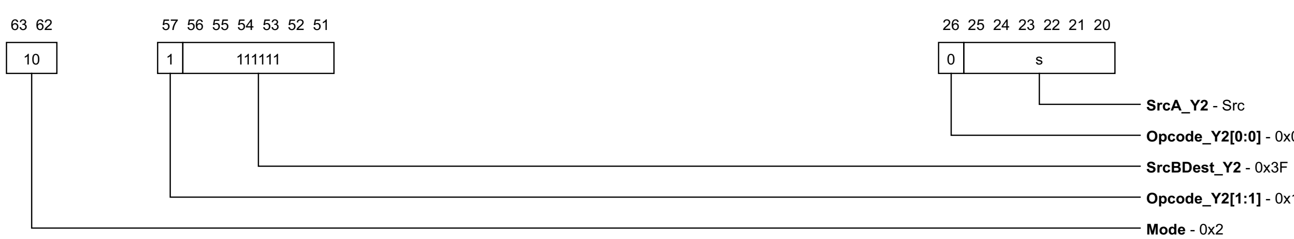

Figure 13-432: prefetch_l3 in Y2 Bits Encoding

prefetch_l3_fault

Prefetch to L3 with Faults

Syntax

```
prefetch_l3_fault Src
```

Example

```
prefetch_l3_fault r5
```

Description

Prefetch a cache line from memory into the level 3 cache, that is only at the home tile of the cache line, with signalling of address or access interrupts. The address to be prefetched is read from the source operand.

Functional Description

```
memoryPrefetch (rf[Src], 4, true);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

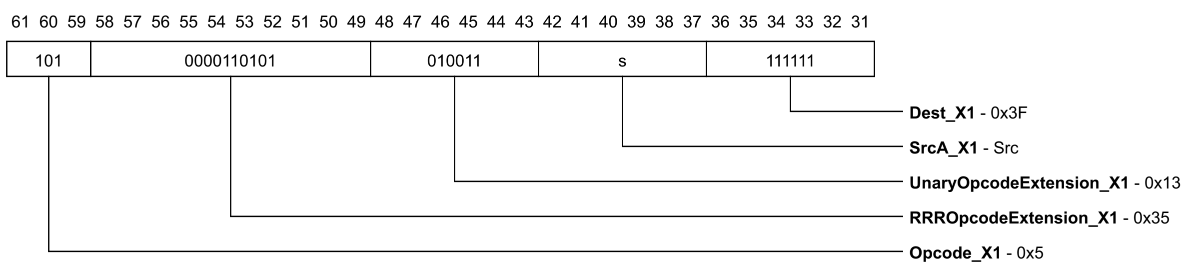

Figure 13-433: prefetch_l3_fault in X1 Bits Encoding

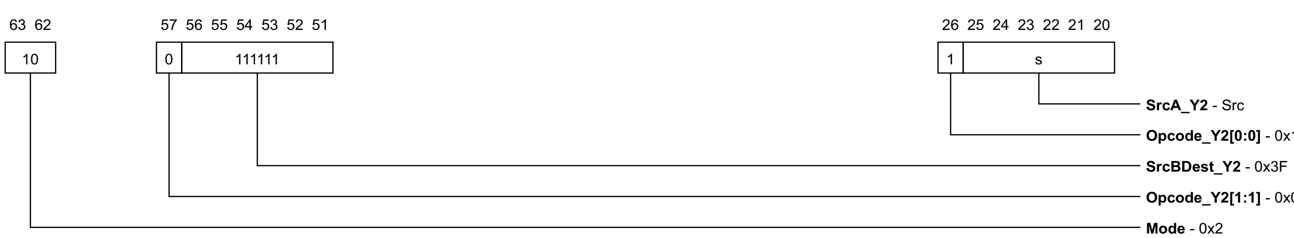

Figure 13-434: prefetch_l3_fault in Y2 Bits Encoding

raise

Raise Signal

Syntax

```
raise
```

Example

```
raise
```

Description

Causes an illegal instruction interrupt to occur. The Illegal Instruction is guaranteed to always cause an illegal instruction interrupt for all current and future derivations of the architecture. The runtime can use the `BreakpointDistinguisher` fields to characterize the illegal instruction as a request to raise a signal; it must be bundled with a “`moveli zero, VAL`” instruction where the low 6 bits of VAL are the signal and the next 4 bits hold an optional `si_code` value.

Functional Description

```
illegalInstruction ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

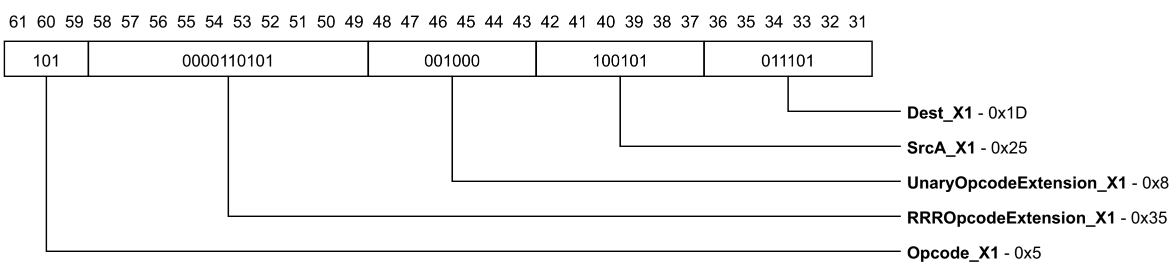

Figure 13-435: raise in X1 Bits Encoding

CHAPTER 14 SIMD INSTRUCTIONS

14.1 Overview

The following sections provide detailed descriptions of simd instructions listed alphabetically.

v1add	Vector One Byte Add
v1addi	Vector One Byte Add Immediate
v1adduc	Vector One Byte Add Unsigned Clamped
v1adiffu	Vector One Byte Absolute Difference Unsigned
v1avgu	Vector One Byte Average Unsigned
v1cmpeq	Vector One Byte Set Equal To
v1cmpeqi	Vector One Byte Set Equal To Immediate
v1cmples	Vector One Byte Set Less Than or Equal
v1cmpleu	Vector One Byte Set Less Than or Equal Unsigned
v1cmplts	Vector One Byte Set Less Than
v1cmpltsi	Vector One Byte Set Less Than Immediate
v1cmpltu	Vector One Byte Set Less Than Unsigned
v1cmpltui	Vector One Byte Set Less Than Unsigned Immediate
v1cmpne	Vector One Byte Set Not Equal To
v1ddotpu	Vector One Byte Dual Dot Product Unsigned
v1dotpua	Vector One Byte Dual Dot Product Unsigned and Accumulate
v1ddotpus	Vector One Byte Dual Dot Product Unsigned Signed
v1ddotpusa	Vector One Byte Dual Dot Product Unsigned Signed and Accumulate
v1dotp	Vector One Byte Dot Product
v1dotpa	Vector One Byte Dot Product and Accumulate
v1dotpu	Vector One Byte Dot Product Unsigned
v1dotpua	Vector One Byte Dot Product Unsigned and Accumulate
v1dotpus	Vector One Byte Dot Product Unsigned Signed
v1dotpusa	Vector One Byte Dot Product Unsigned Signed and Accumulate
v1int_h	Vector One Byte Interleave High
v1int_l	Vector One Byte Interleave Low

v1maxu	Vector One Byte Maximum Unsigned
v1maxui	Vector One Byte Maximum Unsigned Immediate
v1minu	Vector One Byte Minimum Unsigned
v1minui	Vector One Byte Minimum Unsigned Immediate
v1mnz	Vector One Byte Mask Not Zero
v1multu	Vector One Byte Multiply and Truncate Unsigned
v1mulu	Vector One Byte Multiply Unsigned
v1mulus	Vector One Byte Multiply Unsigned Signed
v1mz	Vector One Byte Mask Zero
v1sadau	Vector One Byte Sum of Absolute Difference Accumulate Unsigned
v1sadu	Vector One Byte Sum of Absolute Difference Unsigned
v1shl	Vector One Byte Shift Left
v1shli	Vector One Byte Shift Left Immediate
v1shrs	Vector One Byte Shift Right Signed
v1shrsi	Vector One Byte Shift Right Signed Immediate
v1shru	Vector One Byte Shift Right Unsigned
v1shrui	Vector One Byte Shift Right Unsigned Immediate
v1sub	Vector One Byte Subtract
v1subuc	Vector One Byte Subtract Unsigned Clamped
v2add	Vector Two Byte Add
v2addi	Vector Two Byte Add Immediate
v2addsc	Vector Two Byte Add Signed Clamped
v2adiffs	Vector Two Byte Absolute Difference Signed
v2avgs	Vector Two Byte Average Signed
v2cmpeq	Vector Two Byte Set Equal To
v2cmpeqi	Vector Two Byte Set Equal To Immediate
v2cmples	Vector Two Byte Set Less Than or Equal
v2cmpleu	Vector Two Byte Set Less Than or Equal Unsigned
v2cmplts	Vector Two Byte Set Less Than
v2cmpltsi	Vector Two Byte Set Less Than Immediate
v2cmpltu	Vector Two Byte Set Less Than Unsigned
v2cmpltui	Vector Two Byte Set Less Than Unsigned Immediate
v2cmpne	Vector Two Byte Set Not Equal To
v2dotp	Vector Two Byte Dot Product
v2dotpa	Vector Two Byte Dot Product and Accumulate
v2int_h	Vector Two Byte Interleave High
v2int_l	Vector Two Byte Interleave Low

v2maxs	Vector Two Byte Maximum Signed
v2maxsi	Vector Two Byte Maximum Signed Immediate
v2mins	Vector Two Byte Minimum Signed
v2minsi	Vector Two Byte Minimum Signed Immediate
v2mnz	Vector Two Byte Mask Not Zero
v2mulfsc	Vector Two Byte Multiply Fixed point Signed Clamped
v2muls	Vector Two Byte Multiply Signed
v2mults	Vector Two Byte Multiply and Truncate Signed
v2mz	Vector Two Byte Mask Zero
v2packh	Vector Two Bytes Pack High Byte
v2packl	Vector Two Byte Pack Low Byte
v2packuc	Vector Two Byte Pack Unsigned Clamped
v2sadas	Vector Two Byte Sum of Absolute Difference Accumulate Signed
v2sadau	Vector Two Byte Sum of Absolute Difference Accumulate Unsigned
v2sads	Vector Two Byte Sum of Absolute Difference Signed
v2sadu	Vector Two Byte Sum of Absolute Difference Unsigned
v2shl	Vector Two Byte Shift Left
v2shli	Vector Two Byte Shift Left Immediate
v2shlsc	Vector Two Byte Shift Left Signed Clamped
v2shrs	Vector Two Byte Shift Right Signed
v2shrsi	Vector Two Byte Shift Right Signed Immediate
v2shru	Vector Two Byte Shift Right Unsigned
v2shrui	Vector Two Byte Shift Right Unsigned Immediate
v2sub	Vector Two Byte Subtract
v2subsc	Vector Two Byte Subtract Signed Clamped
v4add	Vector Four Byte Add
v4addsc	Vector Four Byte Add Signed Clamped
v4int_h	Vector Four Byte Interleave High
v4int_l	Vector Four Byte Interleave Low
v4packsc	Vector Four Byte Pack Signed Clamped
v4shl	Vector Four Byte Shift Left
v4shlsc	Vector Four Byte Shift Left Signed Clamped
v4shrs	Vector Four Byte Shift Right Signed
v4shru	Vector Four Byte Shift Right Unsigned
v4sub	Vector Four Byte Subtract
v4subsc	Vector Four Byte Subtract Signed Clamped

14.2 Instructions

SIMD instructions are described in the sections that follow.

v1add

Vector One Byte Add

Syntax

```
v1add Dest, SrcA, SrcB
```

Example

```
v1add r5, r6, r7
```

Description

Add the eight bytes in the first source operand to the eight bytes in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       (getByte (rf[SrcA], counter) + getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

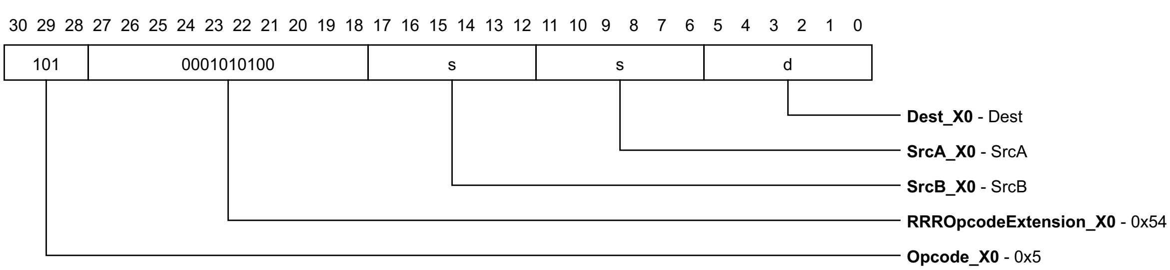

Figure 14-436: v1add in X0 Bits Encoding

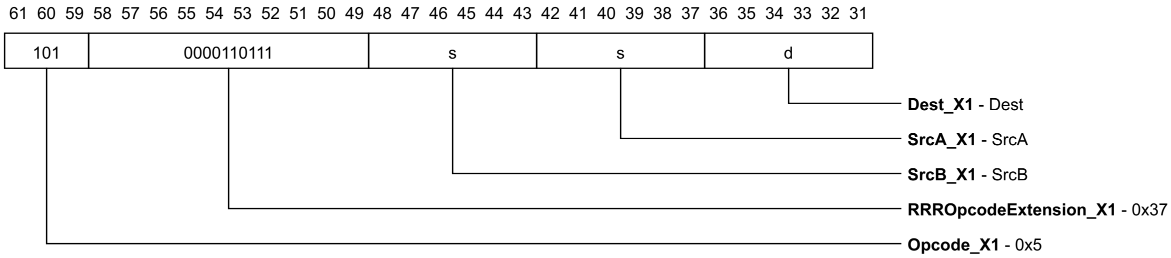

Figure 14-437: v1add in X1 Bits Encoding

v1addi

Vector One Byte Add Immediate

Syntax

```
v1addi Dest, SrcA, Imm8
```

Example

```
v1addi r5, r6, 5
```

Description

Add an immediate to all eight of the bytes in the source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output = setByte (output, counter, (getByte (rf[SrcA], counter) + Imm8));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

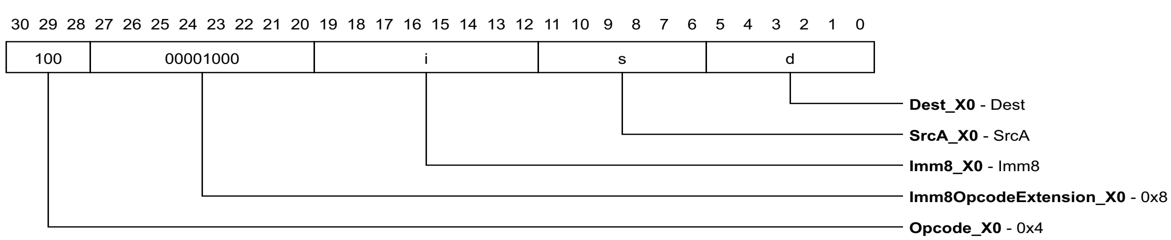

Figure 14-438: v1addi in X0 Bits Encoding

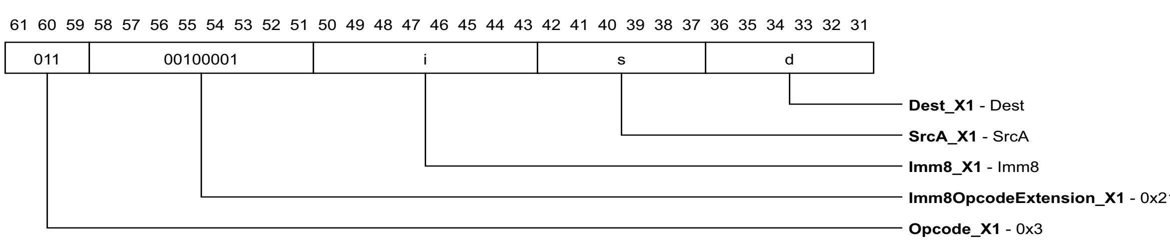

Figure 14-439: v1addi in X1 Bits Encoding

v1adduc

Vector One Byte Add Unsigned Clamped

Syntax

```
v1adduc Dest, SrcA, SrcB
```

Example

```
v1adduc r5, r6, r7
```

Description

Add the eight bytes in the first source operand to the eight bytes in the second source operand and clamp each result to 0 or the maximum positive value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       unsigned_saturate8 (getByte (rf[SrcA], counter) +
                           getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

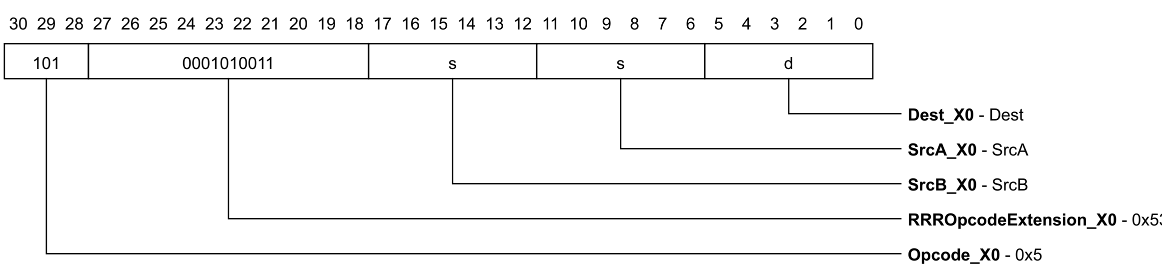

Figure 14-440: v1adduc in X0 Bits Encoding

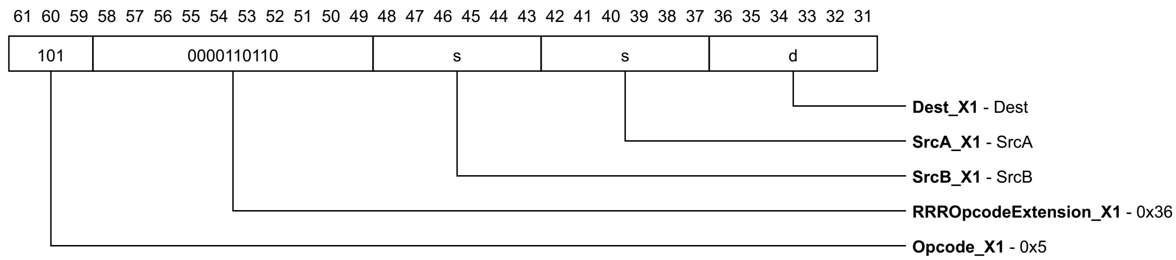

Figure 14-441: v1adduc in X1 Bits Encoding

v1adiffu

Vector One Byte Absolute Difference Unsigned

Syntax

```
v1adiffu Dest, SrcA, SrcB
```

Example

```
v1adiffu r5, r6, r7
```

Description

Compute the absolute differences between the eight bytes in the first source operand and the eight bytes in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       abs (getByte (rf[SrcA], counter) - getByte (SrcB, counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

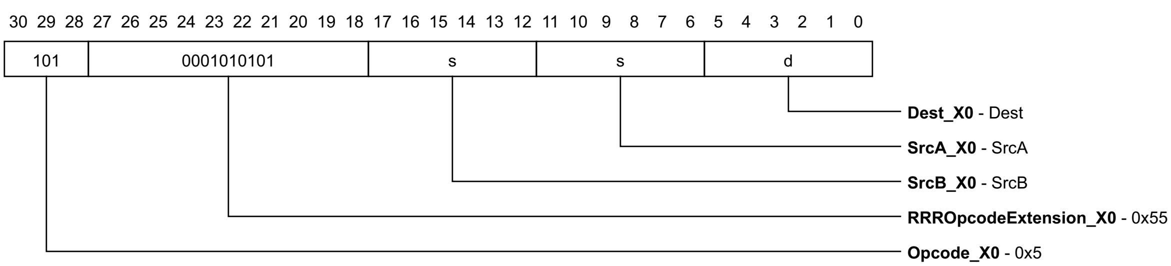

Figure 14-442: v1adiffu in X0 Bits Encoding

v1avgu

Vector One Byte Average Unsigned

Syntax

```
v1avgu Dest, SrcA, SrcB
```

Example

```
v1avgu r5, r6, r7
```

Description

Compute the average of the eight bytes in the first source operand and the eight bytes in the second source operand, rounding upwards.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    UnsignedMachineWord srca = getByte (rf[SrcA], counter);
    UnsignedMachineWord srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca + srcb + 1) >> 1));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

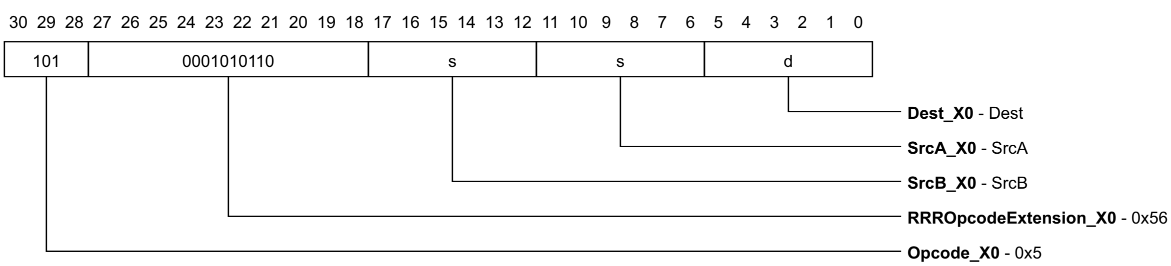

Figure 14-443: v1avgu in X0 Bits Encoding

v1cmpeq

Vector One Byte Set Equal To

Syntax

```
v1cmpeq Dest, SrcA, SrcB
```

Example

```
v1cmpeq r5, r6, r7
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is equal to the byte of the second source operand. Otherwise the result is set to 0. This instruction treats both source bytes as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca == srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

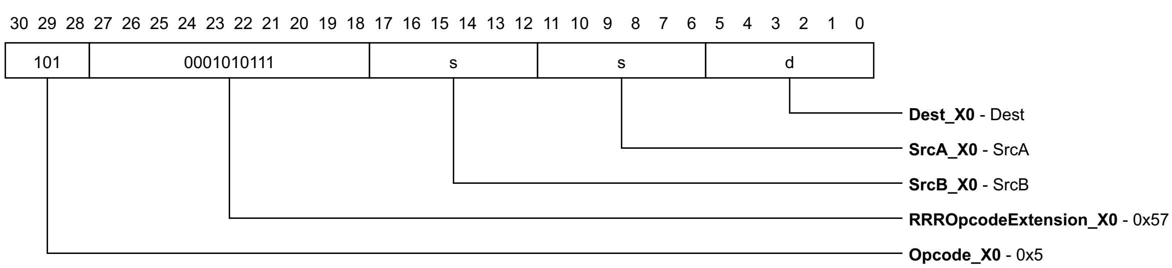

Figure 14-444: v1cmpeq in X0 Bits Encoding

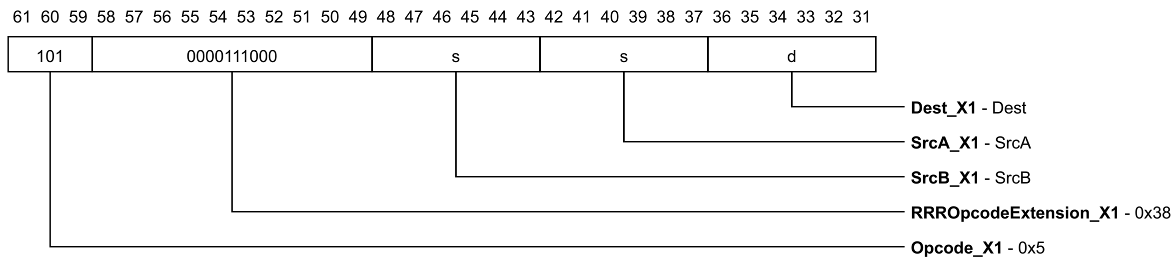

Figure 14-445: v1cmpeq in X1 Bits Encoding

v1cmpeqi

Vector One Byte Set Equal To Immediate

Syntax

```
v1cmpeqi Dest, SrcA, Imm8
```

Example

```
v1cmpeqi r5, r6, 5
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is equal to a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source bytes as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = signExtend8 (Imm8);
    output = setByte (output, counter, ((srca == srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

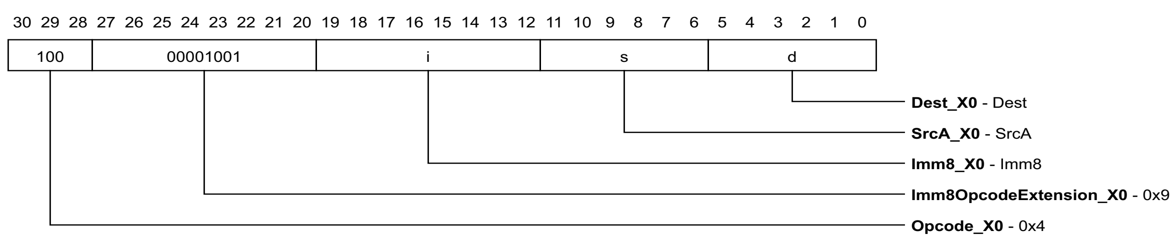

Figure 14-446: v1cmpeqi in X0 Bits Encoding

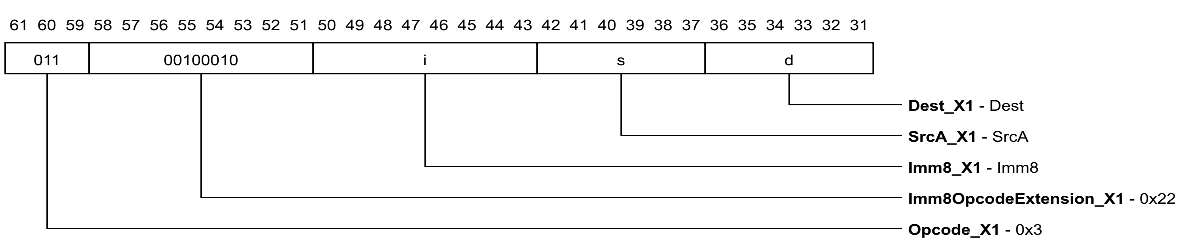

Figure 14-447: v1cmpeqi in X1 Bits Encoding

v1cmples

Vector One Byte Set Less Than or Equal

Syntax

```
v1cmples Dest, SrcA, SrcB
```

Example

```
v1cmples r5, r6, r7
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is less than or equal to the byte of the second source operand. Otherwise the result is set to 0. This instruction treats both source bytes as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca <= srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

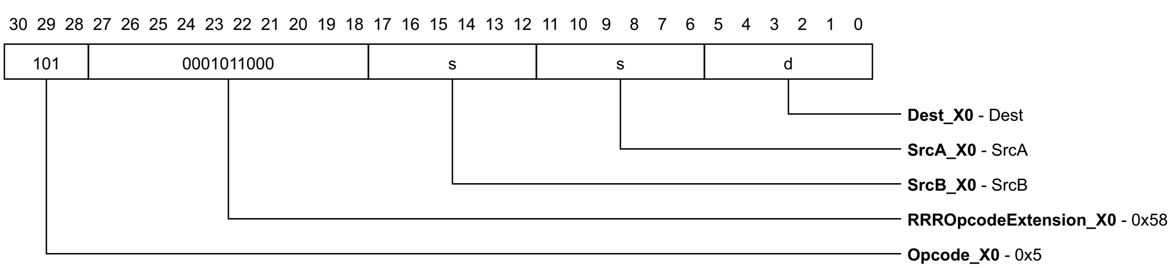

Figure 14-448: v1cmples in X0 Bits Encoding

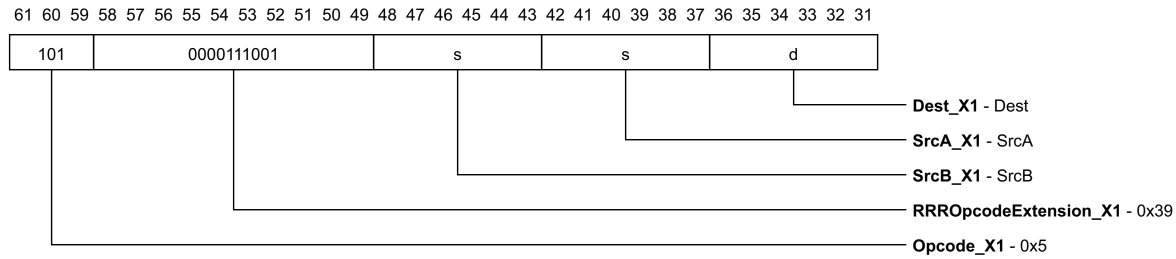

Figure 14-449: v1cmples in X1 Bits Encoding

v1cmpleu

Vector One Byte Set Less Than or Equal Unsigned

Syntax

```
v1cmpleu Dest, SrcA, SrcB
```

Example

```
v1cmpleu r5, r6, r7
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is less than or equal to the byte of the second source operand. Otherwise the result is set to 0. This instruction treats both source bytes as unsigned values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    uint8_t srca = getByte (rf[SrcA], counter);
    uint8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca <= srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

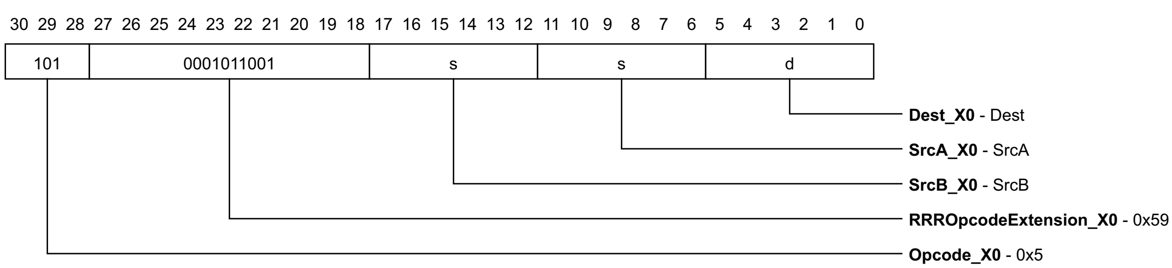

Figure 14-450: v1cmpleu in X0 Bits Encoding

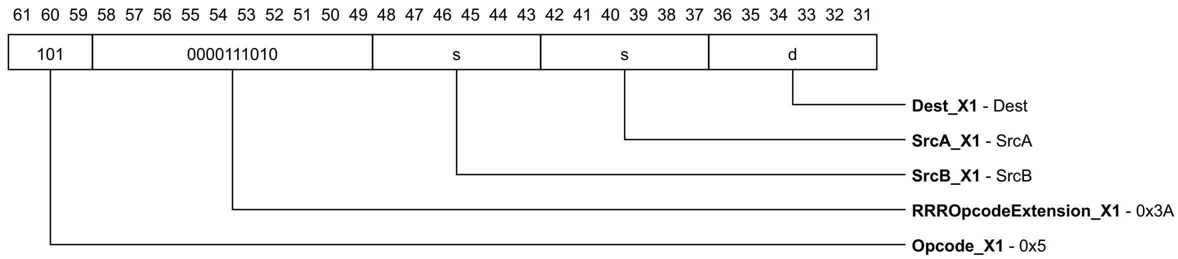

Figure 14-451: v1cmpleu in X1 Bits Encoding

v1cmplts

Vector One Byte Set Less Than

Syntax

```
v1cmplts Dest, SrcA, SrcB
```

Example

```
v1cmplts r5, r6, r7
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is less than the byte of the second source operand. Otherwise the result is set to 0. This instruction treats both source bytes as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

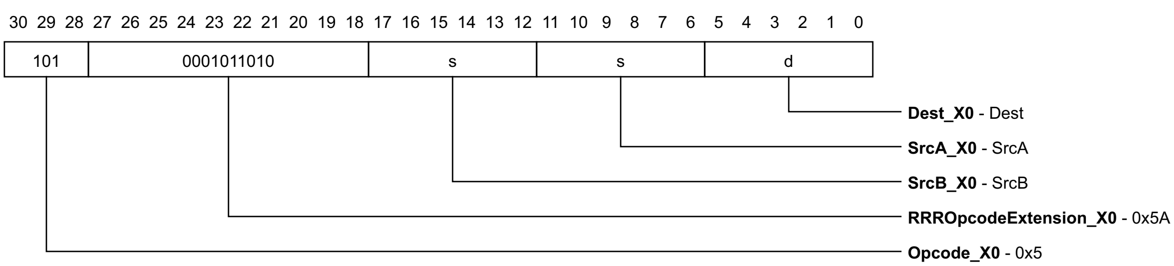

Figure 14-452: v1cmplts in X0 Bits Encoding

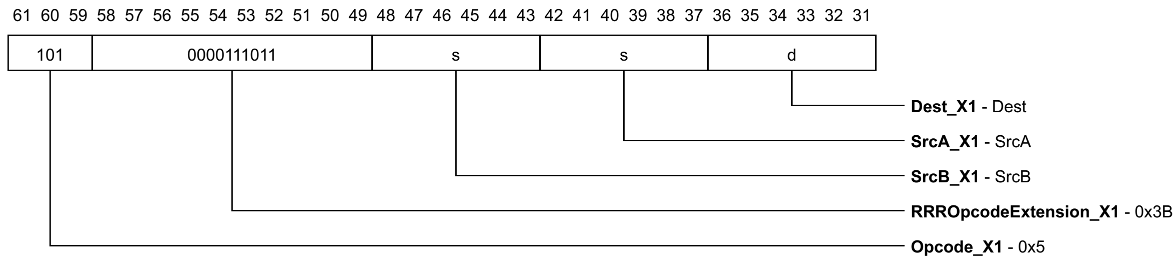

Figure 14-453: v1cmplts in X1 Bits Encoding

v1cmpltsi

Vector One Byte Set Less Than Immediate

Syntax

```
v1cmpltsi Dest, SrcA, Imm8
```

Example

```
v1cmpltsi r5, r6, 5
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source bytes as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = signExtend8 (Imm8) & BYTE_MASK;
    output = setByte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

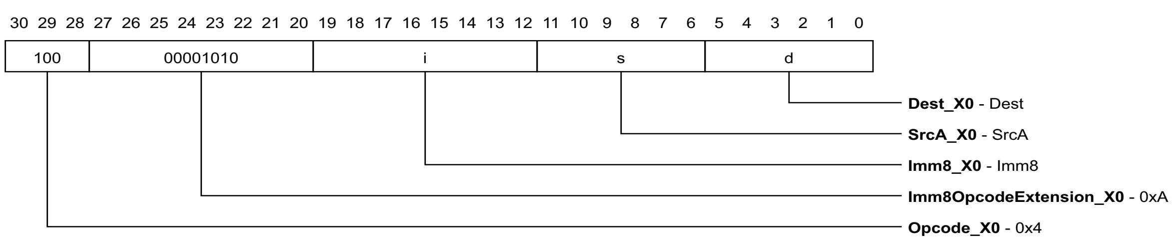

Figure 14-454: v1cmpltsi in X0 Bits Encoding

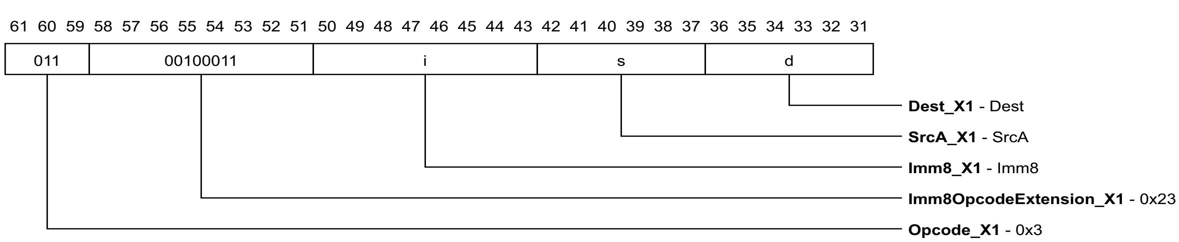

Figure 14-455: v1cmpltsi in X1 Bits Encoding

v1cmpltu

Vector One Byte Set Less Than Unsigned

Syntax

```
v1cmpltu Dest, SrcA, SrcB
```

Example

```
v1cmpltu r5, r6, r7
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is less than the byte of the second source operand. Otherwise the result is set to 0. This instruction treats both source bytes as unsigned values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    uint8_t srca = getByte (rf[SrcA], counter);
    uint8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

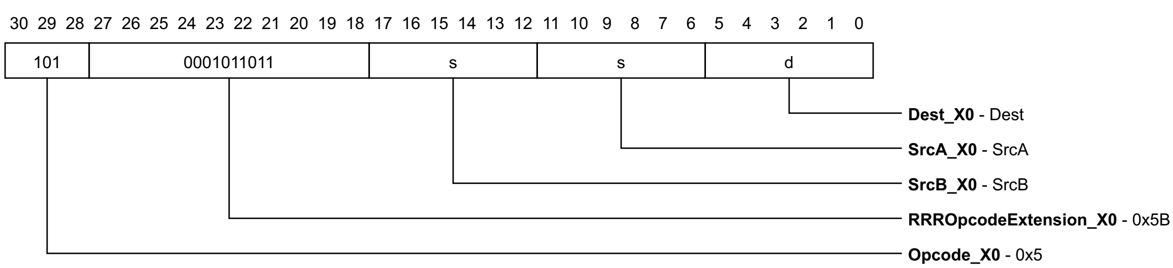

Figure 14-456: v1cmpltu in X0 Bits Encoding

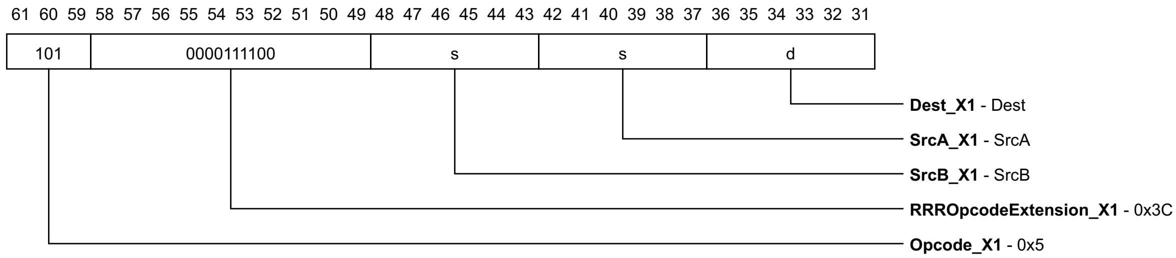

Figure 14-457: v1cmpltu in X1 Bits Encoding

v1cmpltui

Vector One Byte Set Less Than Unsigned Immediate

Syntax

```
v1cmpltui Dest, SrcA, Imm8
```

Example

```
v1cmpltui r5, r6, 5
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source bytes as unsigned values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    uint8_t srca = getByte (rf[SrcA], counter);
    uint8_t srcb = signExtend8 (Imm8);
    output = setByte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

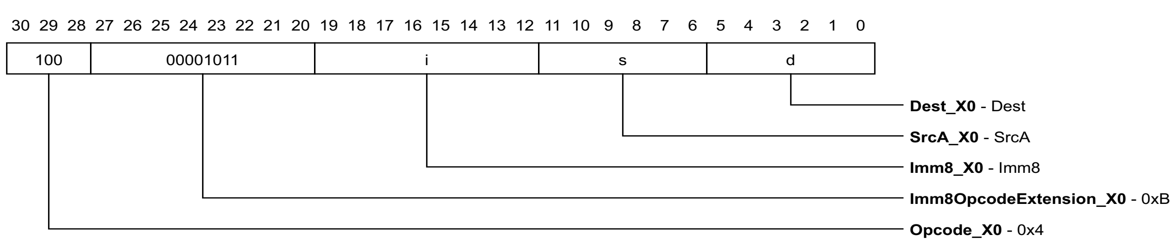

Figure 14-458: v1cmpltui in X0 Bits Encoding

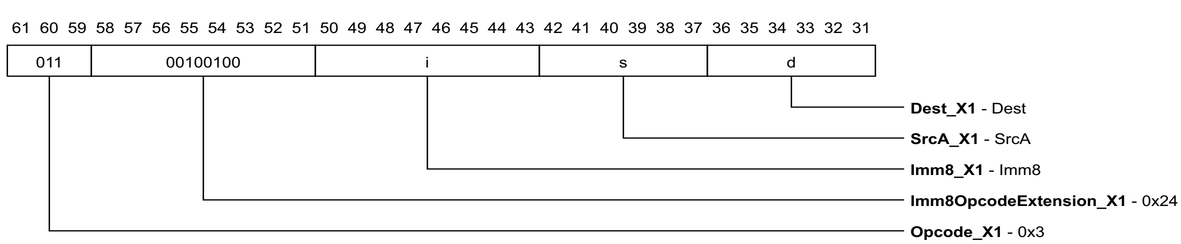

Figure 14-459: v1cmpltui in X1 Bits Encoding

v1cmpne

Vector One Byte Set Not Equal To

Syntax

```
v1cmpne Dest, SrcA, SrcB
```

Example

```
v1cmpne r5, r6, r7
```

Description

Sets each result byte to 1 if the corresponding byte of the first source operand is not equal to the byte of the second source operand. Otherwise the result is set to 0. This instruction treats both source bytes as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca != srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

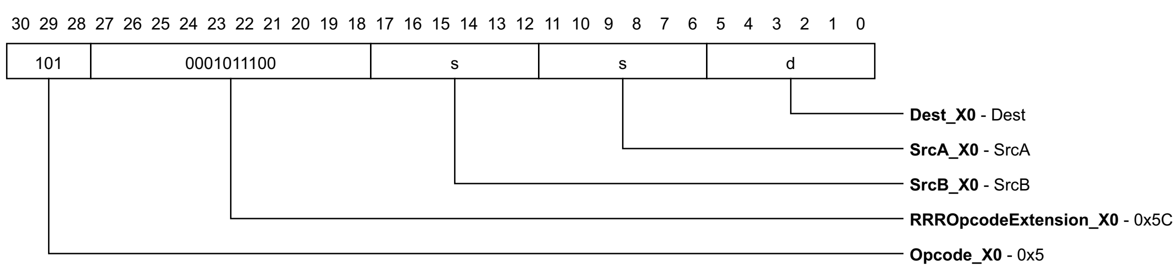

Figure 14-460: v1cmpne in X0 Bits Encoding

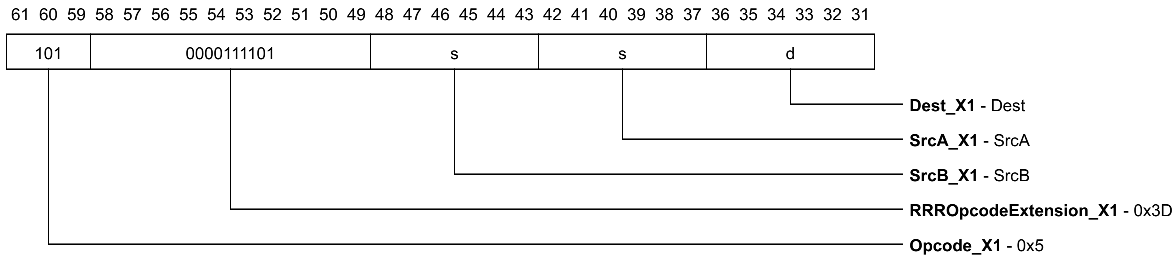

Figure 14-461: v1cmpne in X1 Bits Encoding

v1ddotpu

Vector One Byte Dual Dot Product Unsigned

Syntax

```
v1ddotpu Dest, SrcA, SrcB
```

Example

```
v1ddotpu r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand. The low 32-bits of the result are set to the sum of products of the low-order four 8-bit quantities, and the high 32-bits of the result are set to the sum of the products of the high-order four 8-bit quantities. The quantities in the operands are treated as unsigned values.

Functional Description

```
uint64_t packed_output = 0;
int32_t output;
uint32_t counter;
uint32_t half;
for (half = 0; half < 2; half++)
    {
      uint32_t offset = half * ((WORD_SIZE / 2) / 8);
      output = 0;
      for (counter = 0; counter < ((WORD_SIZE / 2) / 8); counter++)
       {
         output +=
              ((uint16_t) getByte (rf[SrcA], counter + offset) *
               (uint16_t) getByte (rf[SrcB], counter + offset));
       }
      packed_output = set4Byte (packed_output, half, output);
    }

rf[Dest] = packed_output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

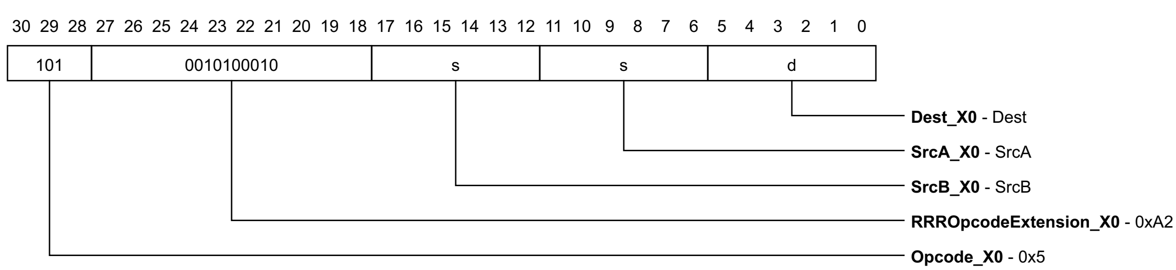

Figure 14-462: v1ddotpu in X0 Bits Encoding

v1ddotpua

Vector One Byte Dual Dot Product Unsigned and Accumulate

Syntax

```
v1ddotpua Dest, SrcA, SrcB
```

Example

```
v1ddotpua r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand. The low 32-bits of the result are set to the sum low-order 32-bit quantity of the designation and the products of the low-order four 8-bit quantities, and the high 32-bits of the result are set to the sum of the high-order 32-bit quantity of the destination and the products of the high-order four 8-bit quantities. The quantities in the operands are treated as unsigned values.

Functional Description

```
uint64_t packed_output = 0;
int32_t output;
uint32_t counter;
uint32_t half;
for (half = 0; half < 2; half++)
   {
      uint32_t offset = half * ((WORD_SIZE / 2) / 8);
      output = 0;
      for (counter = 0; counter < ((WORD_SIZE / 2) / 8); counter++)
      {
          output +=
             ((uint16_t) getByte (rf[SrcA], counter + offset) *
              (uint16_t) getByte (rf[SrcB], counter + offset));
      }
      packed_output =
        set4Byte (packed_output, half, get4Byte (rf[Dest], half) + output);
   }

rf[Dest] = packed_output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

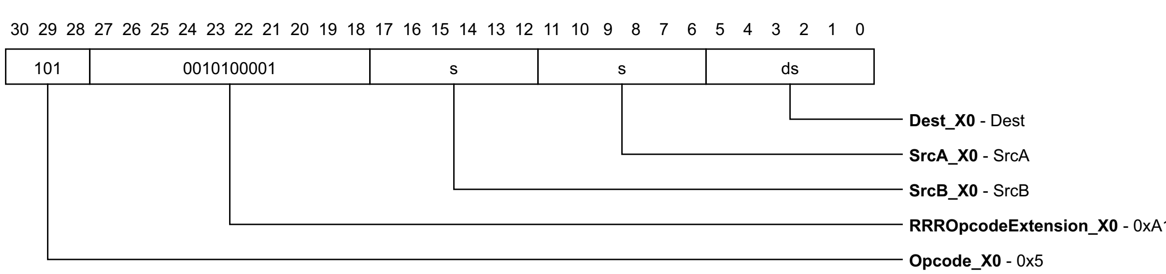

Figure 14-463: v1ddotpua in X0 Bits Encoding

v1ddotpus

Vector One Byte Dual Dot Product Unsigned Signed

Syntax

```
v1ddotpus Dest, SrcA, SrcB
```

Example

```
v1ddotpus r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand. The low 32-bits of the result are set to the sum of products of the low-order four 8-bit quantities, and the high 32-bits of the result are set to the sum of the products of the high-order four 8-bit quantities. The quantities in the first operand are treated as unsigned values, and the quantities in the second operand are treated as signed values.

Functional Description

```
uint64_t packed_output = 0;
int32_t output;
uint32_t counter;
uint32_t half;
for (half = 0; half < 2; half++)
  {
    uint32_t offset = half * ((WORD_SIZE / 2) / 8);
    output = 0;
    for (counter = 0; counter < ((WORD_SIZE / 2) / 8); counter++)
      {
output +=
  ((uint16_t) getByte (rf[SrcA], counter + offset) *
   (int16_t) signExtend8 (getByte (rf[SrcB], counter + offset)));
      }
    packed_output = set4Byte (packed_output, half, output);
  }

rf[Dest] = packed_output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

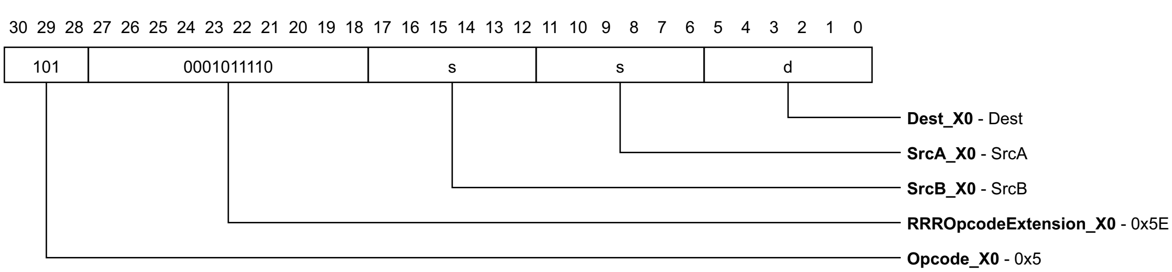

Figure 14-464: v1ddotpus in X0 Bits Encoding

v1ddotpusa

Vector One Byte Dual Dot Product Unsigned Signed and Accumulate

Syntax

```
v1ddotpusa Dest, SrcA, SrcB
```

Example

```
v1ddotpusa r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand. The low 32-bits of the result are set to the sum low-order 32-bit quantity of the designation and the products of the low-order four 8-bit quantities, and the high 32-bits of the result are set to the sum of the high-order 32-bit quantity of the destination and the products of the high-order four 8-bit quantities. The quantities in the first operand are treated as unsigned values, and the quantities in the second operand are treated as signed values.

Functional Description

```
uint64_t packed_output = 0;
int32_t output;
uint32_t counter;
uint32_t half;
for (half = 0; half < 2; half++)
  {
    uint32_t offset = half * ((WORD_SIZE / 2) / 8);
    output = 0;
    for (counter = 0; counter < ((WORD_SIZE / 2) / 8); counter++)
      {
output +=
  ((uint16_t) getByte (rf[SrcA], counter + offset) *
   (int16_t) signExtend8 (getByte (rf[SrcB], counter + offset)));
      }
    packed_output =
      set4Byte (packed_output, half, get4Byte (rf[Dest], half) + output);
  }

rf[Dest] = packed_output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

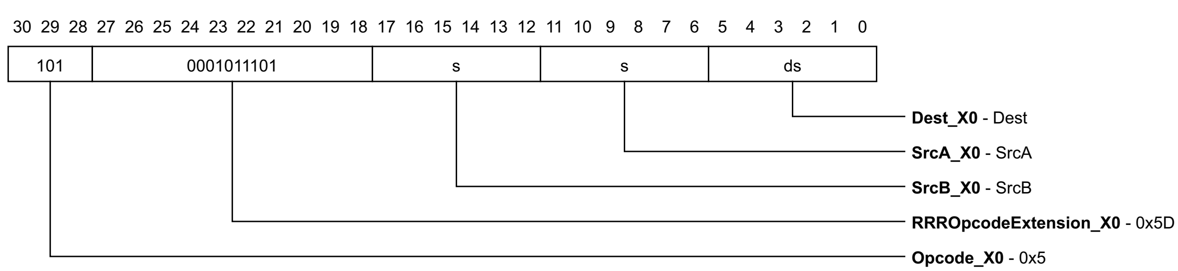

Figure 14-465: v1ddotpusa in X0 Bits Encoding

v1dotp

Vector One Byte Dot Product

Syntax

```
v1dotp Dest, SrcA, SrcB
```

Example

```
v1dotp r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand and sum the products.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 8); counter++)
  {
    output +=
      ((int16_t) signExtend8 (getByte (rf[SrcA], counter)) *
       (int16_t) signExtend8 (getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

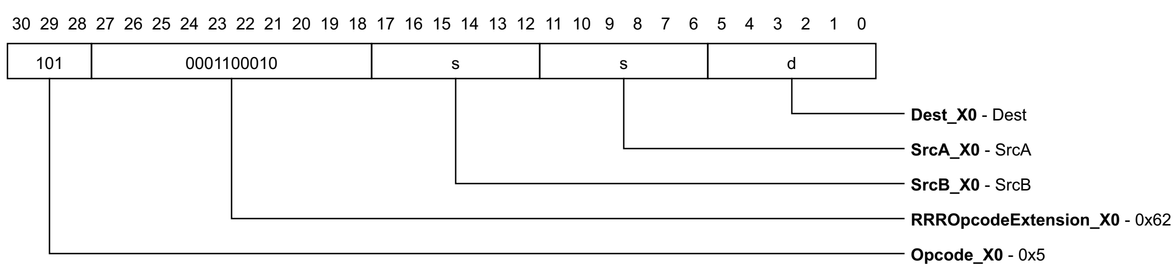

Figure 14-466: v1dotp in X0 Bits Encoding

v1dotpa

Vector One Byte Dot Product and Accumulate

Syntax

```
v1dotpa Dest, SrcA, SrcB
```

Example

```
v1dotpa r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand and sum the products and the destination operand.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 8); counter++)
  {
    output +=
      ((int16_t) signExtend8 (getByte (rf[SrcA], counter)) *
       (int16_t) signExtend8 (getByte (rf[SrcB], counter)));
  }

rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

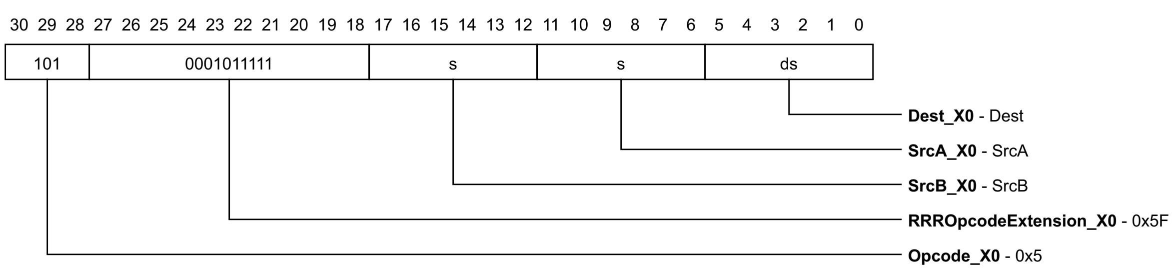

Figure 14-467: v1dotpa in X0 Bits Encoding

v1dotpu

Vector One Byte Dot Product Unsigned

Syntax

```
v1dotpu Dest, SrcA, SrcB
```

Example

```
v1dotpu r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand and sum the products. The quantities in the operands are treated as unsigned values.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 8); counter++)
  {
    output +=
      ((uint16_t) getByte (rf[SrcA], counter) *
       (uint16_t) getByte (rf[SrcB], counter));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

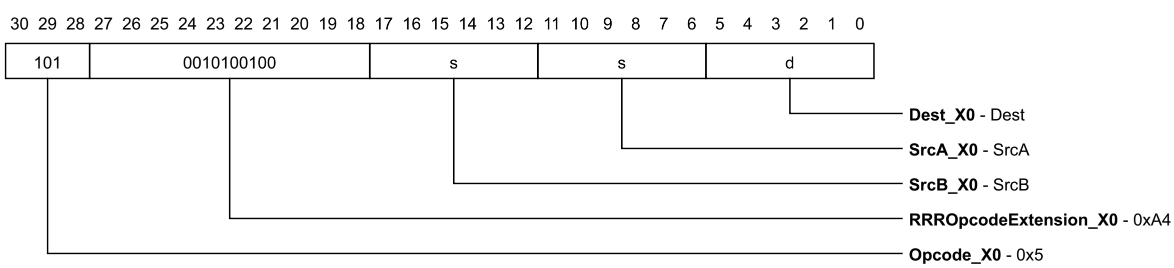

Figure 14-468: v1dotpu in X0 Bits Encoding

v1dotpua

Vector One Byte Dot Product Unsigned and Accumulate

Syntax

```
v1dotpua Dest, SrcA, SrcB
```

Example

```
v1dotpua r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand and sum the products and the destination operand. The quantities in the operands are treated as unsigned values.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 8); counter++)
  {
    output +=
      ((uint16_t) getByte (rf[SrcA], counter) *
       (uint16_t) getByte (rf[SrcB], counter));
  }

rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

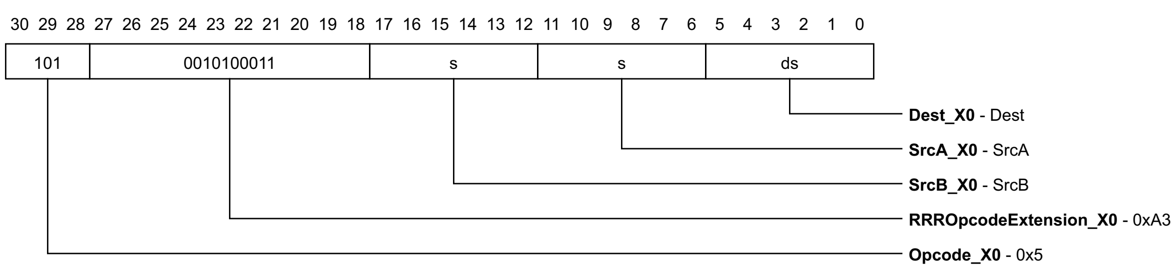

Figure 14-469: v1dotpua in X0 Bits Encoding

v1dotpus

Vector One Byte Dot Product Unsigned Signed

Syntax

```
v1dotpus Dest, SrcA, SrcB
```

Example

```
v1dotpus r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand and sum the products. The quantities in the first operand are treated as unsigned values, and the quantities in the second operand are treated as signed values.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 8); counter++)
  {
    output +=
      ((uint16_t) getByte (rf[SrcA], counter) *
       (int16_t) signExtend8 (getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

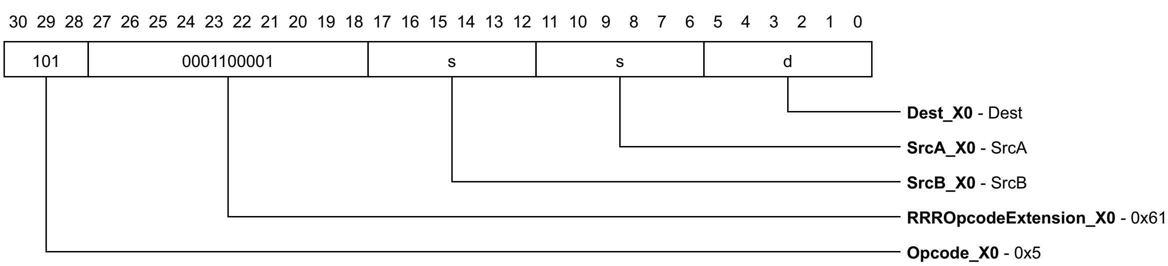

Figure 14-470: v1dotpus in X0 Bits Encoding

v1dotpusa

Vector One Byte Dot Product Unsigned Signed and Accumulate

Syntax

```
v1dotpusa Dest, SrcA, SrcB
```

Example

```
v1dotpusa r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand and sum the products and the destination operand. The quantities in the first operand are treated as unsigned values, and the quantities in the second operand are treated as signed values.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 8); counter++)
  {
    output +=
      ((uint16_t) getByte (rf[SrcA], counter) *
       (int16_t) signExtend8 (getByte (rf[SrcB], counter)));
  }

rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

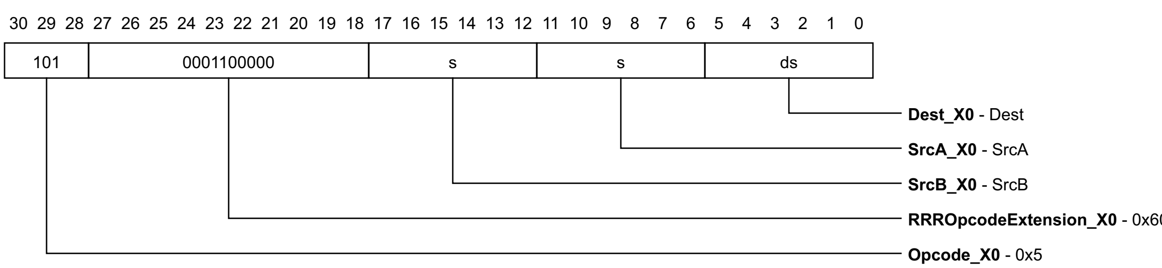

Figure 14-471: v1dotpusa in X0 Bits Encoding

v1int_h

Vector One Byte Interleave High

Syntax

```
v1int_h Dest, SrcA, SrcB
```

Example

```
v1int_h r5, r6, r7
```

Description

Interleave the four high-order bytes of the first operand with the four high-order bytes of the second operand. The high-order byte of the result will be the high-order byte of the first operand. For example if the first operand contains the packed bytes {`A7,A6,A5,A4,A3,A2,A1,A0`} and the second operand contains the packed bytes {`B7,B6,B5,B4,B3,B2,B1,B0`} then the result will be {`A7,B7,A6,B6,A5,B5,A4,B4`}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    bool asel = ((counter & 1) == 1);
    int in_sel = 4 + counter / 2;
    int8_t srca = getByte (rf[SrcA], in_sel);
    int8_t srcb = getByte (rf[SrcB], in_sel);
    output = setByte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

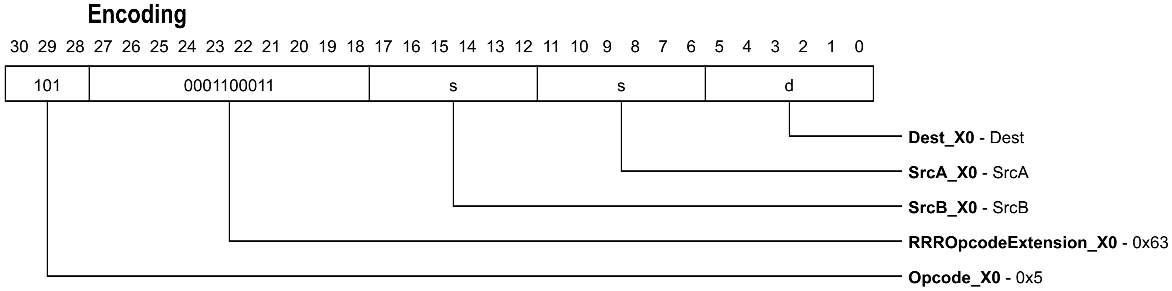

Figure 14-472: v1int_h in X0 Bits Encoding

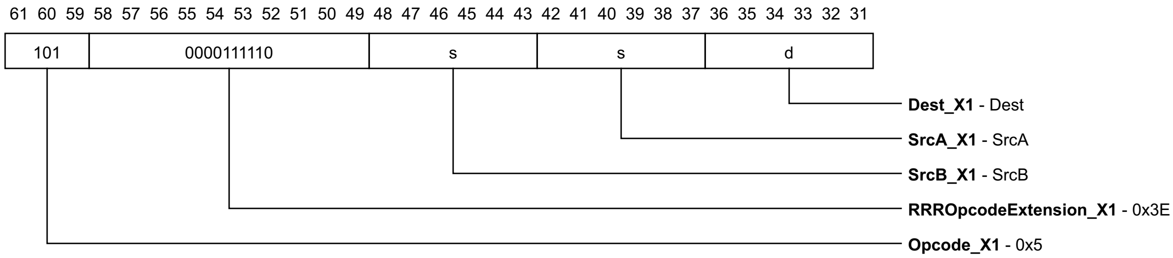

Figure 14-473: v1int_h in X1 Bits Encoding

v1int_l

Vector One Byte Interleave Low

Syntax

```
v1int_l Dest, SrcA, SrcB
```

Example

```
v1int_l r5, r6, r7
```

Description

Interleave the four low-order bytes of the first operand with the four low-order bytes of the second operand. The low-order byte of the result will be the low-order byte of the second operand. For example if the first operand contains the packed bytes {A7,A6,A5,A4,A3,A2,A1,A0} and the second operand contains the packed bytes {B7,B6,B5,B4,B3,B2,B1,B0} then the result will be {A3,B3,A2,B2,A1,B1,A0,B0}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    bool asel = ((counter & 1) == 1);
    int in_sel = 0 + counter / 2;
    int8_t srca = getByte (rf[SrcA], in_sel);
    int8_t srcb = getByte (rf[SrcB], in_sel);
    output = setByte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

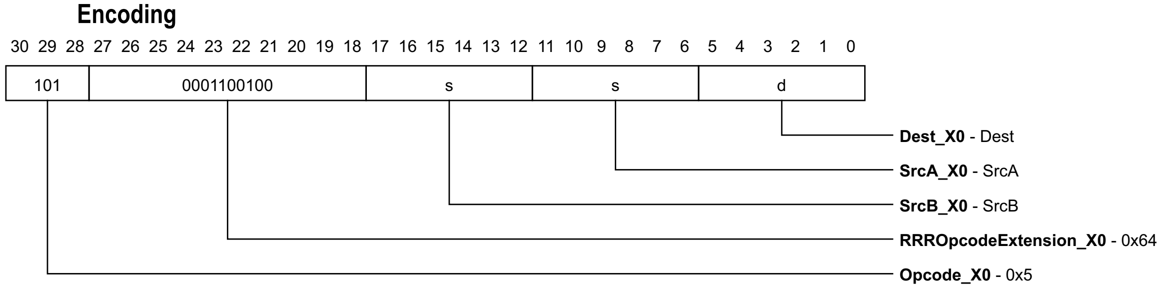

Figure 14-474: v1int_l in X0 Bits Encoding

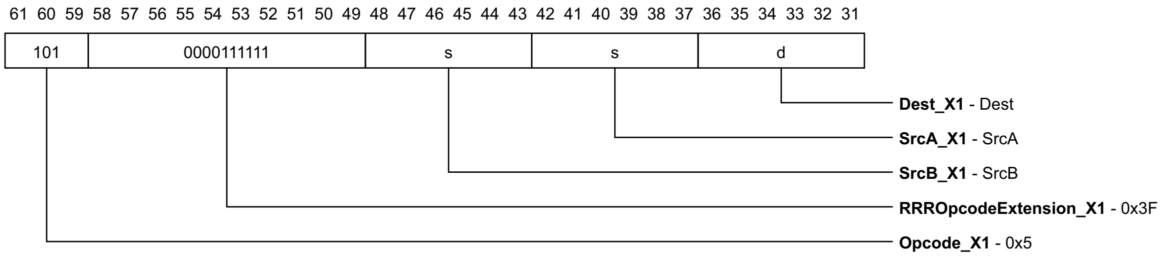

Figure 14-475: v1int_l in X1 Bits Encoding

v1maxu

Vector One Byte Maximum Unsigned

Syntax

```
v1maxu Dest, SrcA, SrcB
```

Example

```
v1maxu r5, r6, r7
```

Description

Set each byte in the destination to the maximum of the corresponding byte in the first source operand and the corresponding byte in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    uint8_t srca = getByte (rf[SrcA], counter);
    uint8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca > srcb) ? srca : srcb));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

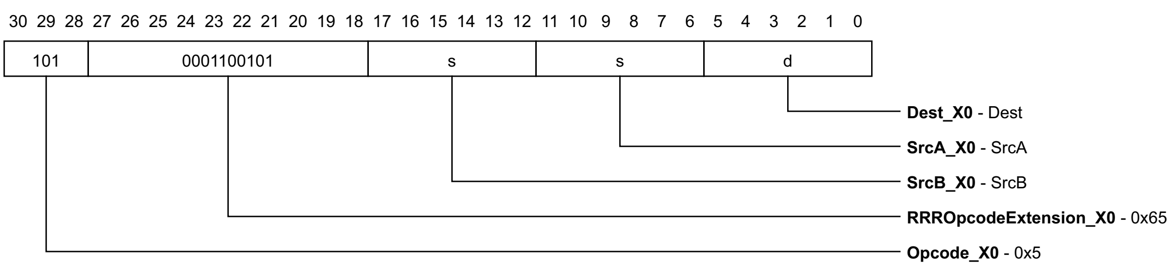

Figure 14-476: v1maxu in X0 Bits Encoding

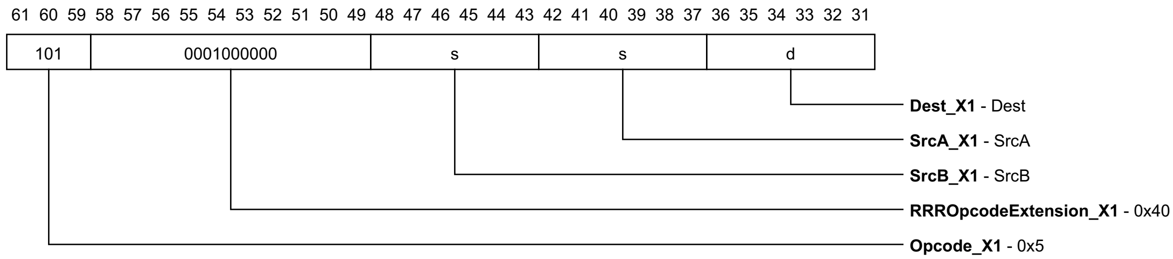

Figure 14-477: v1maxu in X1 Bits Encoding

v1maxui

Vector One Byte Maximum Unsigned Immediate

Syntax

```
v1maxui Dest, SrcA, Imm8
```

Example

```
v1maxui r5, r6, 5
```

Description

Set each byte in the destination to the maximum of the corresponding byte in the first source operand and the sign extended immediate.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
uint8_t immb = Imm8;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    uint8_t srca = getByte (rf[SrcA], counter);
    output = setByte (output, counter, ((srca > immb) ? srca : immb));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

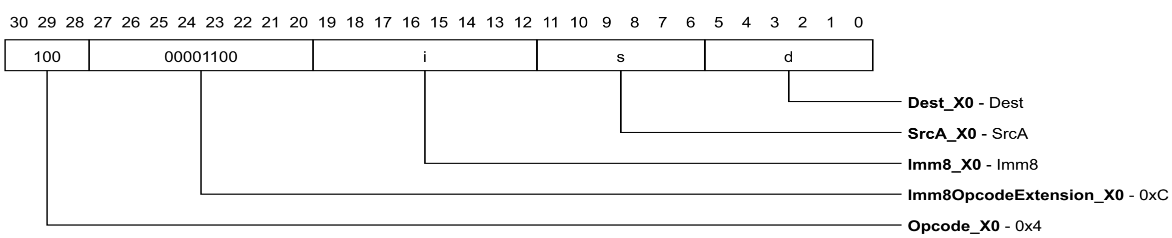

Figure 14-478: v1maxui in X0 Bits Encoding

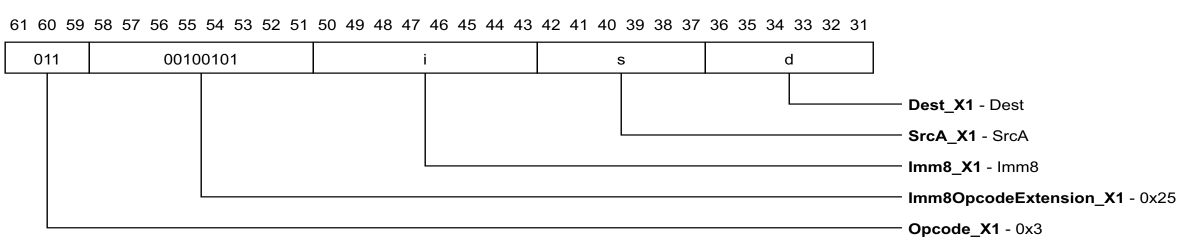

Figure 14-479: v1maxui in X1 Bits Encoding

v1minu

Vector One Byte Minimum Unsigned

Syntax

```
v1minu Dest, SrcA, SrcB
```

Example

```
v1minu r5, r6, r7
```

Description

Set each byte in the destination to the minimum of the corresponding byte in the first source operand and the corresponding byte in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    uint8_t srca = getByte (rf[SrcA], counter);
    uint8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca < srcb) ? srca : srcb));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

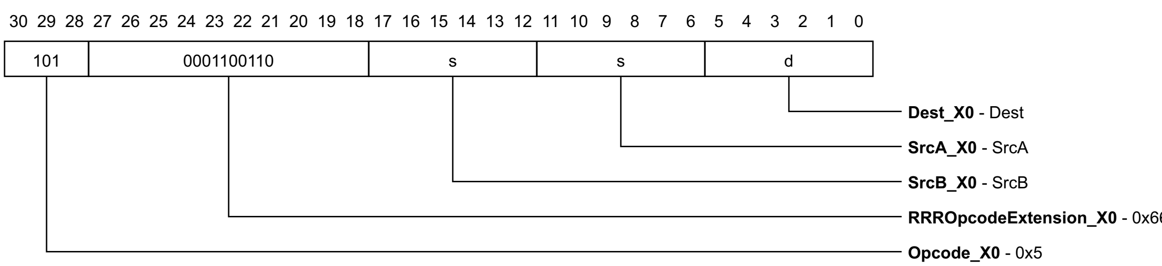

Figure 14-480: v1minu in X0 Bits Encoding

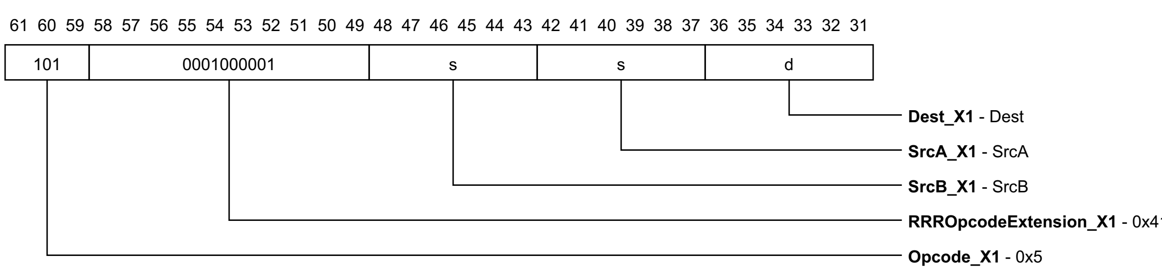

Figure 14-481: v1minu in X1 Bits Encoding

v1minui

Vector One Byte Minimum Unsigned Immediate

Syntax

```
v1minui Dest, SrcA, Imm8
```

Example

```
v1minui r5, r6, 5
```

Description

Set each bytes in the destination to the minimum of the corresponding byte in the first source operand and the sign extended immediate.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
uint8_t immb = Imm8;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    uint8_t srca = getByte (rf[SrcA], counter);
    output = setByte (output, counter, ((srca < immb) ? srca : immb));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

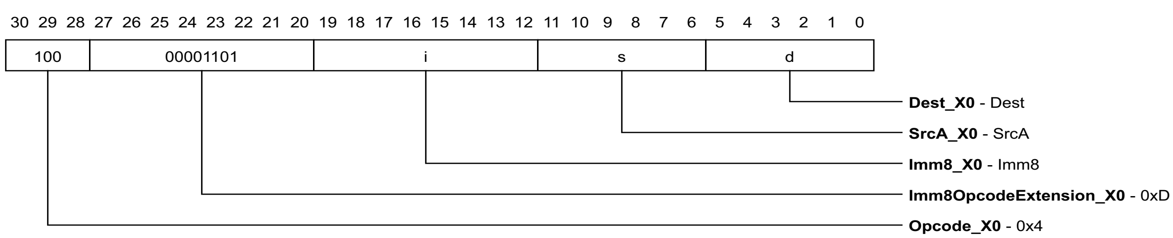

Figure 14-482: v1minui in X0 Bits Encoding

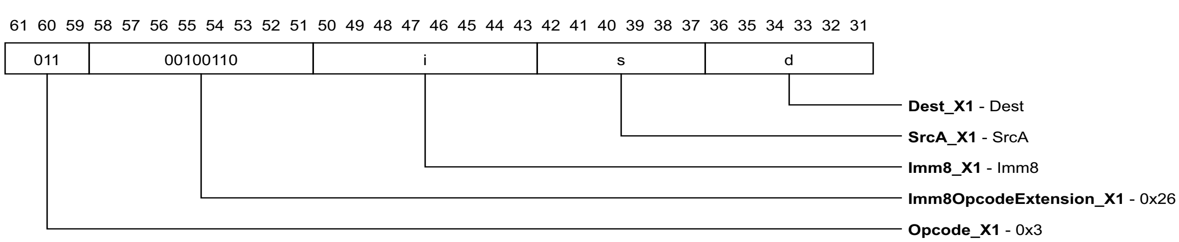

Figure 14-483: v1minui in X1 Bits Encoding

v1mnz

Vector One Byte Mask Not Zero

Syntax

```
v1mnz Dest, SrcA, SrcB
```

Example

```
v1mnz r5, r6, r7
```

Description

Set each byte in the destination to the corresponding byte of the second operand if the corresponding byte of the first operand is not zero, otherwise set it to zero (0).

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca != 0) ? srcb : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

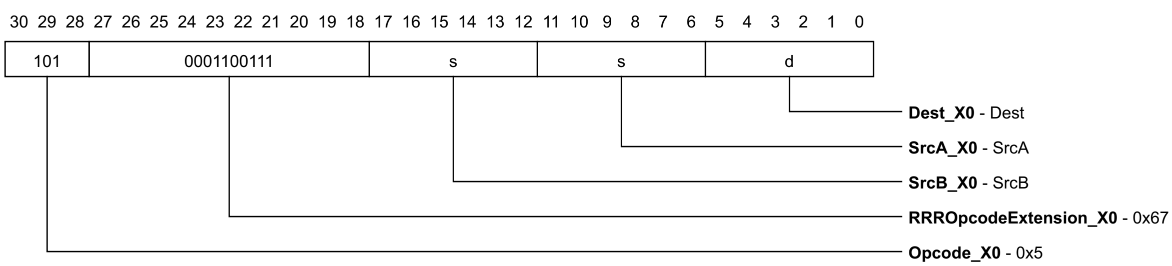

Figure 14-484: v1mnz in X0 Bits Encoding

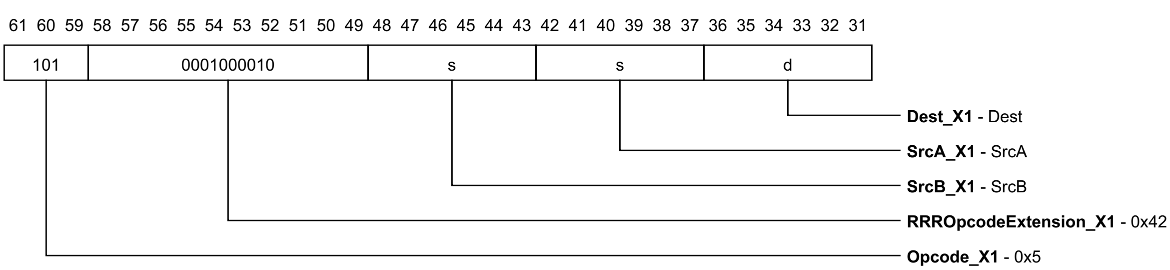

Figure 14-485: v1mnz in X1 Bits Encoding

v1multu

Vector One Byte Multiply and Truncate Unsigned

Syntax

```
v1multu Dest, SrcA, SrcB
```

Example

```
v1multu r5, r6, r7
```

Description

Multiply the eight 8-bit quantities in the first source operand by the eight 8-bit quantities in the second source operand. The result is truncated to the low-order 8 bits of the 16-bit product.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 8); counter++)
  {
    output =
      setByte (output, counter,
       ((uint8_t) getByte (rf[SrcA], counter) *
      (uint8_t) getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

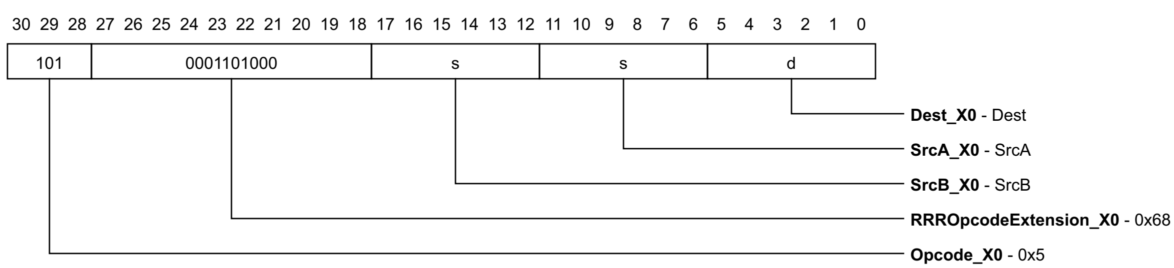

Figure 14-486: v1multu in X0 Bits Encoding

v1mulu

Vector One Byte Multiply Unsigned

Syntax

```
v1mulu Dest, SrcA, SrcB
```

Example

```
v1mulu r5, r6, r7
```

Description

Multiply the four low-order 8-bit quantities in the first source operand by the four low-order 8-bit quantities in the second source operand. The 16-bit results are packed.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       ((uint16_t) getByte (rf[SrcA], counter) *
        (uint16_t) getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

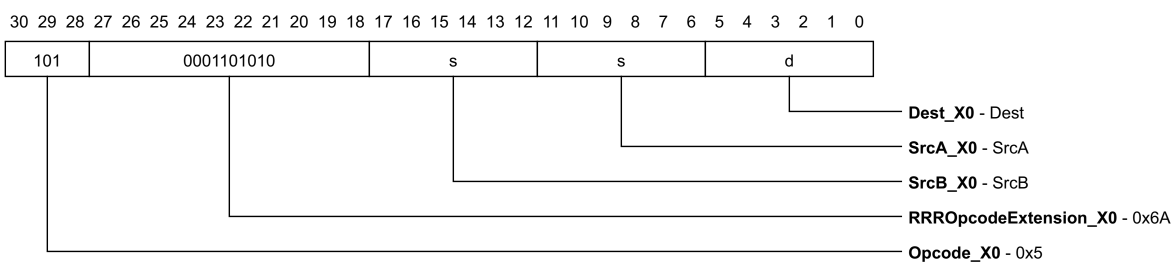

Figure 14-487: v1mulu in X0 Bits Encoding

v1mulus

Vector One Byte Multiply Unsigned Signed

Syntax

```
v1mulus Dest, SrcA, SrcB
```

Example

```
v1mulus r5, r6, r7
```

Description

Multiply the four low-order 8-bit quantities in the first source operand by the four low-order 8-bit quantities in the second source operand. The 16-bit results are packed. The quantities in the first operand are treated as unsigned values, and the quantities in the second operand are treated as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       ((int16_t) getByte (rf[SrcA], counter) *
        (int16_t) signExtend8 (getByte (rf[SrcB], counter))));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

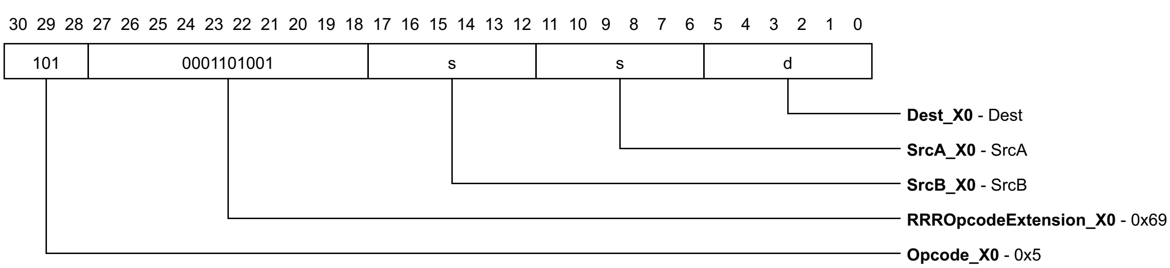

Figure 14-488: v1mulus in X0 Bits Encoding

v1mz

Vector One Byte Mask Zero

Syntax

```
v1mz Dest, SrcA, SrcB
```

Example

```
v1mz r5, r6, r7
```

Description

Set each byte in the destination to the corresponding byte of the second operand if the corresponding byte of the first operand is zero, otherwise set it to zero (0).

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    int8_t srca = getByte (rf[SrcA], counter);
    int8_t srcb = getByte (rf[SrcB], counter);
    output = setByte (output, counter, ((srca == 0) ? srcb : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

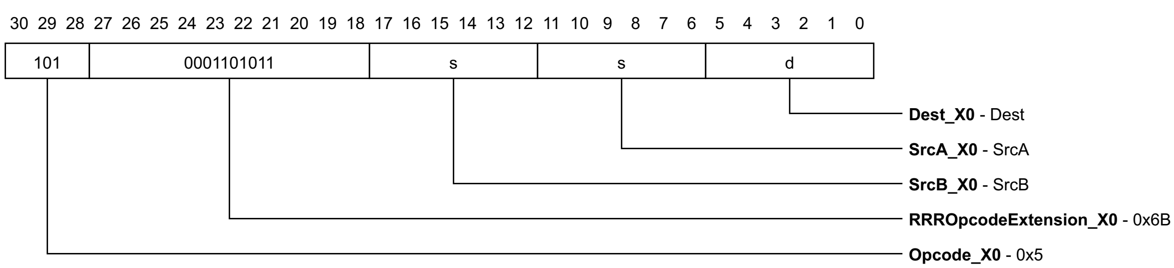

Figure 14-489: v1mz in X0 Bits Encoding

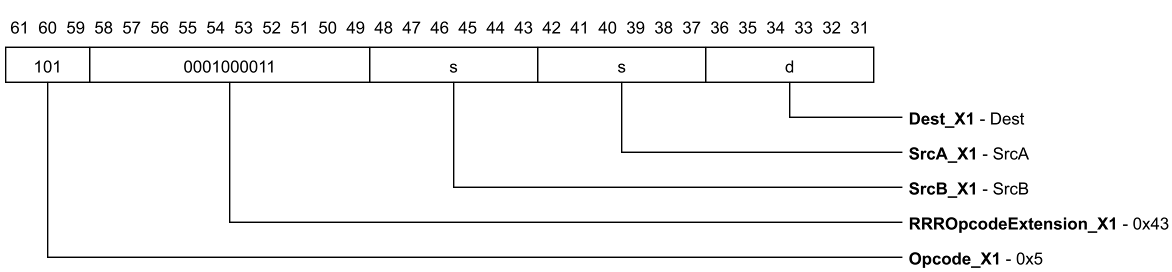

Figure 14-490: v1mz in X1 Bits Encoding

v1sadau

Vector One Byte Sum of Absolute Difference Accumulate Unsigned

Syntax

```
v1sadau Dest, SrcA, SrcB
```

Example

```
v1sadau r5, r6, r7
```

Description

Sum the absolute differences between the eight bytes in the first source operand and the eight bytes in the second source operand and accumulate the sum into the destination register.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output += abs (getByte (rf[SrcA], counter) - getByte (rf[SrcB], counter));
  }

rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

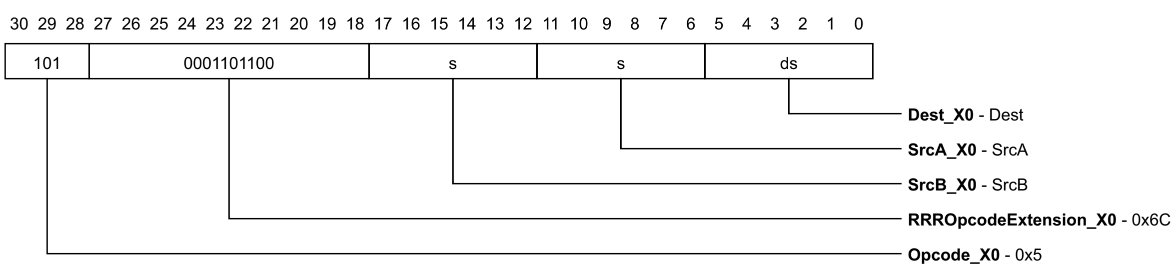

Figure 14-491: v1sadau in X0 Bits Encoding

v1sadu

Vector One Byte Sum of Absolute Difference Unsigned

Syntax

```
v1sadu Dest, SrcA, SrcB
```

Example

```
v1sadu r5, r6, r7
```

Description

Sum the absolute differences between the eight bytes in the first source operand and the eight bytes in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output += abs (getByte (rf[SrcA], counter) - getByte (rf[SrcB], counter));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

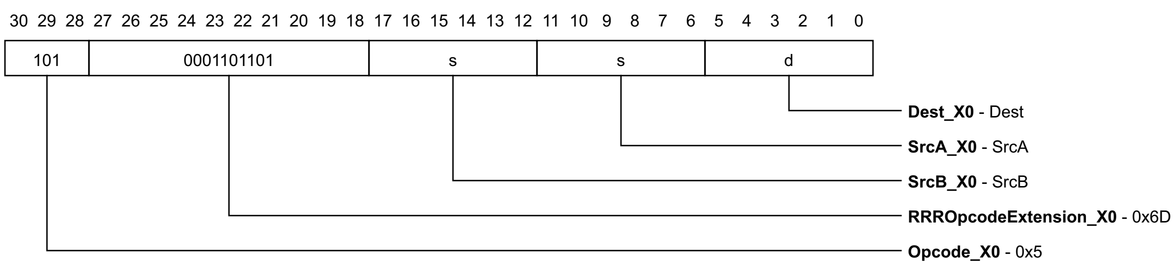

Figure 14-492: v1sadu in X0 Bits Encoding

v1shl

Vector One Byte Shift Left

Syntax

```
v1shl Dest, SrcA, SrcB
```

Example

```
v1shl r5, r6, r7
```

Description

Logically shift each of the eight bytes in the first source operand to the left by the second source operand. The effective shift amount is the specified operand modulo the number of bits in a byte. Logical left shift shifts zeros into the low ordered bits in a byte.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       (getByte (rf[SrcA], counter) <<
       (((UnsignedMachineWord) rf[SrcB]) % BYTE_SIZE)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

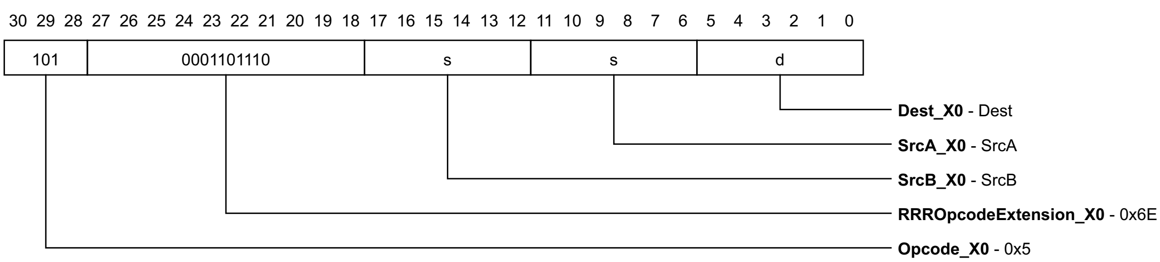

Figure 14-493: v1shl in X0 Bits Encoding

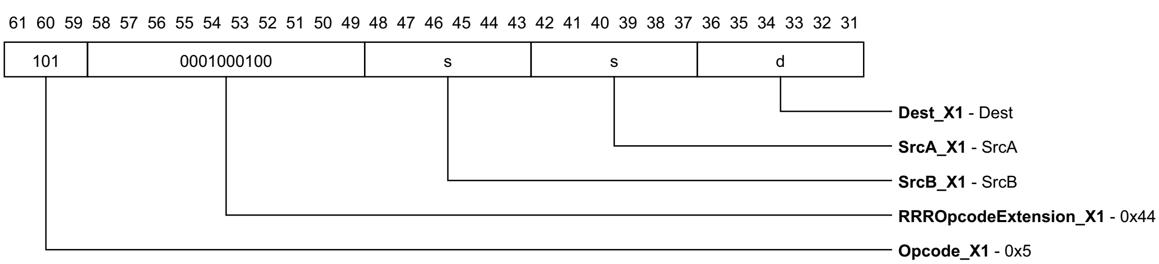

Figure 14-494: v1shl in X1 Bits Encoding

v1shli

Vector One Byte Shift Left Immediate

Syntax

```
v1shli Dest, SrcA, ShAmt
```

Example

```
v1shli r5, r6, 5
```

Description

Logically shift each of the eight bytes in the first source operand to the left by an immediate. The effective shift amount is the specified immediate modulo the number of bits in a byte. Left shifts shift zeros into the low ordered bits in a byte and are suitable to be used as unsigned multiplication by powers of two.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       (getByte (rf[SrcA], counter) <<
       (((UnsignedMachineWord) ShAmt) % BYTE_SIZE)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

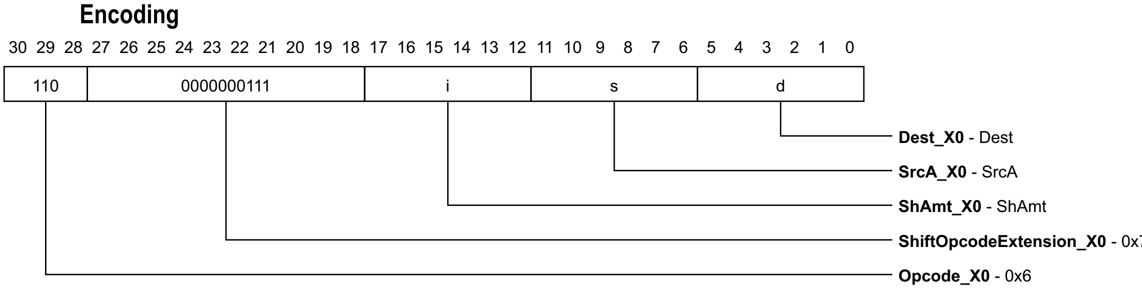

Figure 14-495: v1shli in X0 Bits Encoding

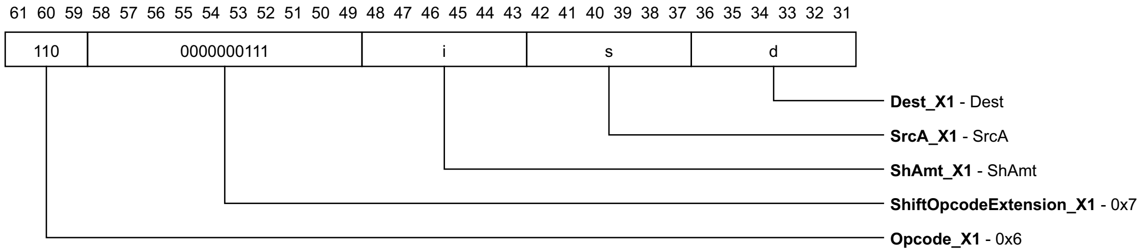

Figure 14-496: v1shli in X1 Bits Encoding

v1shrs

Vector One Byte Shift Right Signed

Syntax

```
v1shrs Dest, SrcA, SrcB
```

Example

```
v1shrs r5, r6, r7
```

Description

Arithmetically shift each of the eight bytes in the first source operand to the right by the second source operand. The effective shift amount is the specified operand modulo the number of bits in a byte. Arithmetic right shift shifts the high ordered bit into the high ordered bits in a byte.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       (signExtend8 (getByte (rf[SrcA], counter)) >>
       (((UnsignedMachineWord) rf[SrcB]) % BYTE_SIZE)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

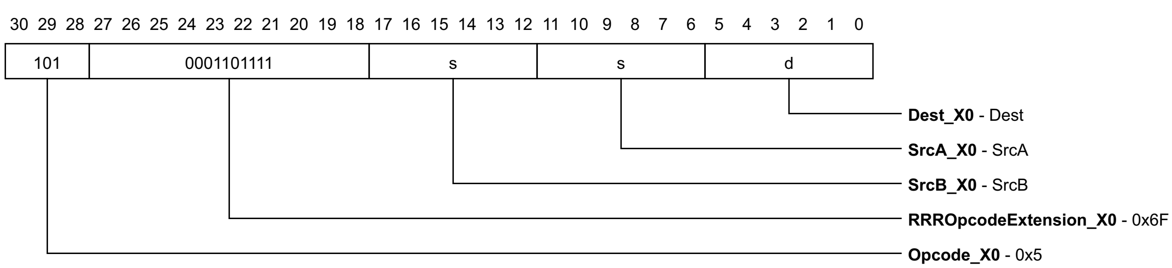

Figure 14-497: v1shrs in X0 Bits Encoding

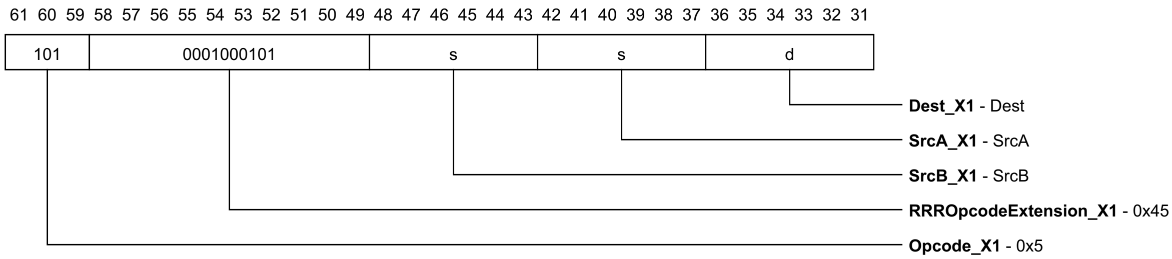

Figure 14-498: v1shrs in X1 Bits Encoding

v1shrsi

Vector One Byte Shift Right Signed Immediate

Syntax

```
v1shrsi Dest, SrcA, ShAmt
```

Example

```
v1shrsi r5, r6, 5
```

Description

Arithmetically shift each of the eight bytes in the first source operand to the right by an immediate. The effective shift amount is the specified immediate modulo the number of bits in a byte. Arithmetic right shifts shift the high ordered bit into the high ordered bits in a byte.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       (signExtend8 (getByte (rf[SrcA], counter)) >>
       (((UnsignedMachineWord) ShAmt) % BYTE_SIZE)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

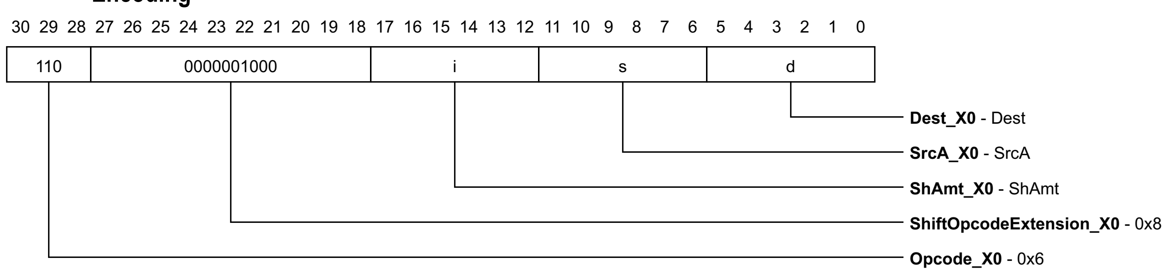

Figure 14-499: v1shrsi in X0 Bits Encoding

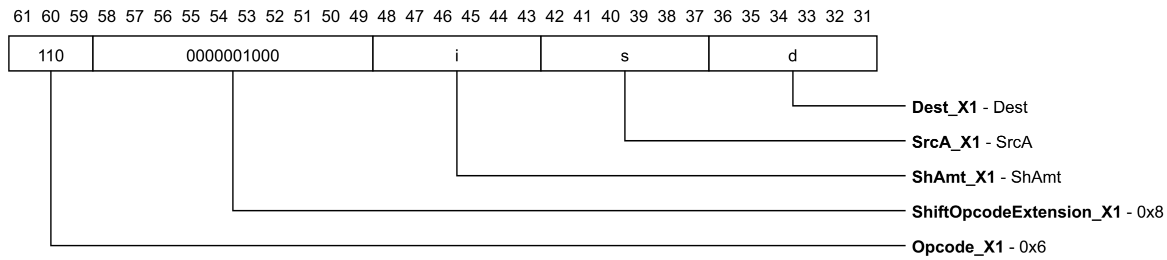

Figure 14-500: v1shrsi in X1 Bits Encoding

v1shru

Vector One Byte Shift Right Unsigned

Syntax

```
v1shru Dest, SrcA, SrcB
```

Example

```
v1shru r5, r6, r7
```

Description

Logically shift each of the eight bytes in the first source operand to the right by the second source operand. The effective shift amount is the specified operand modulo the number of bits in a byte. Logical right shift shifts zeros into the high ordered bits in a byte.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       (getByte (rf[SrcA], counter) >>
       (((UnsignedMachineWord) rf[SrcB]) % BYTE_SIZE)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

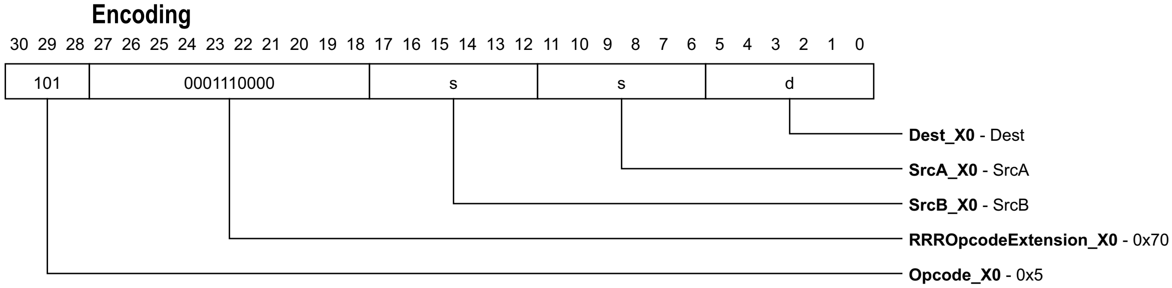

Figure 14-501: v1shru in X0 Bits Encoding

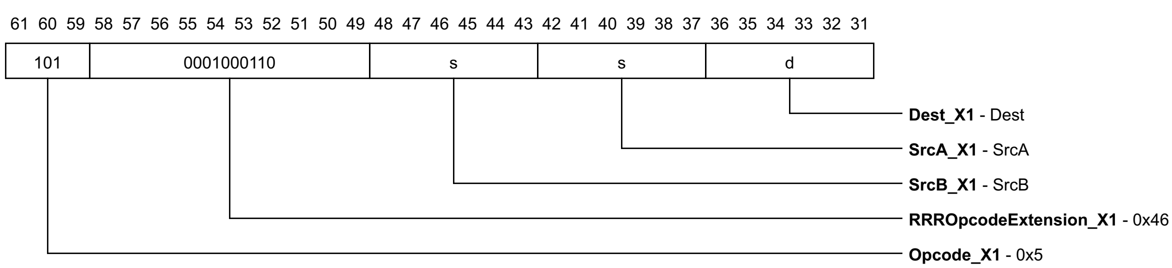

Figure 14-502: v1shru in X1 Bits Encoding

v1shrui

Vector One Byte Shift Right Unsigned Immediate

Syntax

```
v1shrui Dest, SrcA, ShAmt
```

Example

```
v1shrui r5, r6, 5
```

Description

Logically shift each of the eight bytes in the first source operand to the right by an immediate. The effective shift amount is the specified immediate modulo the number of bits in a byte. Logical right shifts shift zeros into the high ordered bits in a byte and are suitable to be used as unsigned integer division by powers of two.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       (getByte (rf[SrcA], counter) >>
       (((UnsignedMachineWord) ShAmt) % BYTE_SIZE)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

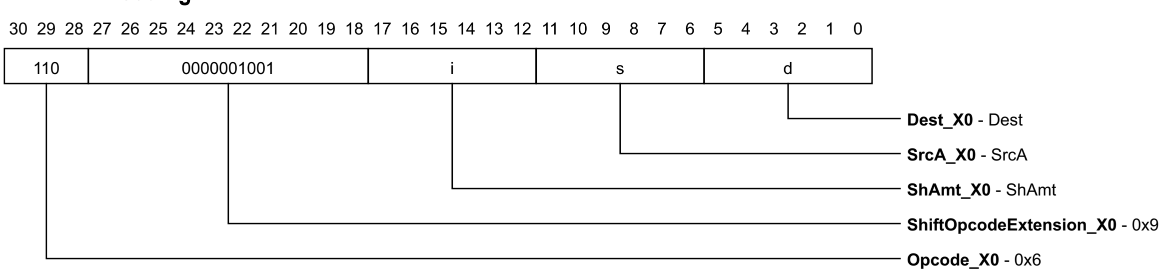

Figure 14-503: v1shrui in X0 Bits Encoding

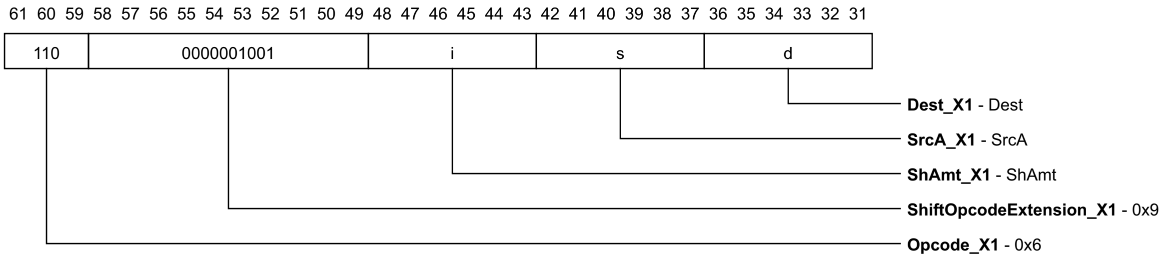

Figure 14-504: v1shrui in X1 Bits Encoding

v1sub

Vector One Byte Subtract

Syntax

```
v1sub Dest, SrcA, SrcB
```

Example

```
v1sub r5, r6, r7
```

Description

Subtract the eight bytes in the second source operand from the eight bytes in the first source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
        (getByte (rf[SrcA], counter) - getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

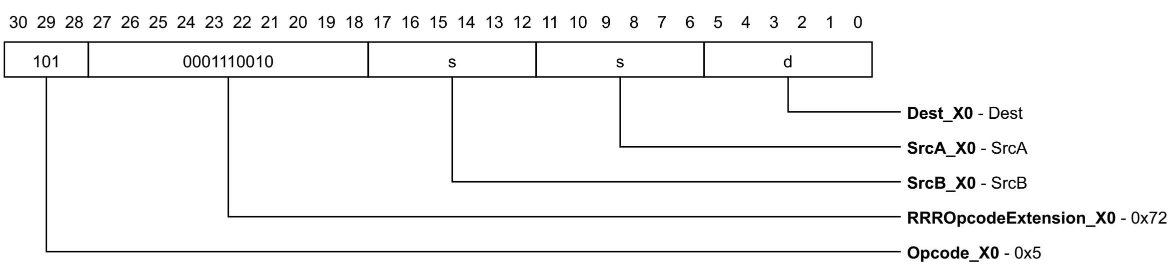

Figure 14-505: v1sub in X0 Bits Encoding

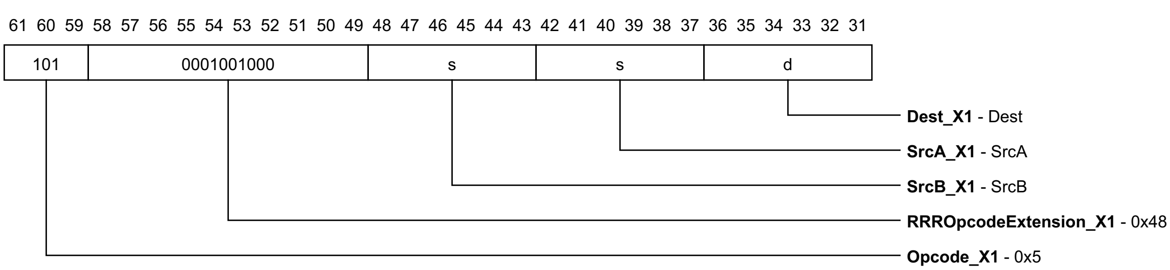

Figure 14-506: v1sub in X1 Bits Encoding

v1subuc

Vector One Byte Subtract Unsigned Clamped

Syntax

```
v1subuc Dest, SrcA, SrcB
```

Example

```
v1subuc r5, r6, r7
```

Description

Subtract the eight bytes in the second source operand from the eight bytes in the first source operand and clamp each result to 0 or the maximum positive value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    output =
      setByte (output, counter,
       unsigned_saturate8 (getByte (rf[SrcA], counter) -
                           getByte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

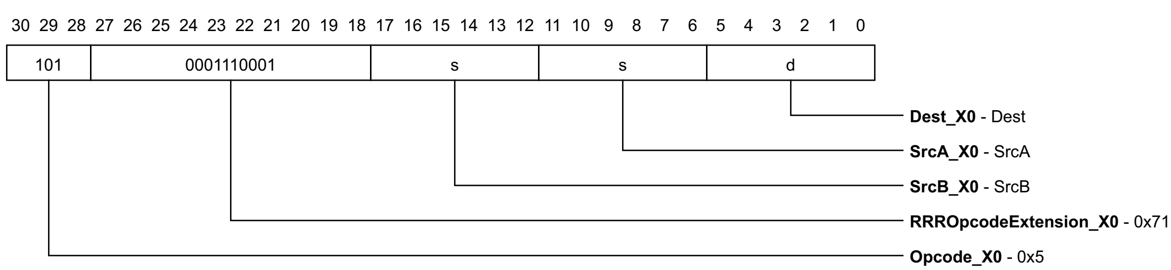

Figure 14-507: v1subuc in X0 Bits Encoding

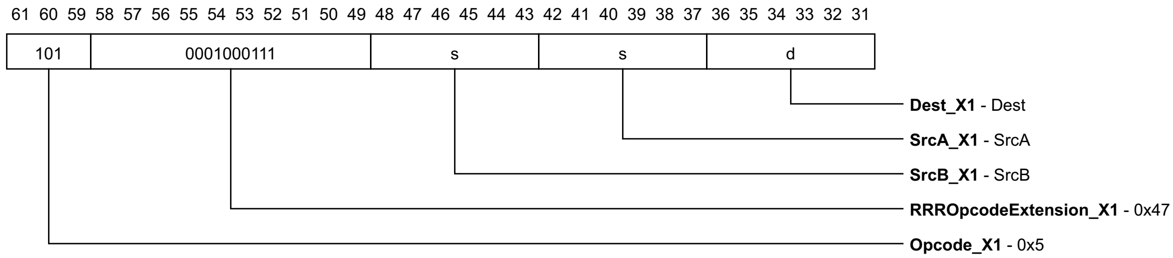

Figure 14-508: v1subuc in X1 Bits Encoding

v2add

Vector Two Byte Add

Syntax

```
v2add Dest, SrcA, SrcB
```

Example

```
v2add r5, r6, r7
```

Description

Add the four 16-bit quantities in the first source operand to the four 16-bit quantities in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (get2Byte (rf[SrcA], counter) +
        get2Byte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

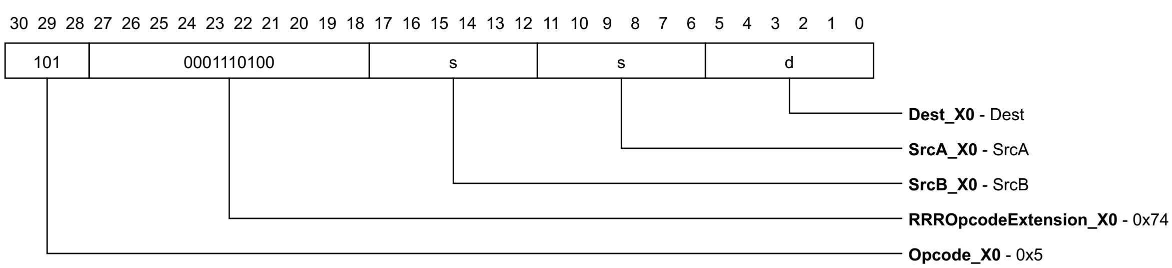

Figure 14-509: v2add in X0 Bits Encoding

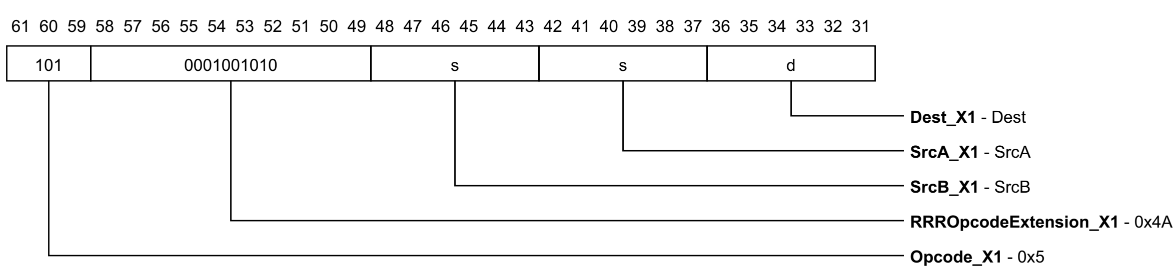

Figure 14-510: v2add in X1 Bits Encoding

v2addi

Vector Two Byte Add Immediate

Syntax

```
v2addi Dest, SrcA, Imm8
```

Example

```
v2addi r5, r6, 5
```

Description

Add a sign extended immediate to both of the 16-bit quantities in the source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (get2Byte (rf[SrcA], counter) + signExtend8 (Imm8)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

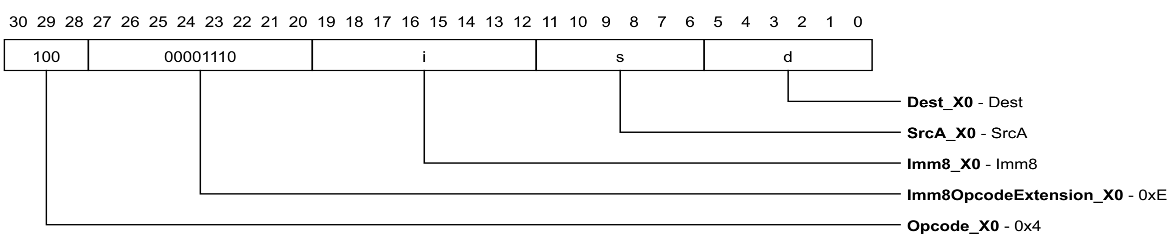

Figure 14-511: v2addi in X0 Bits Encoding

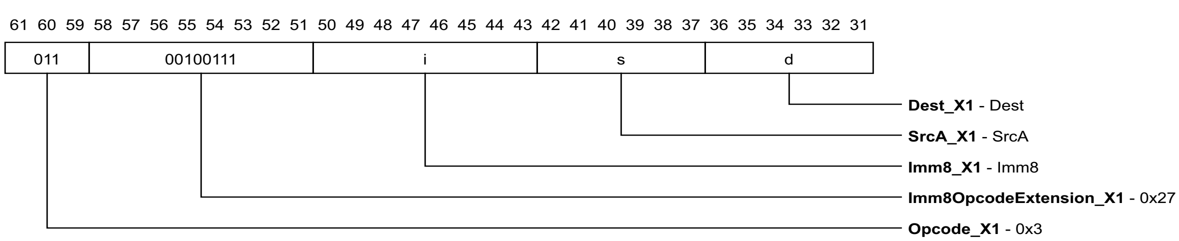

Figure 14-512: v2addi in X1 Bits Encoding

v2addsc

Vector Two Byte Add Signed Clamped

Syntax

```
v2addsc Dest, SrcA, SrcB
```

Example

```
v2addsc r5, r6, r7
```

Description

Add the four 16-bit quantities in the first source operand to the four 16-bit quantities in the second source operand and clamp each result to the minimum negative or maximum positive 16-bit value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       signed_saturate16 (signExtend16 (get2Byte (rf[SrcA], counter))
                        +
                        signExtend16 (get2Byte
                                    (rf[SrcB], counter))));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

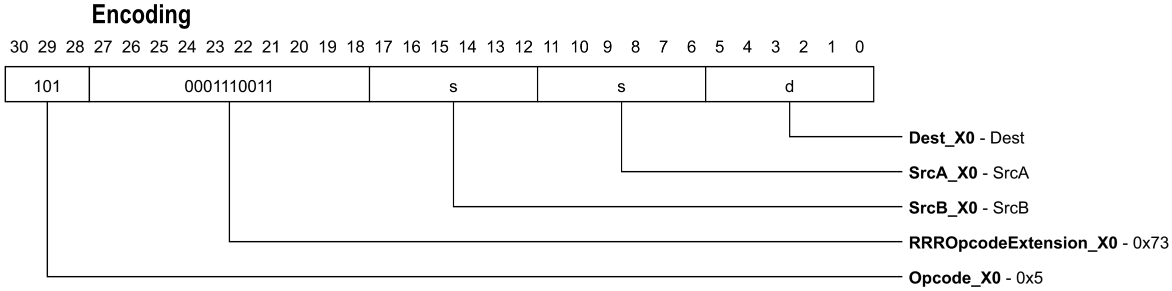

Figure 14-513: v2addsc in X0 Bits Encoding

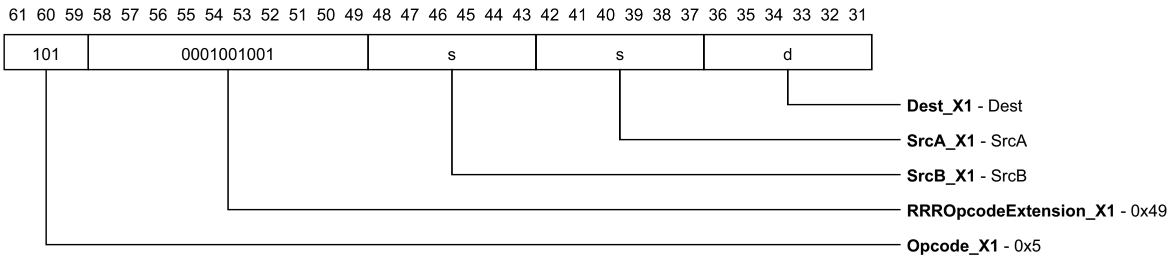

Figure 14-514: v2addsc in X1 Bits Encoding

v2adiffs

Vector Two Byte Absolute Difference Signed

Syntax

```
v2adiffs Dest, SrcA, SrcB
```

Example

```
v2adiffs r5, r6, r7
```

Description

Compute the absolute differences between the four 16-bit quantities in the first source operand and the four 16-bit quantities in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       abs (signExtend16 (get2Byte (rf[SrcA], counter)) -
            signExtend16 (get2Byte (SrcB, counter))));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

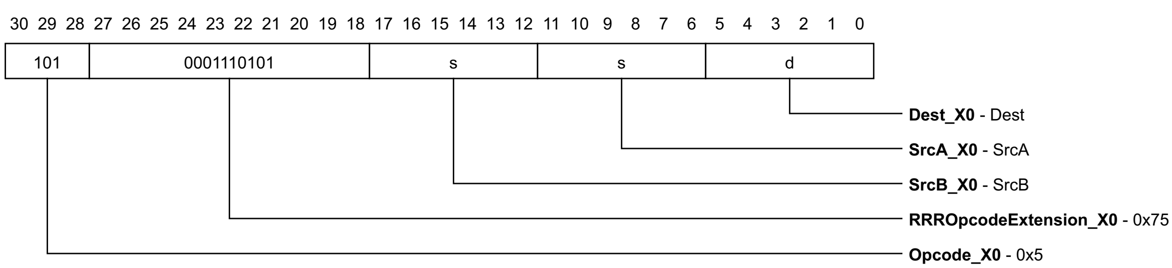

Figure 14-515: v2adiffs in X0 Bits Encoding

v2avgs

Vector Two Byte Average Signed

Syntax

```
v2avgs Dest, SrcA, SrcB
```

Example

```
v2avgs r5, r6, r7
```

Description

Compute the average between the four 16-bit quantities in the first source operand and the four 16-bit quantities in the second source operand, rounding upwards.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    SignedMachineWord srca = signExtend16 (get2Byte (rf[SrcA], counter));
    SignedMachineWord srcb = signExtend16 (get2Byte (rf[SrcB], counter));
    output = set2Byte (output, counter, ((srca + srcb + 1) >> 1));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

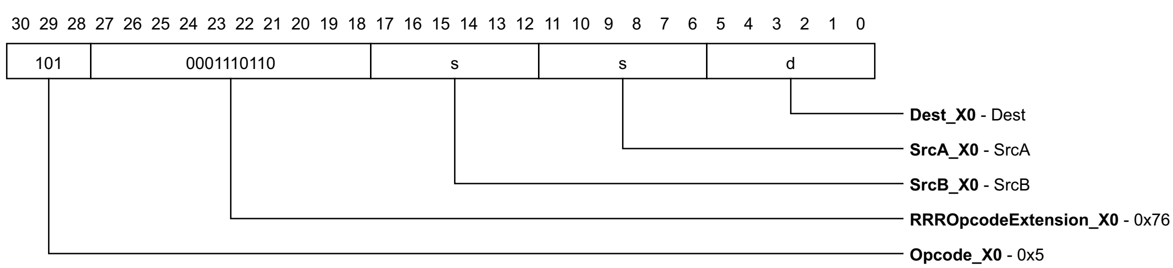

Figure 14-516: v2avgs in X0 Bits Encoding

v2cmpeq

Vector Two Byte Set Equal To

Syntax

```
v2cmpeq Dest, SrcA, SrcB
```

Example

```
v2cmpeq r5, r6, r7
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is equal to the corresponding 16-bit quantity of the second source operand. Otherwise the result is set to 0.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca == srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

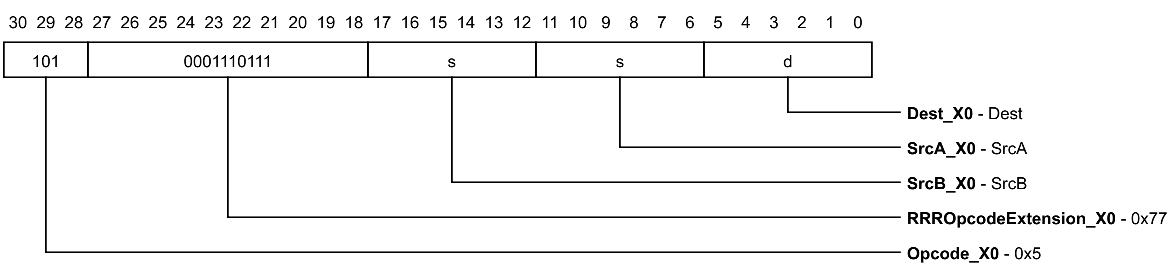

Figure 14-517: v2cmpeq in X0 Bits Encoding

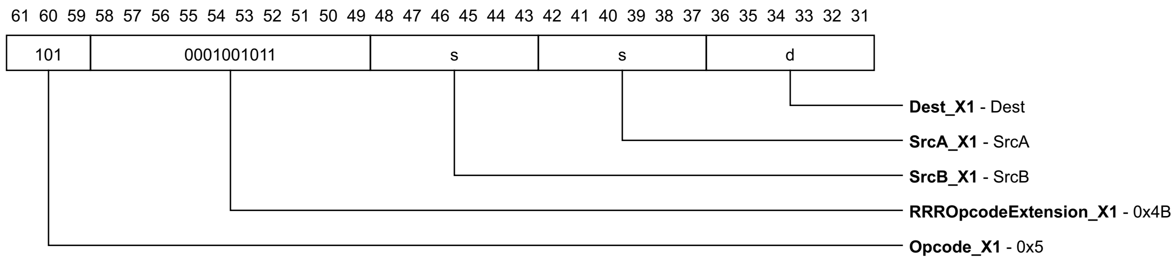

Figure 14-518: v2cmpeq in X1 Bits Encoding

v2cmpeqi

Vector Two Byte Set Equal To Immediate

Syntax

```
v2cmpeqi Dest, SrcA, Imm8
```

Example

```
v2cmpeqi r5, r6, 5
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is equal to a sign extended immediate. Otherwise the result is set to 0.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = signExtend8 (Imm8);
    output = set2Byte (output, counter, ((srca == srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

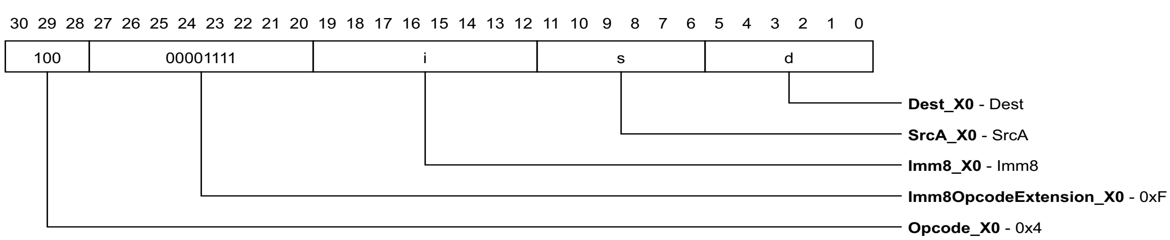

Figure 14-519: v2cmpeqi in X0 Bits Encoding

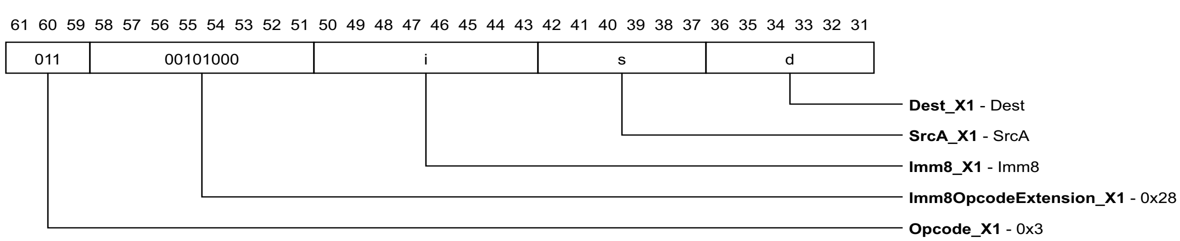

Figure 14-520: v2cmpeqi in X1 Bits Encoding

v2cmples

Vector Two Byte Set Less Than or Equal

Syntax

```
v2cmples Dest, SrcA, SrcB
```

Example

```
v2cmples r5, r6, r7
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is less than or equal to the corresponding 16-bit quantity of the second source operand. Otherwise the result is set to 0. This instruction treats both source 16-bit quantities as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca <= srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

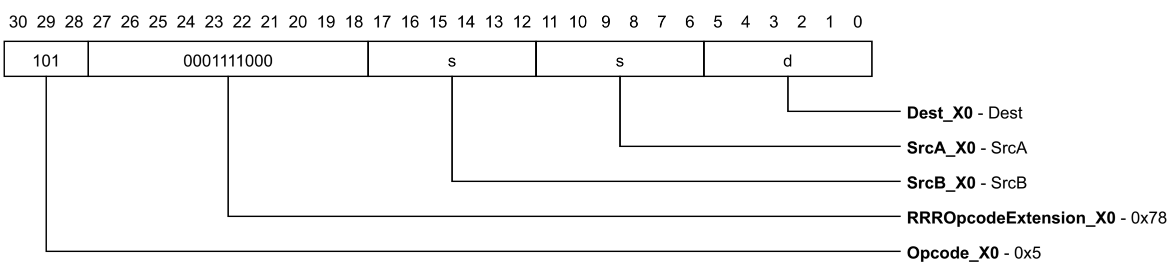

Figure 14-521: v2cmples in X0 Bits Encoding

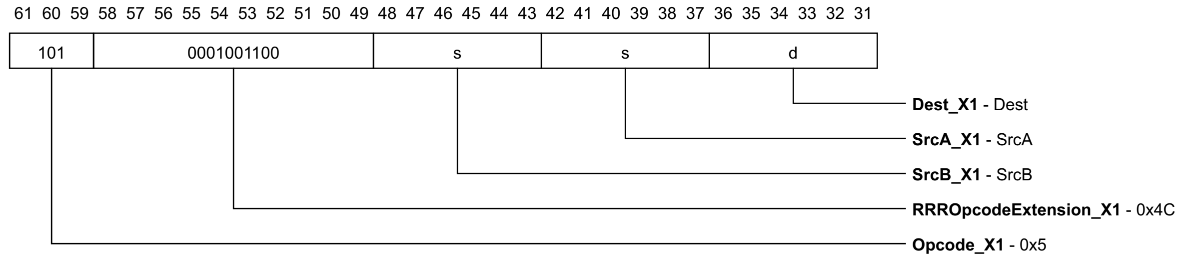

Figure 14-522: v2cmples in X1 Bits Encoding

v2cmpleu

Vector Two Byte Set Less Than or Equal Unsigned

Syntax

```
v2cmpleu Dest, SrcA, SrcB
```

Example

```
v2cmpleu r5, r6, r7
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is less than or equal to the corresponding 16-bit quantity of the second source operand. Otherwise the result is set to 0. This instruction treats both source 16-bit quantity as unsigned values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    uint16_t srca = get2Byte (rf[SrcA], counter);
    uint16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca <= srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

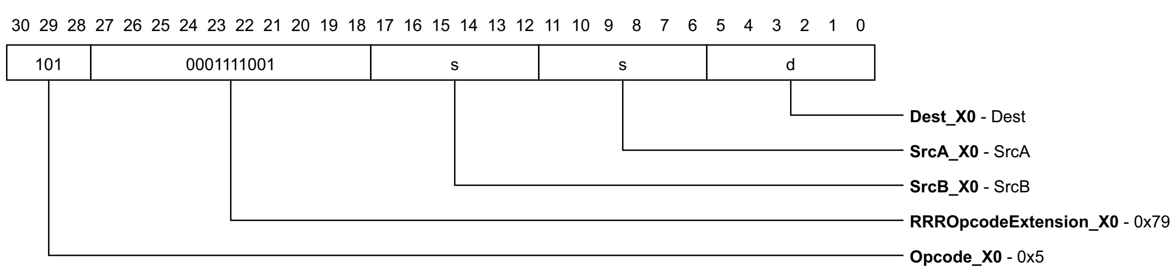

Figure 14-523: v2cmpleu in X0 Bits Encoding

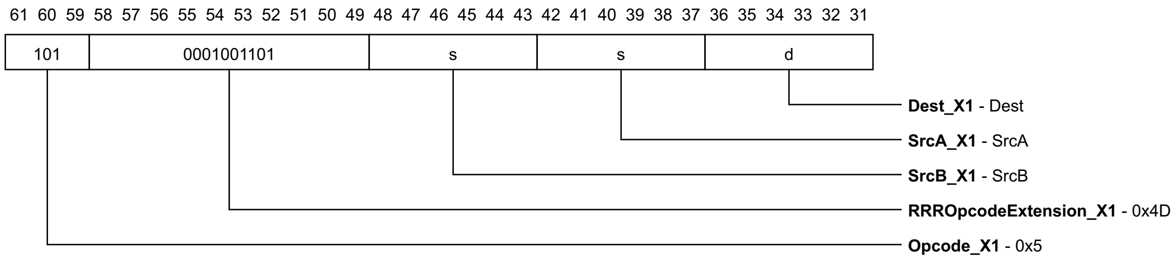

Figure 14-524: v2cmpleu in X1 Bits Encoding

v2cmplts

Vector Two Byte Set Less Than

Syntax

```
v2cmplts Dest, SrcA, SrcB
```

Example

```
v2cmplts r5, r6, r7
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is less than the corresponding 16-bit quantity of the second source operand. Otherwise the result is set to 0. This instruction treats both source 16-bit quantities as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

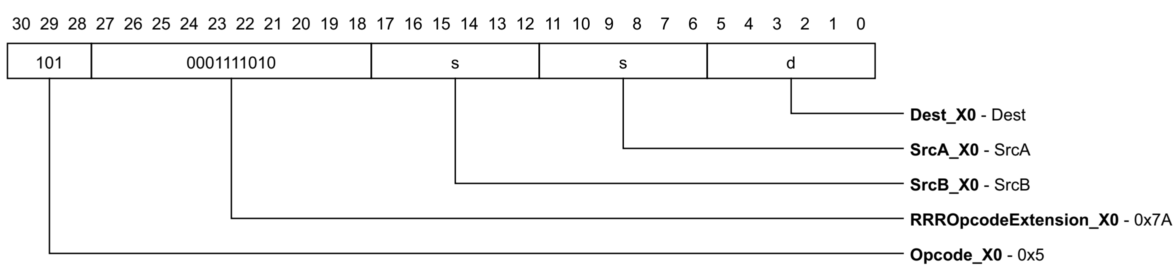

Figure 14-525: v2cmplts in X0 Bits Encoding

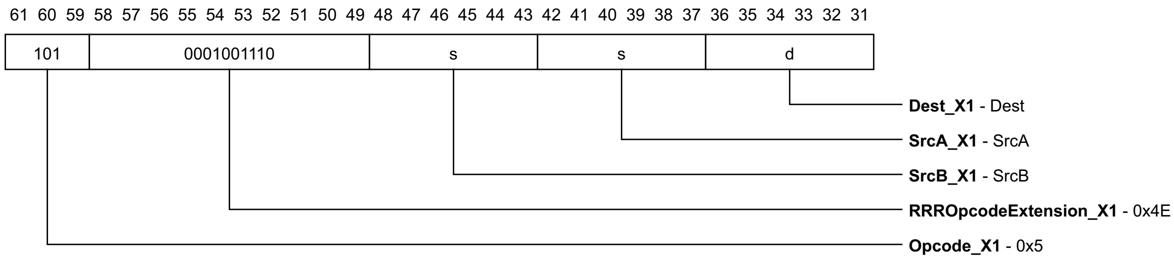

Figure 14-526: v2cmplts in X1 Bits Encoding

v2cmpltsi

Vector Two Byte Set Less Than Immediate

Syntax

```
v2cmpltsi Dest, SrcA, Imm8
```

Example

```
v2cmpltsi r5, r6, 5
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats the first source operand as 16-bit quantities as signed values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = signExtend8 (Imm8);
    output = set2Byte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

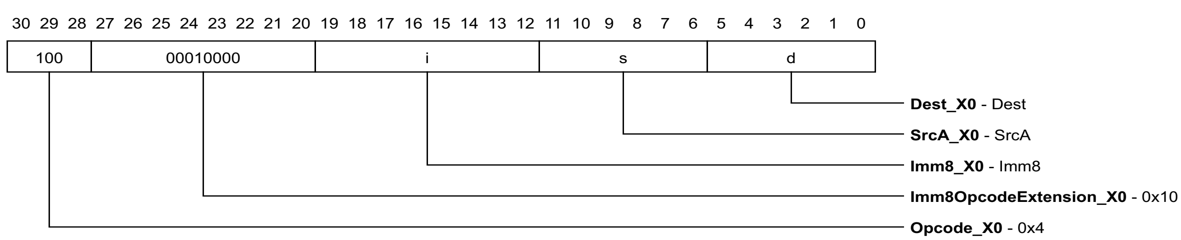

Figure 14-527: v2cmpltsi in X0 Bits Encoding

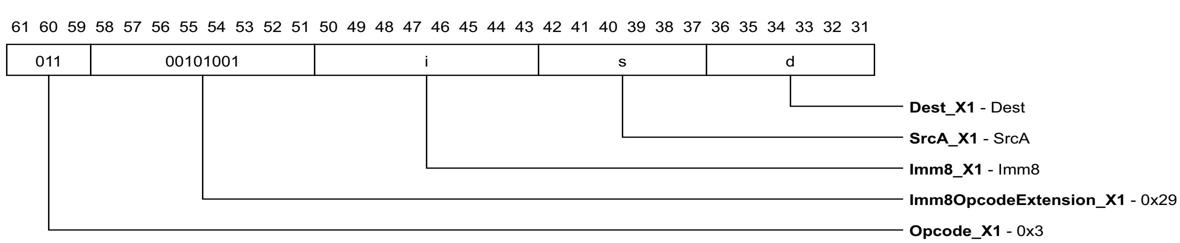

Figure 14-528: v2cmpltsi in X1 Bits Encoding

v2cmpltu

Vector Two Byte Set Less Than Unsigned

Syntax

```
v2cmpltu Dest, SrcA, SrcB
```

Example

```
v2cmpltu r5, r6, r7
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is less than the corresponding 16-bit quantity of the second source operand. Otherwise the result is set to 0. This instruction treats both source 16-bit quantities as unsigned values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    uint16_t srca = get2Byte (rf[SrcA], counter);
    uint16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

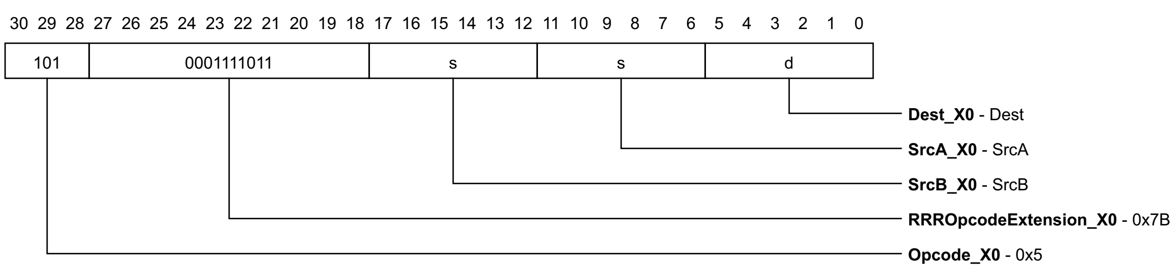

Figure 14-529: v2cmpltu in X0 Bits Encoding

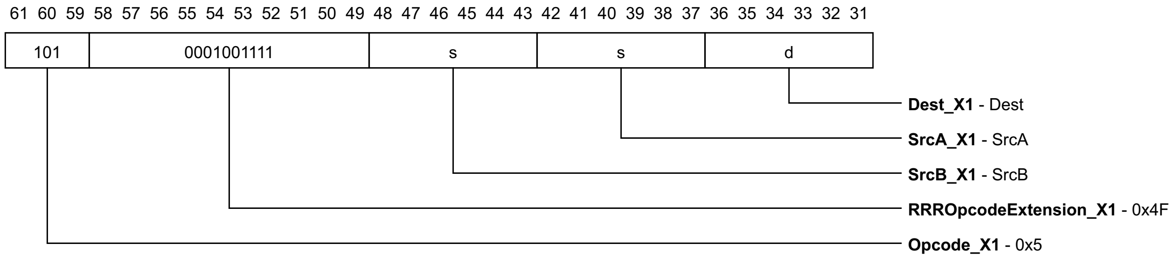

Figure 14-530: v2cmpltu in X1 Bits Encoding

v2cmpltui

Vector Two Byte Set Less Than Unsigned Immediate

Syntax

```
v2cmpltui Dest, SrcA, Imm8
```

Example

```
v2cmpltui r5, r6, 5
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats the first source operand 16-bit quantities as unsigned values.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    uint16_t srca = get2Byte (rf[SrcA], counter);
    uint16_t srcb = signExtend8 (Imm8);
    output = set2Byte (output, counter, ((srca < srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

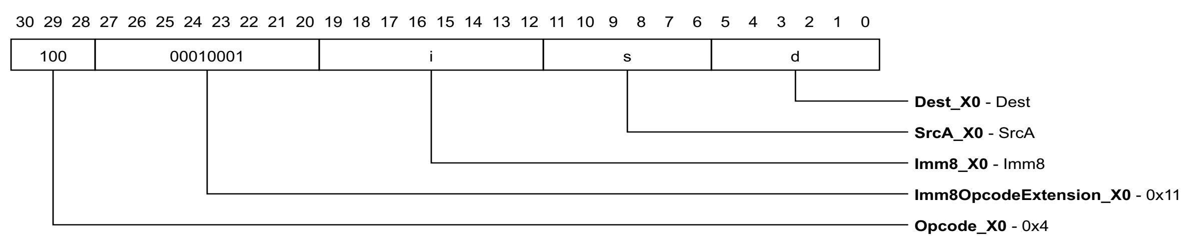

Figure 14-531: v2cmpltui in X0 Bits Encoding

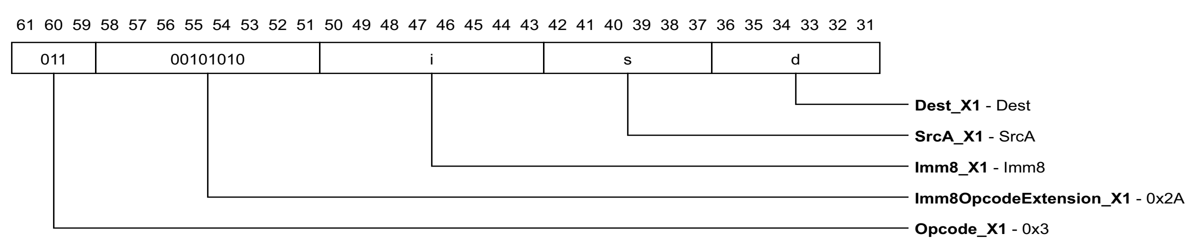

Figure 14-532: v2cmpltui in X1 Bits Encoding

v2cmpne

Vector Two Byte Set Not Equal To

Syntax

```
v2cmpne Dest, SrcA, SrcB
```

Example

```
v2cmpne r5, r6, r7
```

Description

Sets each result 16-bit quantity to one if the corresponding 16-bit quantity of the first source operand is not equal to the 16-bit quantity of the second source operand. Otherwise the result is set to 0.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca != srcb) ? 1 : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

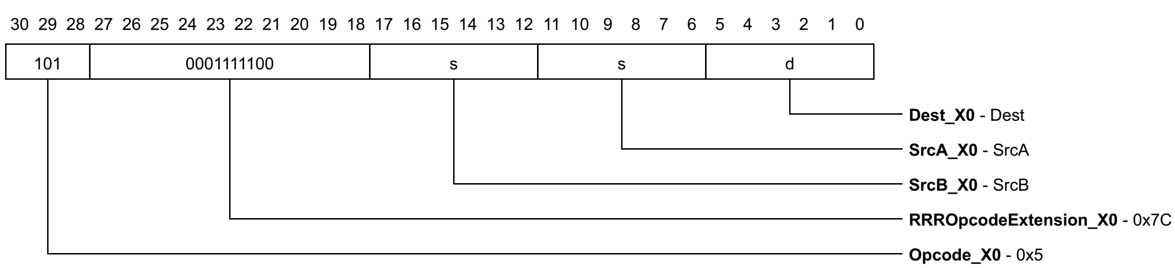

Figure 14-533: v2cmpne in X0 Bits Encoding

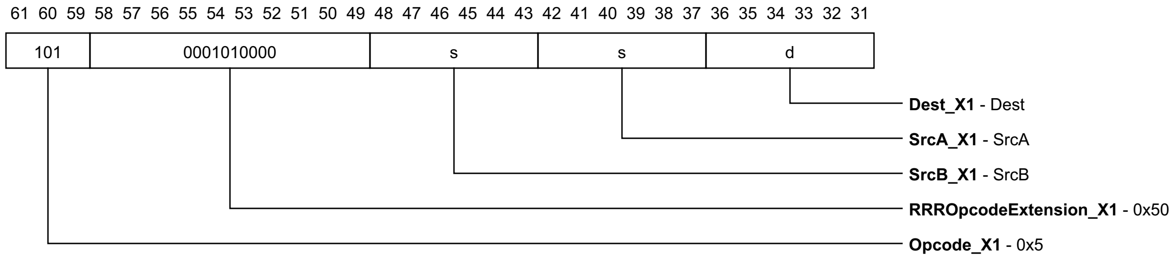

Figure 14-534: v2cmpne in X1 Bits Encoding

v2dotp

Vector Two Byte Dot Product

Syntax

```
v2dotp Dest, SrcA, SrcB
```

Example

```
v2dotp r5, r6, r7
```

Description

Multiply the four 16-bit quantities in the first source operand by the four 16-bit quantities in the second source operand and sum the products.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output +=
      ((int32_t) signExtend16 (get2Byte (rf[SrcA], counter)) *
       (int32_t) signExtend16 (get2Byte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

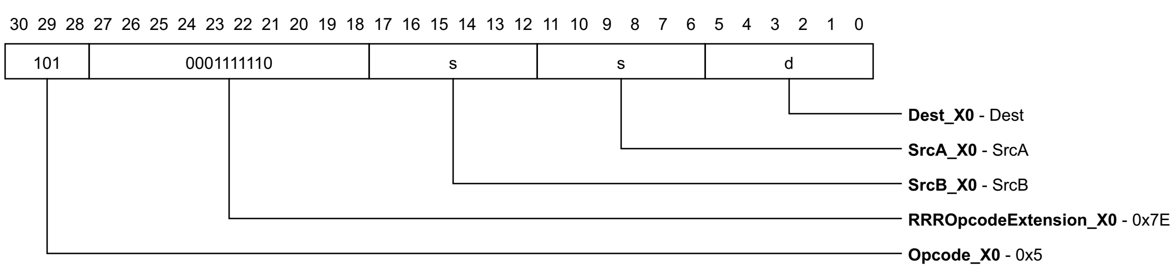

Figure 14-535: v2dotp in X0 Bits Encoding

v2dotpa

Vector Two Byte Dot Product and Accumulate

Syntax

```
v2dotpa Dest, SrcA, SrcB
```

Example

```
v2dotpa r5, r6, r7
```

Description

Multiply the four 16-bit quantities in the first source operand by the four 16-bit quantities in the second source operand and sum the products and the destination operand.

Functional Description

```
int64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output +=
      ((int32_t) signExtend16 (get2Byte (rf[SrcA], counter)) *
       (int32_t) signExtend16 (get2Byte (rf[SrcB], counter)));
  }

rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

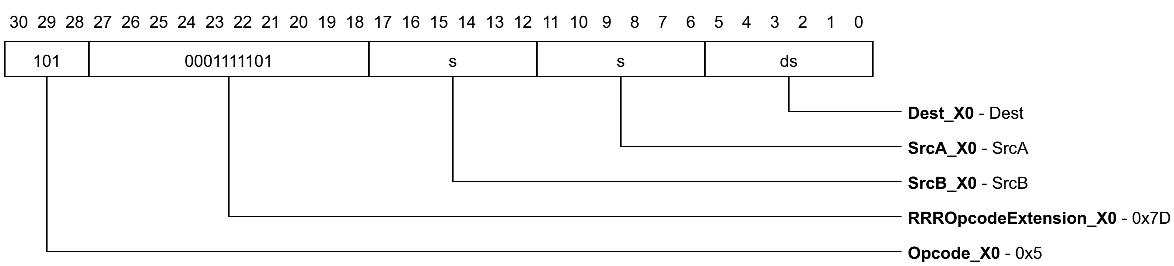

Figure 14-536: v2dotpa in X0 Bits Encoding

v2int_h

Vector Two Byte Interleave High

Syntax

```
v2int_h Dest, SrcA, SrcB
```

Example

```
v2int_h r5, r6, r7
```

Description

Interleave the two high-order 16-bit quantities of the first operand with the two high-order 16-bit quantities of the second operand. The high-order 16-bits of the result will be the high-order 16-bits of the first operand. For example if the first operand contains the packed 16-bit quantities {A3,A2,A1,A0} and the second operand contains the packed 16-bit quantities {B3,B2,B1,B0} then the result will be {A3,B3,A2,B2}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    bool asel = ((counter & 1) == 1);
    int in_sel = 2 + counter / 2;
    int16_t srca = get2Byte (rf[SrcA], in_sel);
    int16_t srcb = get2Byte (rf[SrcB], in_sel);
    output = set2Byte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

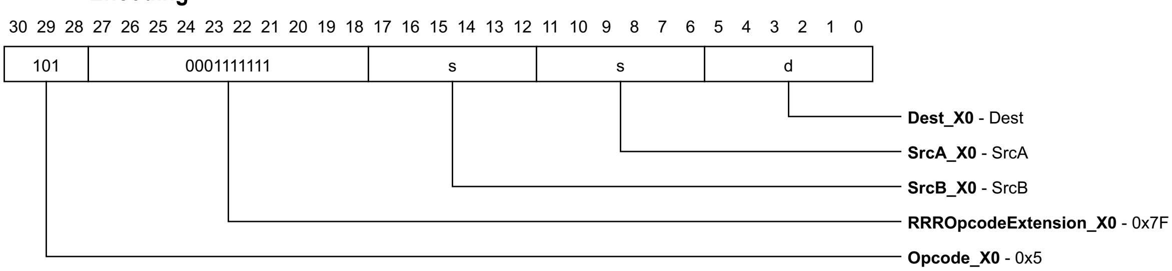

Figure 14-537: v2int_h in X0 Bits Encoding

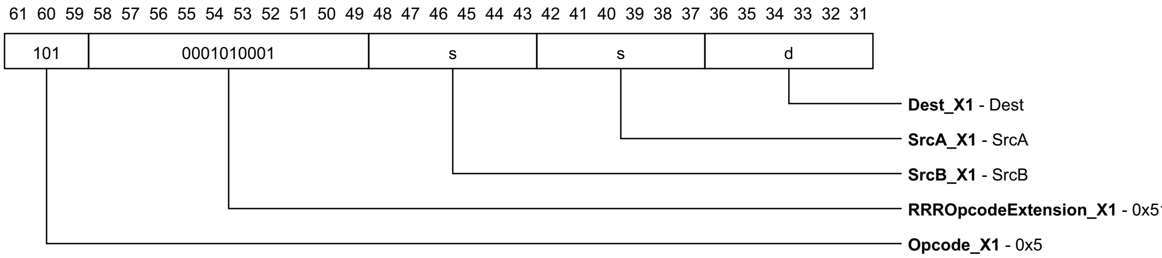

Figure 14-538: v2int_h in X1 Bits Encoding

v2int_l

Vector Two Byte Interleave Low

Syntax

```
v2int_l Dest, SrcA, SrcB
```

Example

```
v2int_l r5, r6, r7
```

Description

Interleave the two low-order 16-bit quantities of the first operand with the two low-order 16-bit quantities of the second operand. The low-order 16-bits of the result will be the low-order 16-bits of the second operand. For example if the first operand contains the packed 16-bit quantities {`A3,A2,A1,A0`} and the second operand contains the packed 16-bit quantities {`B3,B2,B1,B0`} then the result will be {`A1,B1,A0,B0`}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    bool asel = ((counter & 1) == 1);
    int in_sel = 0 + counter / 2;
    int16_t srca = get2Byte (rf[SrcA], in_sel);
    int16_t srcb = get2Byte (rf[SrcB], in_sel);
    output = set2Byte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

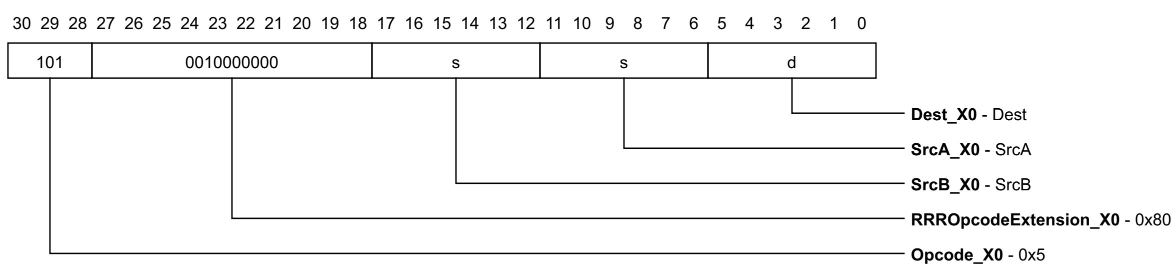

Figure 14-539: v2int_l in X0 Bits Encoding

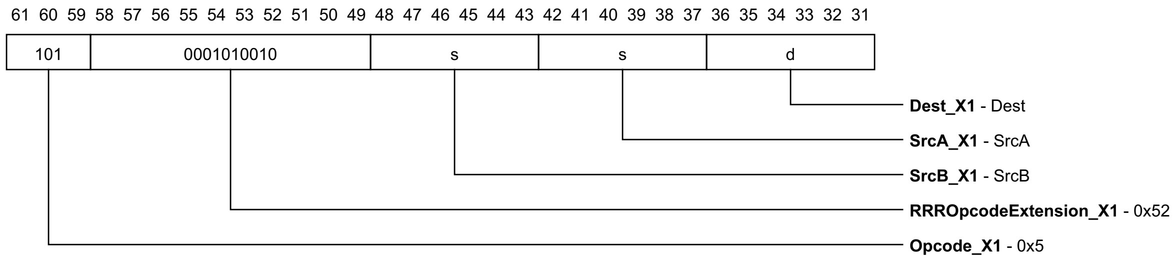

Figure 14-540: v2int_l in X1 Bits Encoding

v2maxs

Vector Two Byte Maximum Signed

Syntax

```
v2maxs Dest, SrcA, SrcB
```

Example

```
v2maxs r5, r6, r7
```

Description

Set each 16-bit quantity in the destination to the maximum of the corresponding 16-bit quantity in the first source operand and the corresponding 16-bit quantity in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca > srcb) ? srca : srcb));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

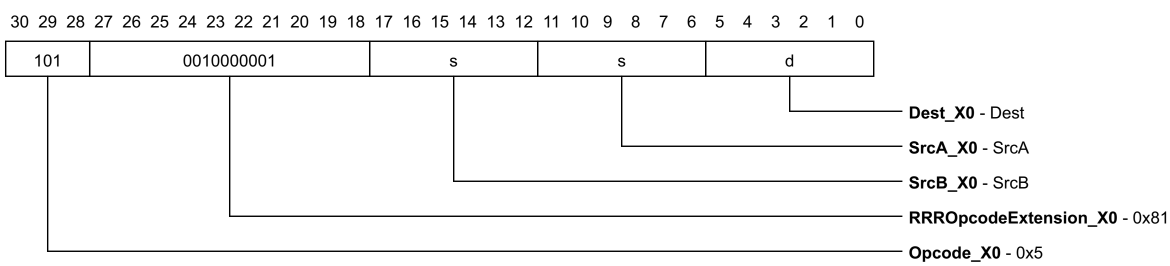

Figure 14-541: v2maxs in X0 Bits Encoding

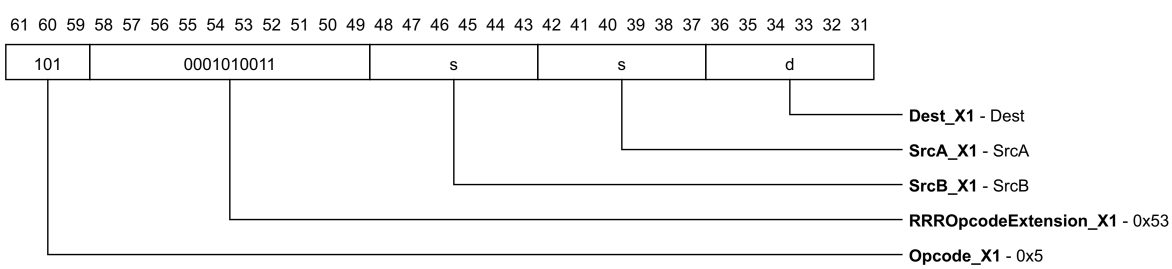

Figure 14-542: v2maxs in X1 Bits Encoding

v2maxsi

Vector Two Byte Maximum Signed Immediate

Syntax

```
v2maxsi Dest, SrcA, Imm8
```

Example

```
v2maxsi r5, r6, 5
```

Description

Set each 16-bit quantity in the destination to the maximum of the corresponding 16-bit quantity in the first source operand and the sign extended immediate.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    output =
      set2Byte (output, counter,
       ((srca > signExtend8 (Imm8)) ? srca : signExtend8 (Imm8)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

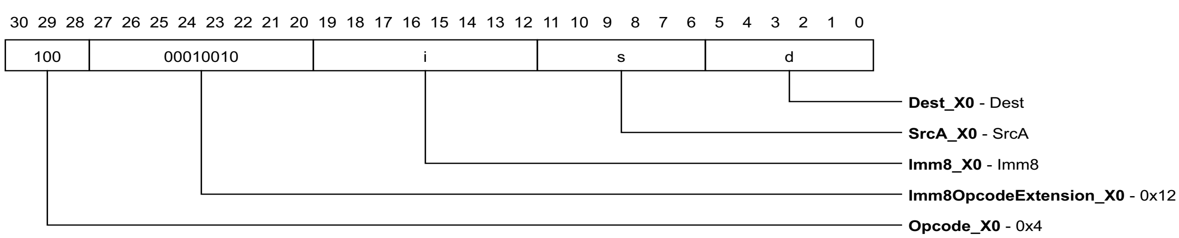

Figure 14-543: v2maxsi in X0 Bits Encoding

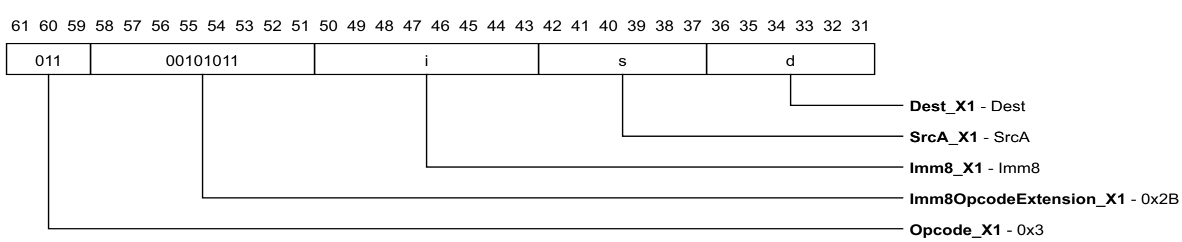

Figure 14-544: v2maxsi in X1 Bits Encoding

v2mins

Vector Two Byte Minimum Signed

Syntax

```
v2mins Dest, SrcA, SrcB
```

Example

```
v2mins r5, r6, r7
```

Description

Set each 16-bit quantity in the destination to the minimum of the corresponding 16-bit quantity in the first source operand and the corresponding 16-bit quantity in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca < srcb) ? srca : srcb));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

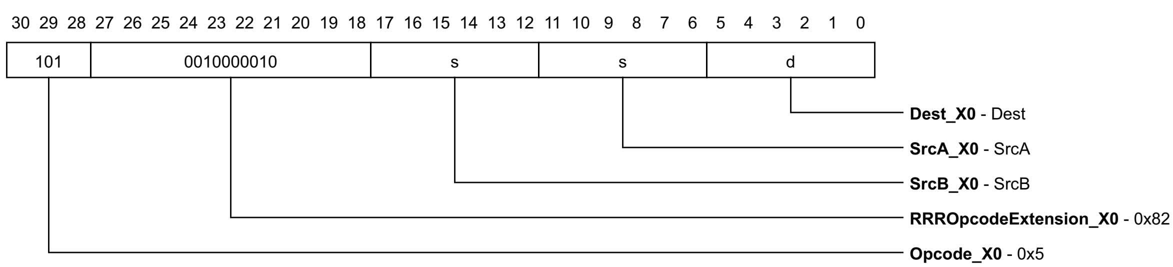

Figure 14-545: v2mins in X0 Bits Encoding

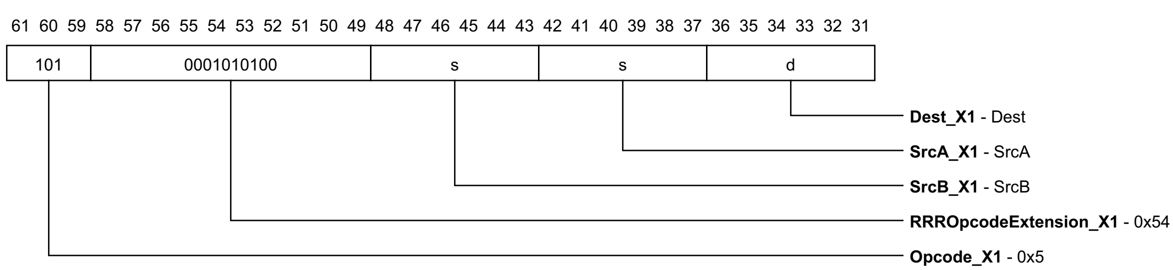

Figure 14-546: v2mins in X1 Bits Encoding

v2minsi

Vector Two Byte Minimum Signed Immediate

Syntax

```
v2minsi Dest, SrcA, Imm8
```

Example

```
v2minsi r5, r6, 5
```

Description

Set each 16-bit quantity in the destination to the minimum of the corresponding 16-bit quantity in the first source operand and the sign extended immediate.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    output =
      set2Byte (output, counter,
       ((srca < signExtend8 (Imm8)) ? srca : signExtend8 (Imm8)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

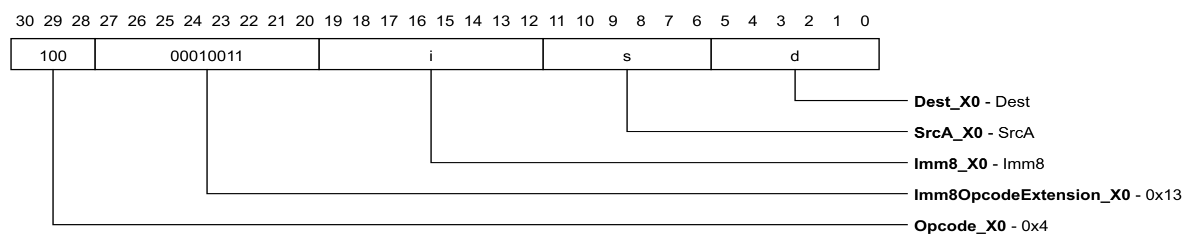

Figure 14-547: v2minsi in X0 Bits Encoding

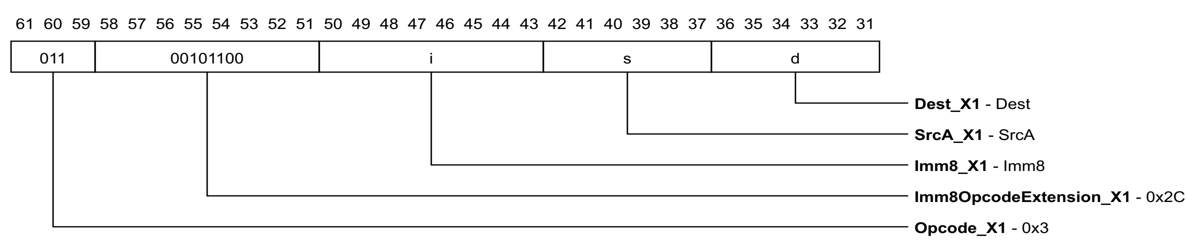

Figure 14-548: v2minsi in X1 Bits Encoding

v2mnz

Vector Two Byte Mask Not Zero

Syntax

```
v2mnz Dest, SrcA, SrcB
```

Example

```
v2mnz r5, r6, r7
```

Description

Set each 16-bit quantity in the destination to the corresponding 16-bit quantity of the second operand if the corresponding 16-bit quantity of the first operand is not zero, otherwise set it to zero (0).

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca != 0) ? srcb : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

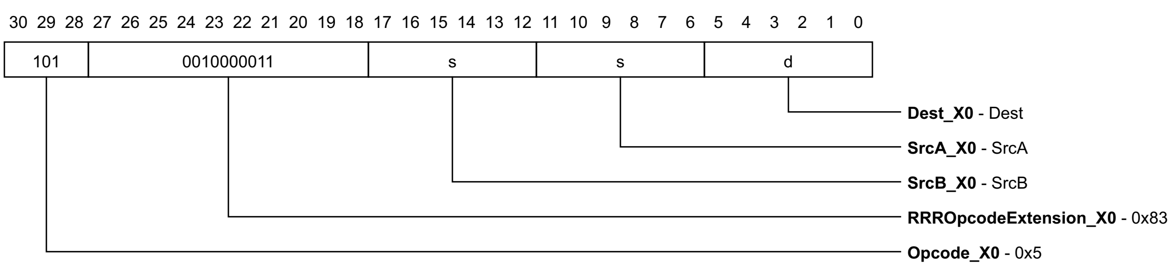

Figure 14-549: v2mnz in X0 Bits Encoding

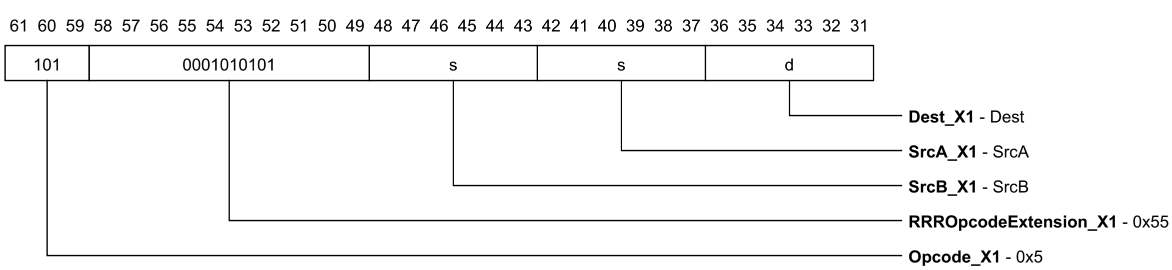

Figure 14-550: v2mnz in X1 Bits Encoding

v2mulfsc

Vector Two Byte Multiply Fixed point Signed Clamped

Syntax

```
v2mulfsc Dest, SrcA, SrcB
```

Example

```
v2mulfsc r5, r6, r7
```

Description

Multiply the two low-order 16-bit quantities in the first source operand by the two low-order 16-bit quantities in the second source operand. The multiplier result is shifted left 1 bit and clamped to the maximum positive 32-bit value. The operands are treated as 16-bit signed fractions with the decimal point below the sign bit.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    int64_t mul_res =
      signExtend16 (get2Byte (rf[SrcA], counter)) *
      signExtend16 (get2Byte (rf[SrcB], counter));
    mul_res = signed_saturate32 (mul_res << 1);
    output = set4Byte (output, counter, mul_res);
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

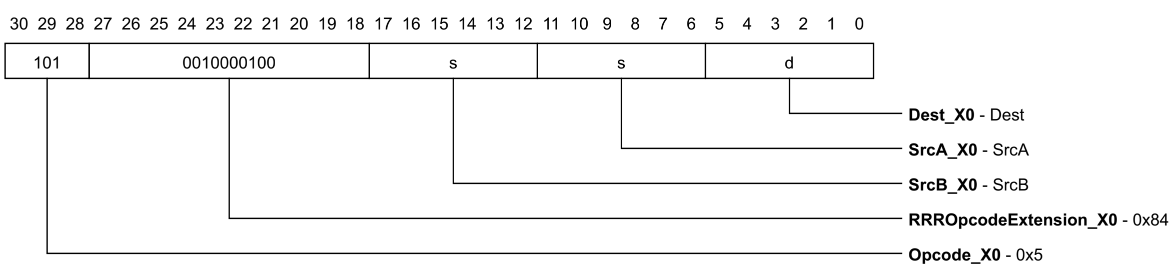

Figure 14-551: v2mulfsc in X0 Bits Encoding

v2muls

Vector Two Byte Multiply Signed

Syntax

```
v2muls Dest, SrcA, SrcB
```

Example

```
v2muls r5, r6, r7
```

Description

Multiply the two low-order 16-bit quantities in the first source operand by the two low-order 16-bit quantities in the second source operand. The two 32-bit multiplication results are packed into 64-bits.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    output =
      set4Byte (output, counter,
       ((int32_t) signExtend16 (get2Byte (rf[SrcA], counter)) *
        (int32_t) signExtend16 (get2Byte (rf[SrcB], counter))));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

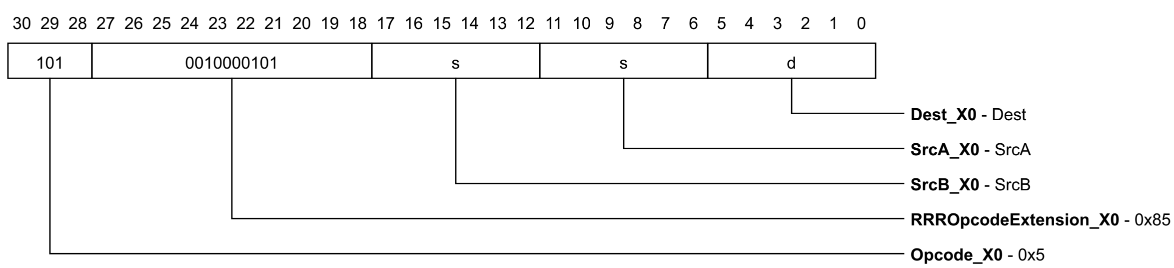

Figure 14-552: v2muls in X0 Bits Encoding

v2mults

Vector Two Byte Multiply and Truncate Signed

Syntax

```
v2mults Dest, SrcA, SrcB
```

Example

```
v2mults r5, r6, r7
```

Description

Multiply the four 16-bit quantities in the first source operand by the four 16-bit quantities in the second source operand. The multiplier result is truncated to the low-order 16-bits of the 32-bit product and packed.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       ((int16_t) get2Byte (rf[SrcA], counter) *
        (int16_t) get2Byte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

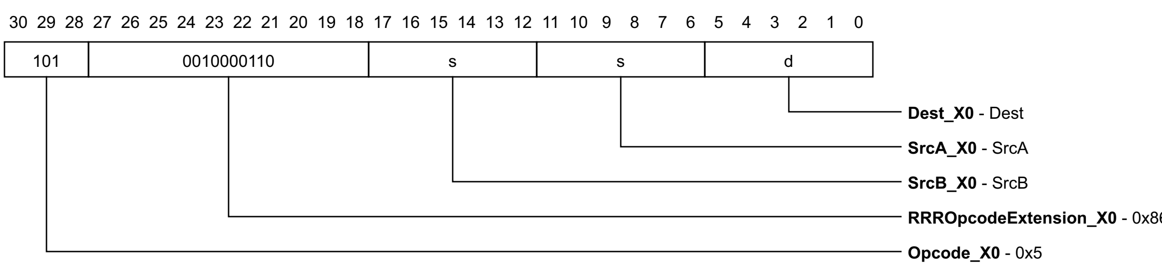

Figure 14-553: v2mults in X0 Bits Encoding

v2mz

Vector Two Byte Mask Zero

Syntax

```
v2mz Dest, SrcA, SrcB
```

Example

```
v2mz r5, r6, r7
```

Description

Set each 16-bit quantity in the destination to the corresponding 16-bit quantity of the second operand if the corresponding 16-bit quantity of the first operand is zero, otherwise set it to zero (0).

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    int16_t srca = get2Byte (rf[SrcA], counter);
    int16_t srcb = get2Byte (rf[SrcB], counter);
    output = set2Byte (output, counter, ((srca == 0) ? srcb : 0));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

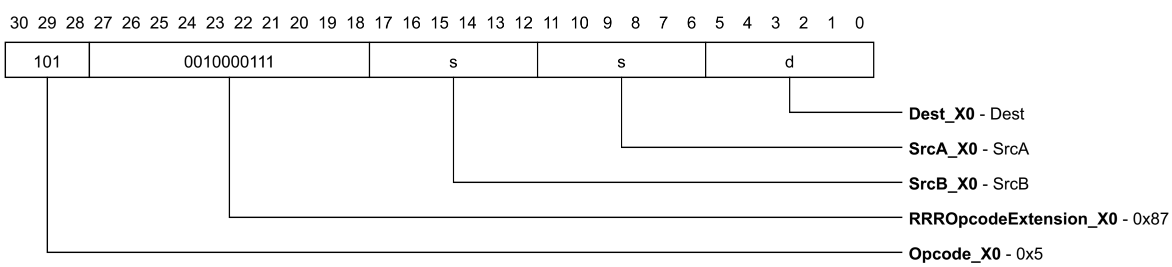

Figure 14-554: v2mz in X0 Bits Encoding

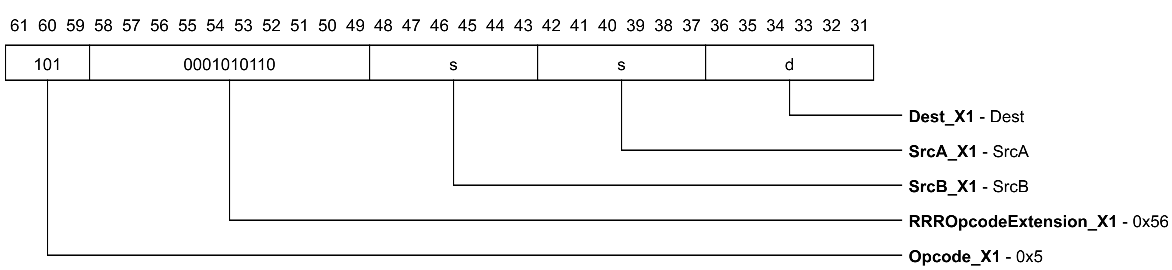

Figure 14-555: v2mz in X1 Bits Encoding

v2packh

Vector Two Bytes Pack High Byte

Syntax

```
v2packh Dest, SrcA, SrcB
```

Example

```
v2packh r5, r6, r7
```

Description

Pack the high-order byte of each of the packed 16-bit quantities of the two source registers into the destination register. The high-order byte of the destination with be the high-order byte of the first operand. For example if the first operand contains the packed bytes {A3_1,A3_0,A2_1,A2_0,A1_1,A1_0,A0_1,A0_0} and the second operand contains the packed bytes {B3_1,B3_0,B2_1,B2_0,B1_1,B1_0,B0_1,B0_0} then the result will be {A3_1,A2_1,A1_1,A0_1,B3_1,B2_1,B1_1,B0_1}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    bool asel = (counter >= 4);
    int in_sel = 1 + (counter & 3) * 2;
    int8_t srca = getByte (rf[SrcA], in_sel);
    int8_t srcb = getByte (rf[SrcB], in_sel);
    output = setByte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

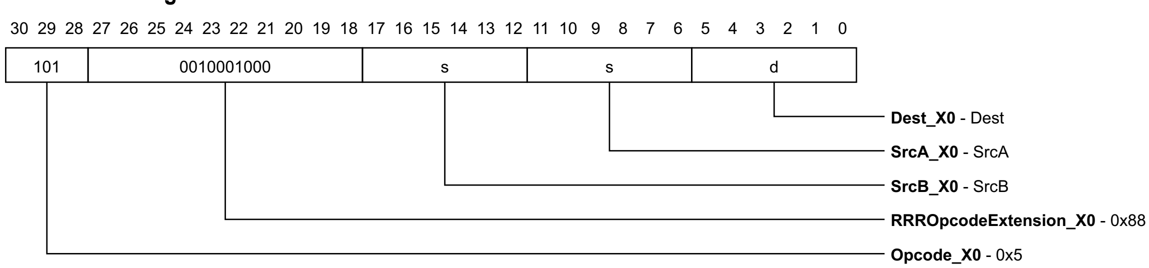

Figure 14-556: v2packh in X0 Bits Encoding

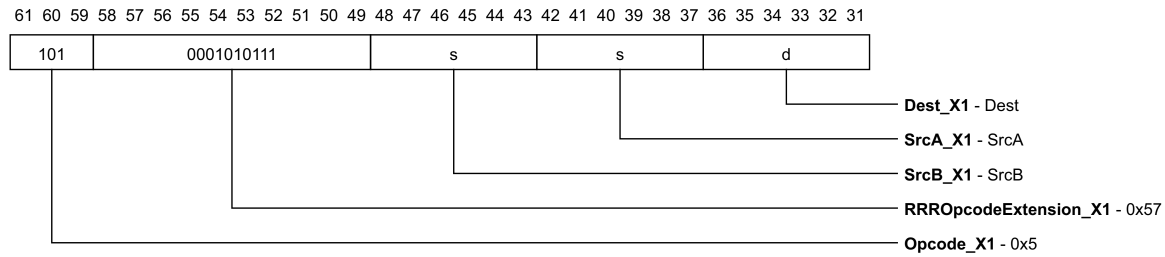

Figure 14-557: v2packh in X1 Bits Encoding

v2packl

Vector Two Byte Pack Low Byte

Syntax

```
v2packl Dest, SrcA, SrcB
```

Example

```
v2packl r5, r6, r7
```

Description

Pack the low-order byte of each of the packed 16-bit quantities of the two source registers into the destination register. The low-order byte of the destination with be the low-order byte of the second operand. For example if the first operand contains the packed bytes {A3_1,A3_0,A2_1,A2_0,A1_1,A1_0,A0_1,A0_0} and the second operand contains the packed bytes {B3_1,B3_0,B2_1,B2_0,B1_1,B1_0,B0_1,B0_0} then the result will be {A3_0,A2_0,A1_0,A0_0,B3_0,B2_0,B1_0,B0_0}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    bool asel = (counter >= 4);
    int in_sel = 0 + (counter & 3) * 2;
    int8_t srca = getByte (rf[SrcA], in_sel);
    int8_t srcb = getByte (rf[SrcB], in_sel);
    output = setByte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

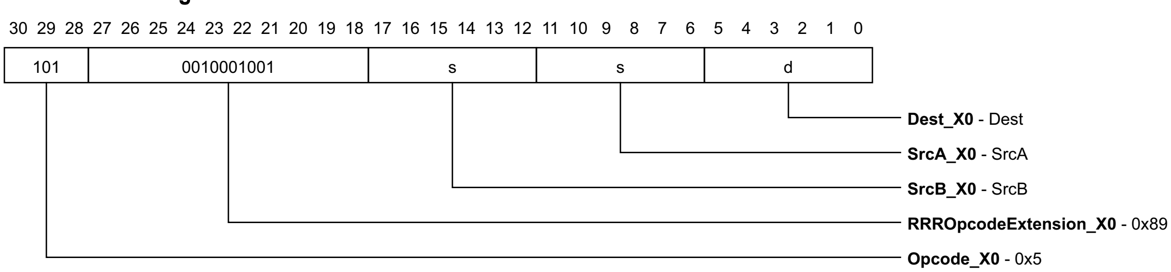

Figure 14-558: v2packl in X0 Bits Encoding

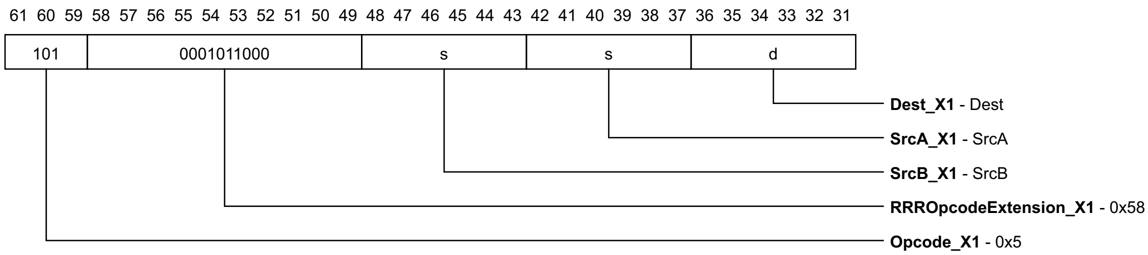

Figure 14-559: v2packl in X1 Bits Encoding

v2packuc

Vector Two Byte Pack Unsigned Clamped

Syntax

```
v2packuc Dest, SrcA, SrcB
```

Example

```
v2packuc r5, r6, r7
```

Description

Clamp each 16-bit quantity of the two source registers to the maximum positive or zero byte value, and then pack the results into the destination register. The high-order byte of the destination will be the clamped high-order 16-bit quantity of the first operand and the low-order byte of the destination will be the clamped low-order 16-bit quantity of the second operand. For example if the first operand contains the packed 16-bit quantities {`A3,A2,A1,A0`} and the second operand contains the packed quantities {`B3,B2,B1,B0`} then the result will be {`sat_A3,sat_A2,sat_A1,sat_A0,sat_B3,sat_B2,sat_B1,sat_B0`}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++)
  {
    bool asel = (counter >= 4);
    int in_sel = counter & 3;
    int16_t srca = signExtend16 (get2Byte (rf[SrcA], in_sel));
    int16_t srcb = signExtend16 (get2Byte (rf[SrcB], in_sel));
    output =
      setByte (output, counter, unsigned_saturate8 (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

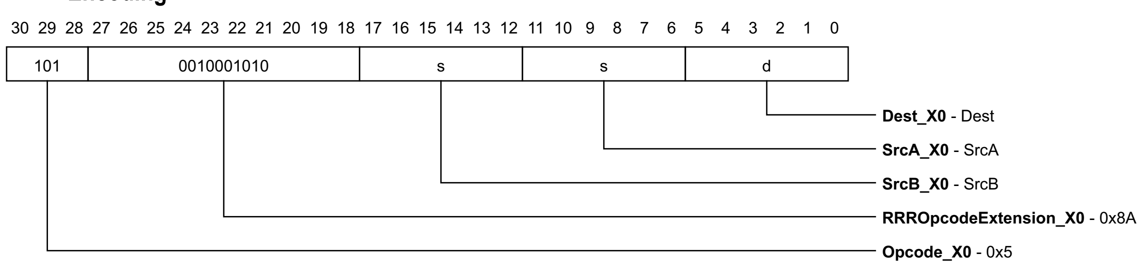

Figure 14-560: v2packuc in X0 Bits Encoding

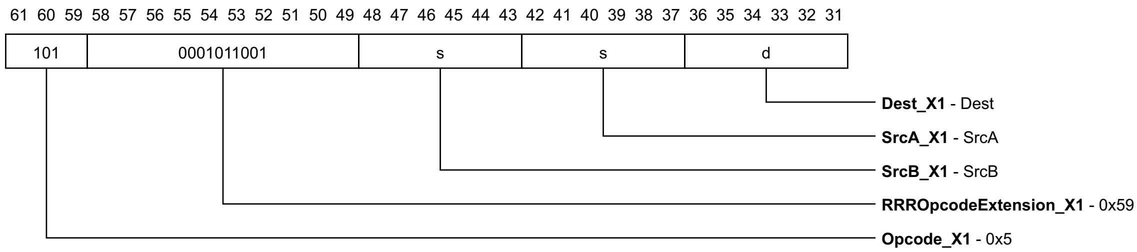

Figure 14-561: v2packuc in X1 Bits Encoding

v2sadas

Vector Two Byte Sum of Absolute Difference Accumulate Signed

Syntax

```
v2sadas Dest, SrcA, SrcB
```

Example

```
v2sadas r5, r6, r7
```

Description

Sum the absolute differences between the four 16-bit quantities in the first source operand and the four 16-bit quantities in the second source operand and accumulate the sum into the destination register.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output +=
      abs (signExtend16 (get2Byte (rf[SrcA], counter)) -
   signExtend16 (get2Byte (rf[SrcB], counter)));
  }

rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

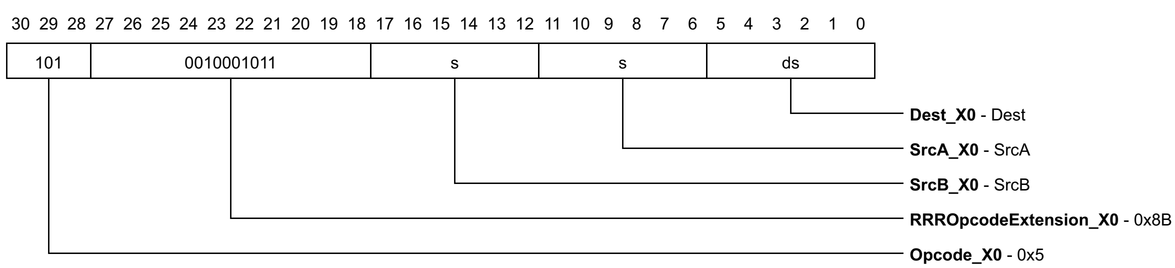

Figure 14-562: v2sadas in X0 Bits Encoding

v2sadau

Vector Two Byte Sum of Absolute Difference Accumulate Unsigned

Syntax

```
v2sadau Dest, SrcA, SrcB
```

Example

```
v2sadau r5, r6, r7
```

Description

Sum the absolute differences between the four 16-bit quantities in the first source operand and the four 16-bit quantities in the second source operand and accumulate the sum into the destination register.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output +=
      abs (get2Byte (rf[SrcA], counter) - get2Byte (rf[SrcB], counter));
  }

rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

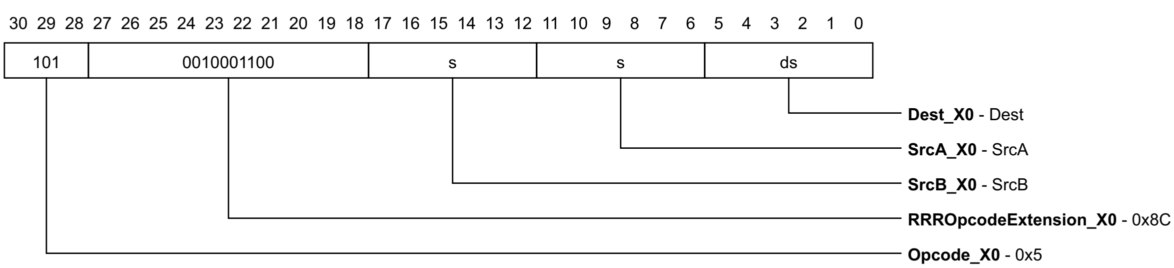

Figure 14-563: v2sadau in X0 Bits Encoding

v2sads

Vector Two Byte Sum of Absolute Difference Signed

Syntax

```
v2sads Dest, SrcA, SrcB
```

Example

```
v2sads r5, r6, r7
```

Description

Sum the absolute differences between the four 16-bit quantities in the first source operand and the four 16-bit quantities in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output +=
      abs (signExtend16 (get2Byte (rf[SrcA], counter)) -
   signExtend16 (get2Byte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

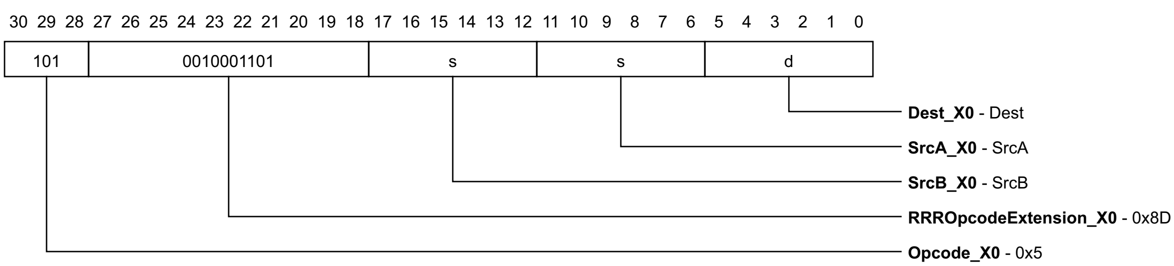

Figure 14-564: v2sads in X0 Bits Encoding

v2sadu

Vector Two Byte Sum of Absolute Difference Unsigned

Syntax

```
v2sadu Dest, SrcA, SrcB
```

Example

```
v2sadu r5, r6, r7
```

Description

Sum the absolute differences between the four 16-bit quantities in the first source operand and the four 16-bit quantities in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output +=
      abs (get2Byte (rf[SrcA], counter) - get2Byte (rf[SrcB], counter));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

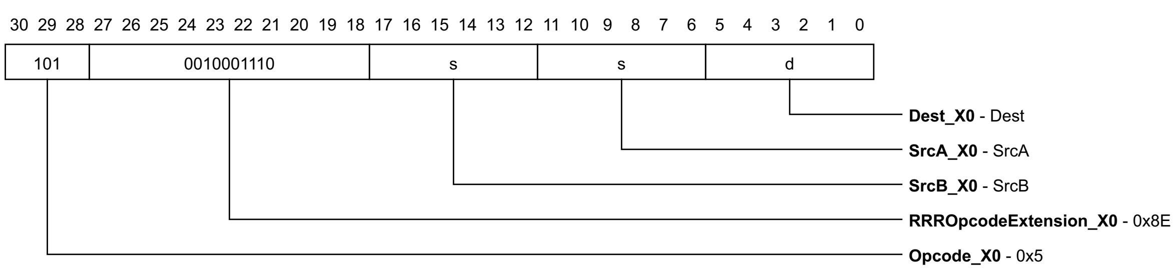

Figure 14-565: v2sadu in X0 Bits Encoding

v2shl

Vector Two Byte Shift Left

Syntax

```
v2shl Dest, SrcA, SrcB
```

Example

```
v2shl r5, r6, r7
```

Description

Logically shift each of the four 16-bit quantities in the first source operand to the left by the second source operand. The effective shift amount is the specified operand modulo 16. Logical left shift shifts zeros into the low ordered bits.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (get2Byte (rf[SrcA], counter) <<
        (((UnsignedMachineWord) rf[SrcB]) % 16)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

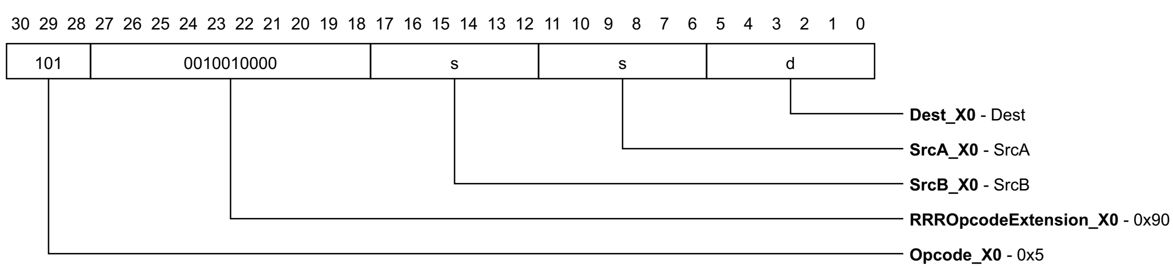

Figure 14-566: v2shl in X0 Bits Encoding

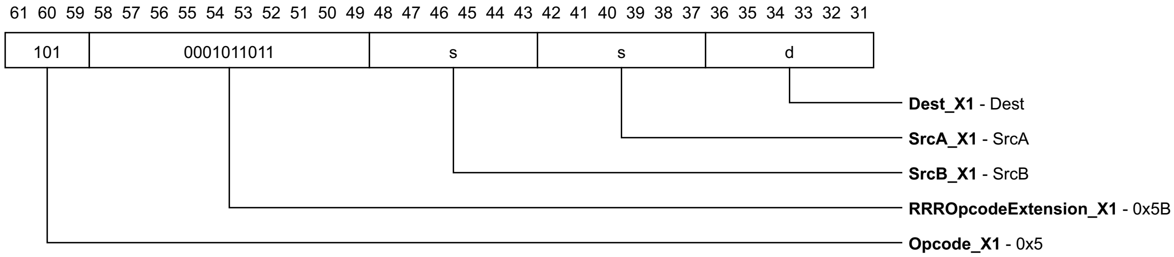

Figure 14-567: v2shl in X1 Bits Encoding

v2shli

Vector Two Byte Shift Left Immediate

Syntax

```
v2shli Dest, SrcA, ShAmt
```

Example

```
v2shli r5, r6, 5
```

Description

Logically shift each of the four 16-bit quantities in the first source operand to the left by an immediate. The effective shift amount is the specified immediate modulo 16. Left shifts shift zeros into the low ordered bits and is suitable to be used as unsigned multiplication by powers of two.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (get2Byte (rf[SrcA], counter) <<
        (((UnsignedMachineWord) ShAmt) % 16)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

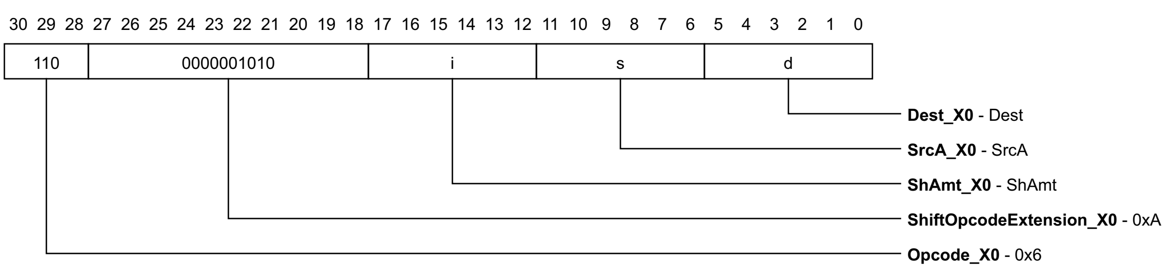

Figure 14-568: v2shli in X0 Bits Encoding

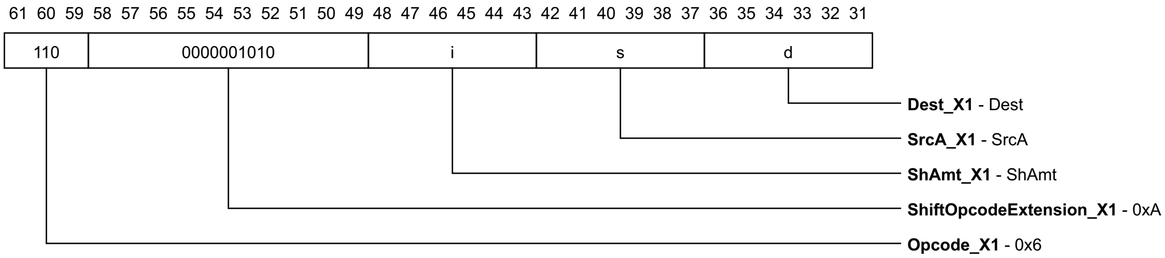

Figure 14-569: v2shli in X1 Bits Encoding

v2shlsc

Vector Two Byte Shift Left Signed Clamped

Syntax

```
v2shlsc Dest, SrcA, SrcB
```

Example

```
v2shlsc r5, r6, r7
```

Description

Logically shift each of the four 16-bit quantities in the first source operand to the left by the second source operand. The effective shift amount is the specified operand modulo 16. Logical left shift shifts zeros into the low ordered bits. If the left shift would arithmetically overflow a 16-bit quantity, the result is clamped to the minimum negative or maximum positive 16-bit value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       signed_saturate16 (signExtend16 (get2Byte (rf[SrcA], counter))
                        << (((UnsignedMachineWord) rf[SrcB]) %
                            16)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

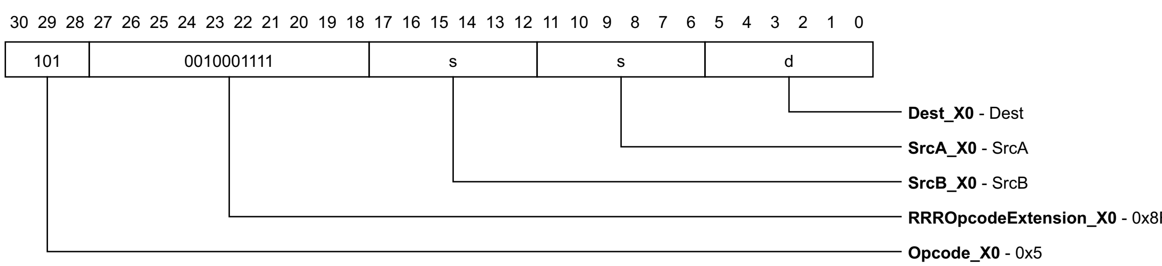

Figure 14-570: v2shlsc in X0 Bits Encoding

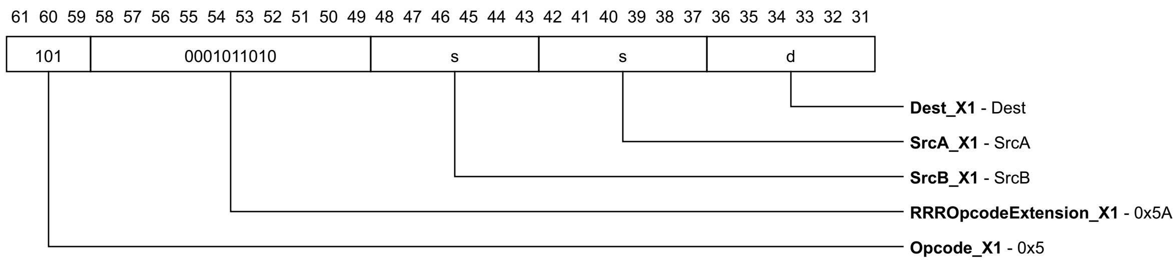

Figure 14-571: v2shlsc in X1 Bits Encoding

v2shrs

Vector Two Byte Shift Right Signed

Syntax

```
v2shrs Dest, SrcA, SrcB
```

Example

```
v2shrs r5, r6, r7
```

Description

Arithmetically shift each of the four 16-bit quantities in the first source operand to the right by the second source operand. The effective shift amount is the specified operand modulo 16. Arithmetic right shift shifts the high ordered bit into the high ordered bits.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (signExtend16 (get2Byte (rf[SrcA], counter)) >>
        (((UnsignedMachineWord) rf[SrcB]) % 16)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

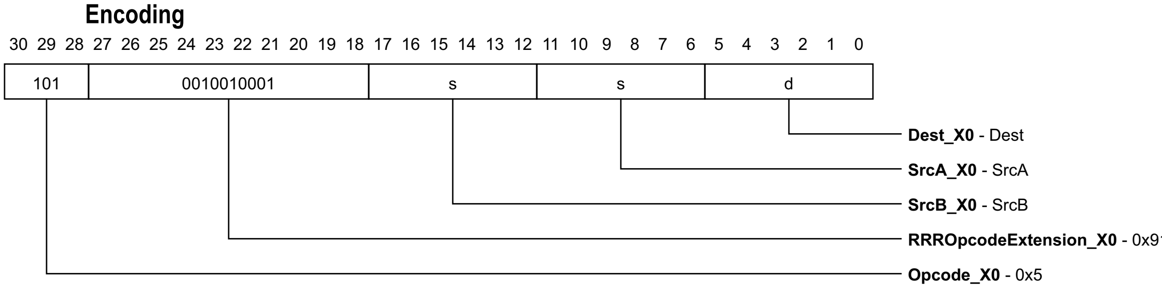

Figure 14-572: v2shrs in X0 Bits Encoding

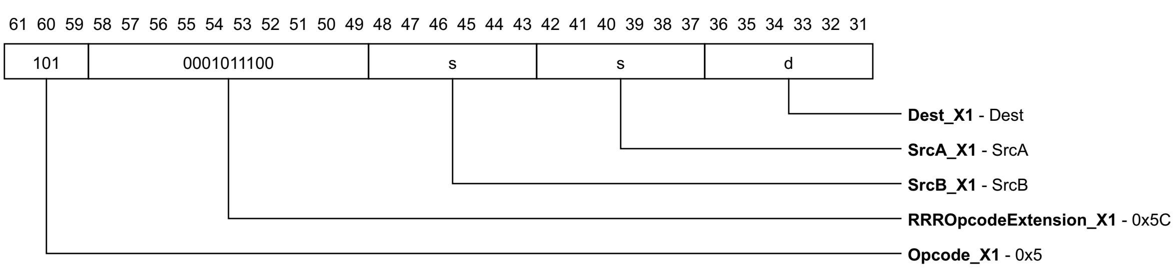

Figure 14-573: v2shrs in X1 Bits Encoding

v2shrsi

Vector Two Byte Shift Right Signed Immediate

Syntax

```
v2shrsi Dest, SrcA, ShAmt
```

Example

```
v2shrsi r5, r6, 5
```

Description

Arithmetically shift each of four 16-bit quantities in the first source operand to the right by an immediate. The effective shift amount is the specified immediate modulo 16. Arithmetic right shifts shift the source high ordered bit into the high ordered bits of the result.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (signExtend16 (get2Byte (rf[SrcA], counter)) >>
        (((UnsignedMachineWord) ShAmt) % 16)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

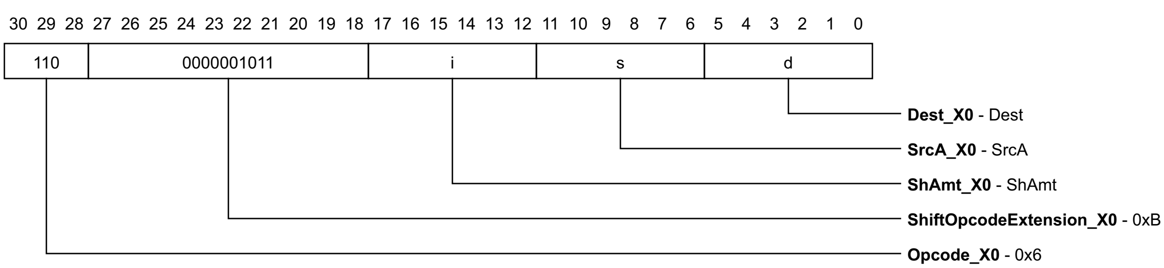

Figure 14-574: v2shrsi in X0 Bits Encoding

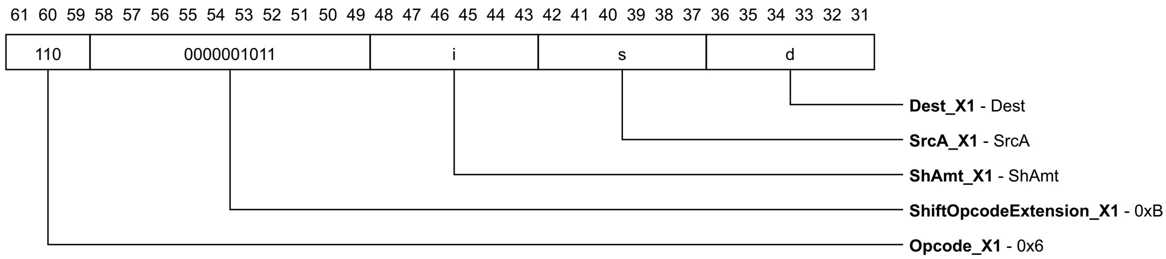

Figure 14-575: v2shrsi in X1 Bits Encoding

v2shru

Vector Two Byte Shift Right Unsigned

Syntax

```
v2shru Dest, SrcA, SrcB
```

Example

```
v2shru r5, r6, r7
```

Description

Logically shift each of the four 16-bit quantities in the first source operand to the right by the second source operand. The effective shift amount is the specified operand modulo 16. Logical right shift shifts zeros into the high ordered bits.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (get2Byte (rf[SrcA], counter) >>
        (((UnsignedMachineWord) rf[SrcB]) % 16)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

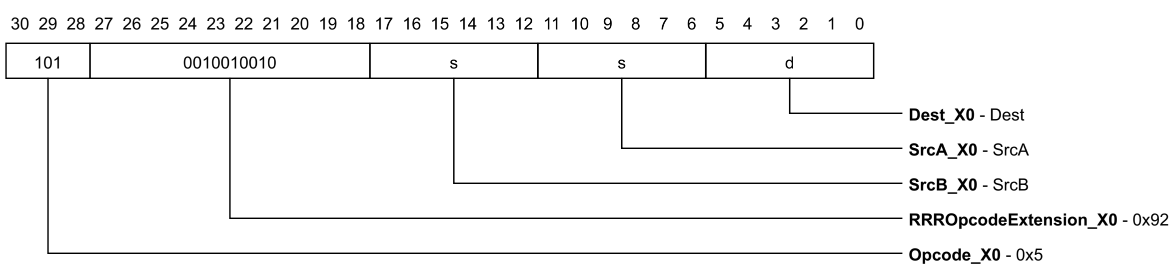

Figure 14-576: v2shru in X0 Bits Encoding

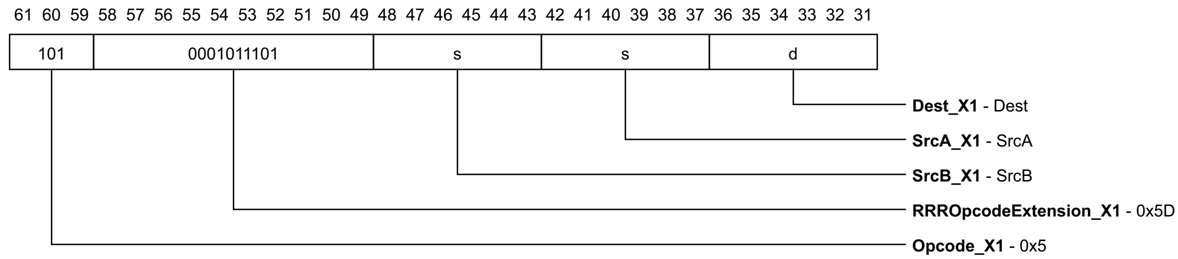

Figure 14-577: v2shru in X1 Bits Encoding

v2shrui

Vector Two Byte Shift Right Unsigned Immediate

Syntax

```
v2shrui Dest, SrcA, ShAmt
```

Example

```
v2shrui r5, r6, 5
```

Description

Logically shift each of the four 16-bit quantities in the first source operand to the right by an immediate. The effective shift amount is the specified immediate modulo 16. Logical right shifts shift zeros into the high ordered bits is suitable to be used as unsigned integer division by powers of two.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (get2Byte (rf[SrcA], counter) >>
        (((UnsignedMachineWord) ShAmt) % 16)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

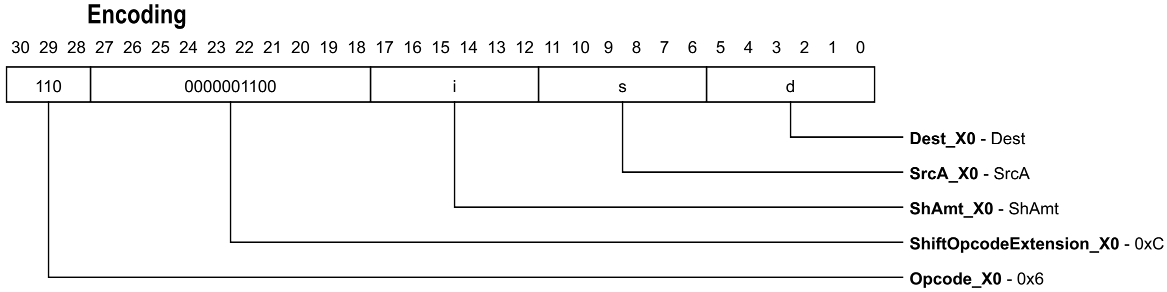

Figure 14-578: v2shrui in X0 Bits Encoding

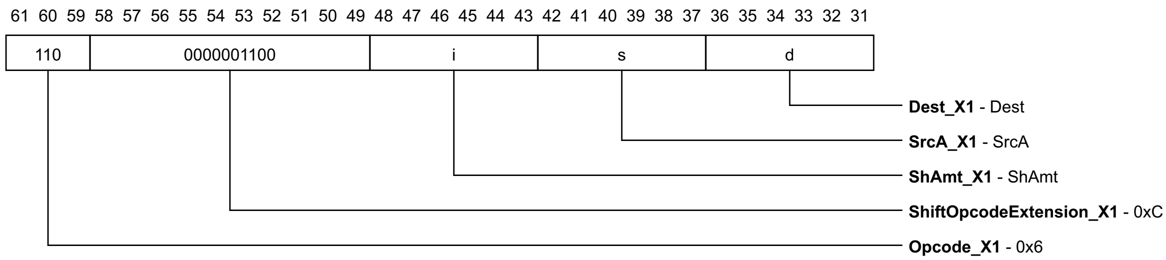

Figure 14-579: v2shrui in X1 Bits Encoding

v2sub

Vector Two Byte Subtract

Syntax

```
v2sub Dest, SrcA, SrcB
```

Example

```
v2sub r5, r6, r7
```

Description

Subtract the four 16-bit quantities in the second source operand from the four 16-bit quantities in the first source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       (get2Byte (rf[SrcA], counter) -
        get2Byte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

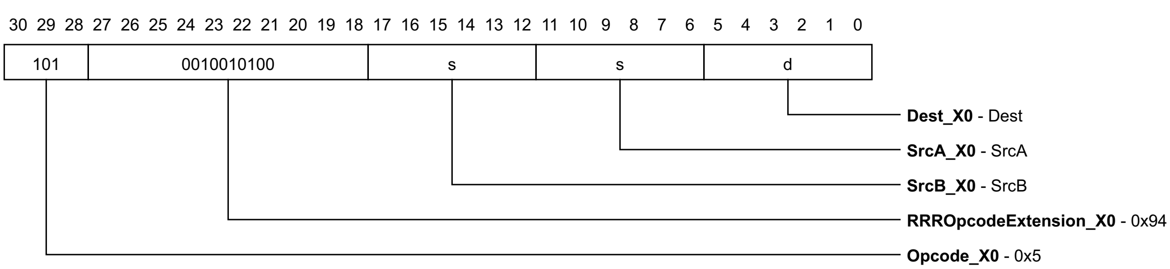

Figure 14-580: v2sub in X0 Bits Encoding

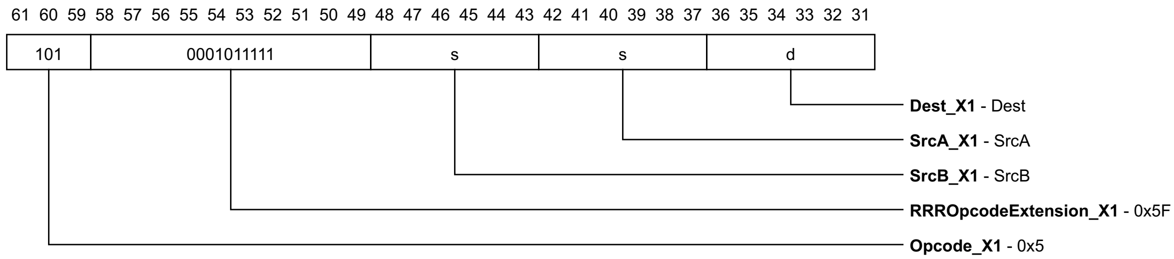

Figure 14-581: v2sub in X1 Bits Encoding

v2subsc

Vector Two Byte Subtract Signed Clamped

Syntax

```
v2subsc Dest, SrcA, SrcB
```

Example

```
v2subsc r5, r6, r7
```

Description

Subtract the four 16-bit quantities in the second source operand from the four 16-bit quantities in the first source operand and clamp each result to the minimum negative value or maximum positive value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    output =
      set2Byte (output, counter,
       signed_saturate16 (signExtend16 (get2Byte (rf[SrcA], counter))
                        -
                        signExtend16 (get2Byte
                                   (rf[SrcB], counter))));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

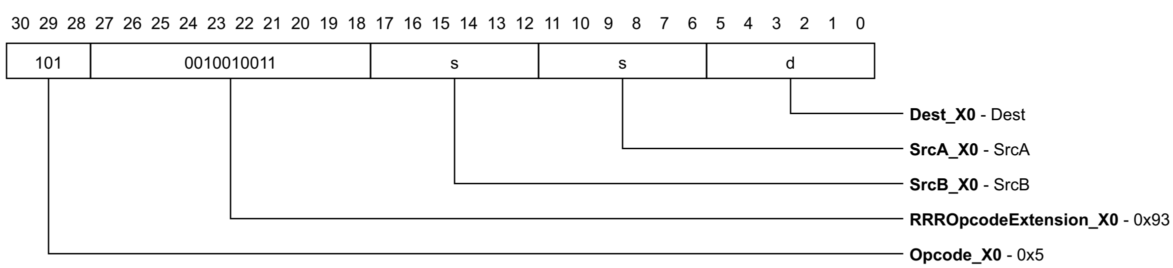

Figure 14-582: v2subsc in X0 Bits Encoding

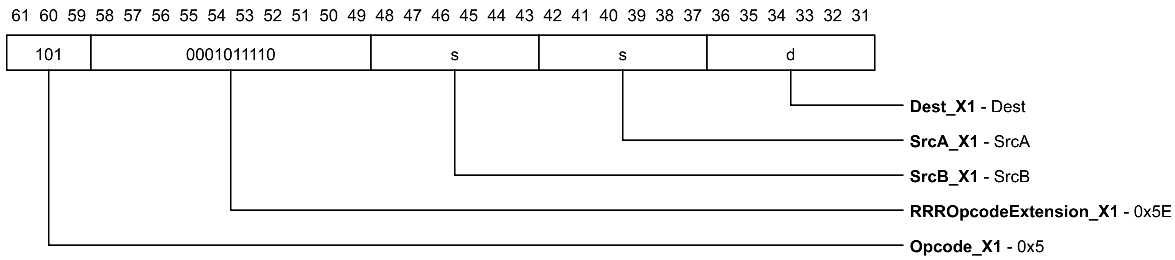

Figure 14-583: v2subsc in X1 Bits Encoding

v4add

Vector Four Byte Add

Syntax

```
v4add Dest, SrcA, SrcB
```

Example

```
v4add r5, r6, r7
```

Description

Add the two 32-bit quantities in the first source operand to the two 32-bit quantities in the second source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    output =
      set4Byte (output, counter,
       (get4Byte (rf[SrcA], counter) +
        get4Byte (rf[SrcB], counter)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

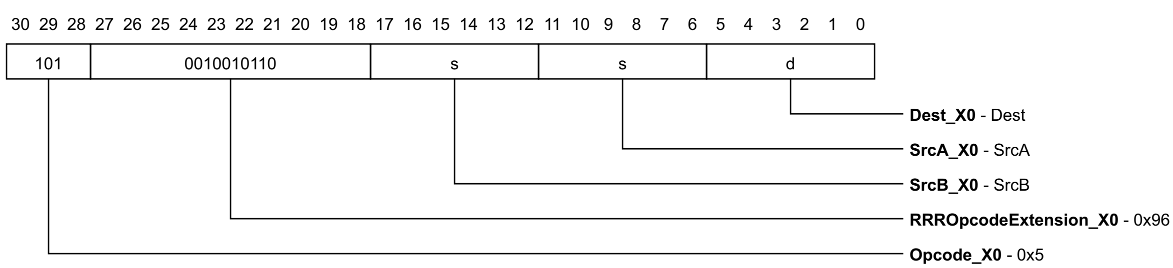

Figure 14-584: v4add in X0 Bits Encoding

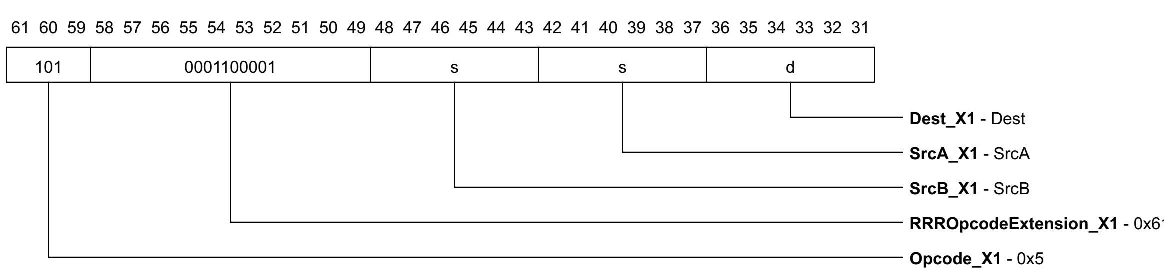

Figure 14-585: v4add in X1 Bits Encoding

v4addsc

Vector Four Byte Add Signed Clamped

Syntax

```
v4addsc Dest, SrcA, SrcB
```

Example

```
v4addsc r5, r6, r7
```

Description

Add the two 32-bit quantities in the first source operand to the two 32-bit quantities in the second source operand and clamp each result to the minimum negative or maximum positive 32-bit value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    output =
      set4Byte (output, counter,
       signed_saturate32 (signExtend32 (get4Byte (rf[SrcA], counter))
                        +
                        signExtend32 (get4Byte
                                   (rf[SrcB], counter))));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

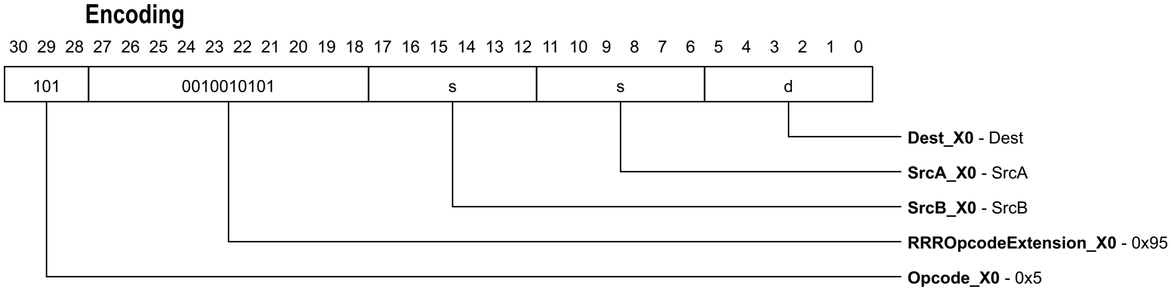

Figure 14-586: v4addsc in X0 Bits Encoding

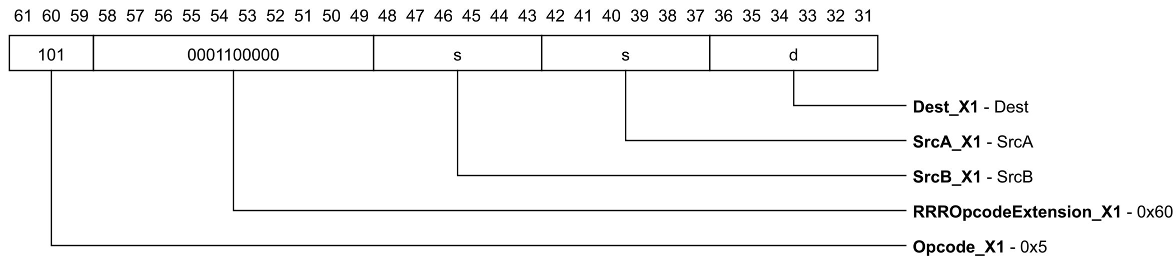

Figure 14-587: v4addsc in X1 Bits Encoding

v4int_h

Vector Four Byte Interleave High

Syntax

```
v4int_h Dest, SrcA, SrcB
```

Example

```
v4int_h r5, r6, r7
```

Description

Interleave the high-order 32-bit quantity of the first operand with the high-order 32-bit quantity of the second operand. The high-order 32-bits of the result will be the high-order 32-bits of the first operand. For example if the first operand contains the packed 32-bit quantities {`A1,A0`} and the second operand contains the packed 32-bit quantities {`B1,B0`} then the result will be {`A1,B1`}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    bool asel = ((counter & 1) == 1);
    int in_sel = 1 + counter / 2;
    int32_t srca = get4Byte (rf[SrcA], in_sel);
    int32_t srcb = get4Byte (rf[SrcB], in_sel);
    output = set4Byte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

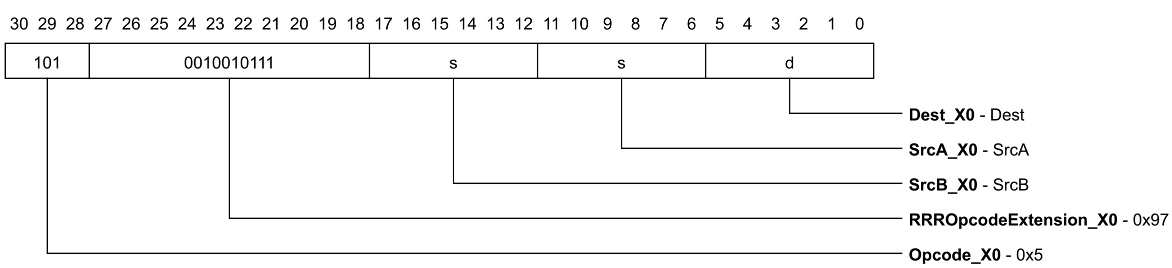

Figure 14-588: v4int_h in X0 Bits Encoding

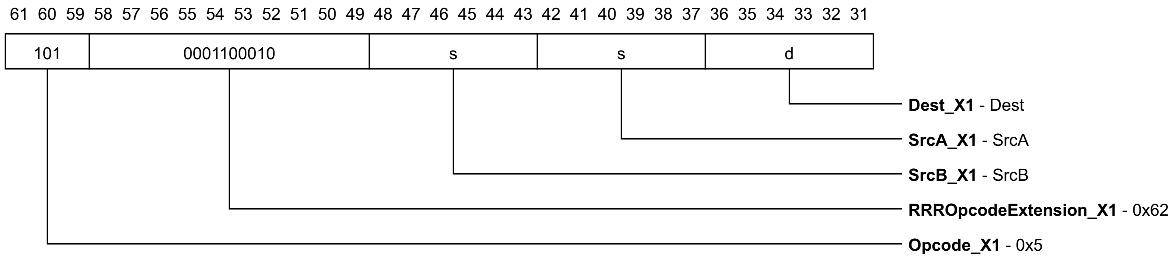

Figure 14-589: v4int_h in X1 Bits Encoding

v4int_l

Vector Four Byte Interleave Low

Syntax

```
v4int_l Dest, SrcA, SrcB
```

Example

```
v4int_l r5, r6, r7
```

Description

Interleave the low-order 32-bit quantity of the first operand with the low-order 32-bit quantity of the second operand. The low-order 32-bits of the result will be the low-order 32-bits of the second operand. For example if the first operand contains the packed 32-bit quantities {`A1,A0`} and the second operand contains the packed 32-bit quantities {`B1,B0`} then the result will be {`A0,B0`}.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    bool asel = ((counter & 1) == 1);
    int in_sel = 0 + counter / 2;
    int32_t srca = get4Byte (rf[SrcA], in_sel);
    int32_t srcb = get4Byte (rf[SrcB], in_sel);
    output = set4Byte (output, counter, (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

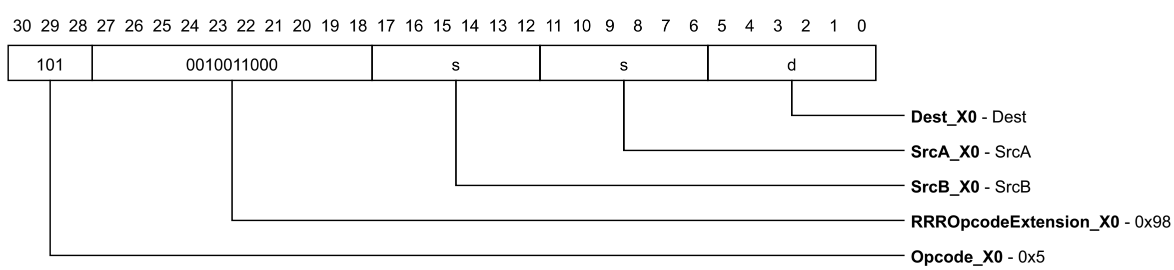

Figure 14-590: v4int_l in X0 Bits Encoding

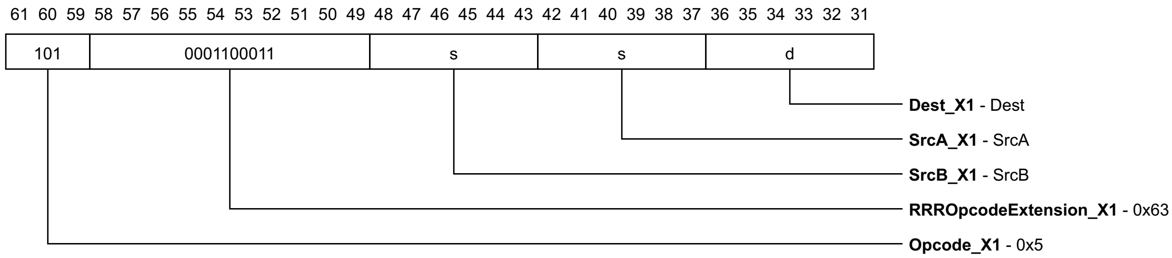

Figure 14-591: v4int_l in X1 Bits Encoding

v4packsc

Vector Four Byte Pack Signed Clamped

Syntax

```
v4packsc Dest, SrcA, SrcB
```

Example

```
v4packsc r5, r6, r7
```

Description

Clamp each of the packed 32-bit quantities in each of the two source registers to the maximum positive or minimum negative 16-bit value, and then pack the results into the destination register. The low-order 16-bit quantity of the destination will be the clamped low-order 32-bit quantity from the second operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 16); counter++)
  {
    bool asel = (counter >= 2);
    int in_sel = counter & 1;
    int64_t srca = signExtend32 (rf[SrcA] >> (in_sel * 32));
    int64_t srcb = signExtend32 (rf[SrcB] >> (in_sel * 32));
    output =
      set2Byte (output, counter, signed_saturate16 (asel ? srca : srcb));
  } rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

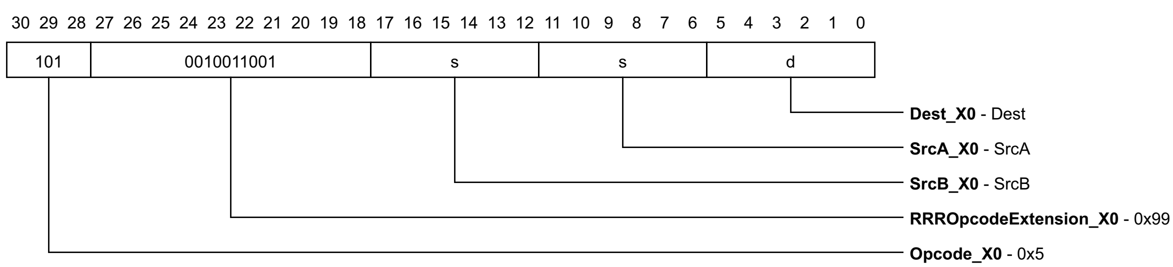

Figure 14-592: v4packsc in X0 Bits Encoding

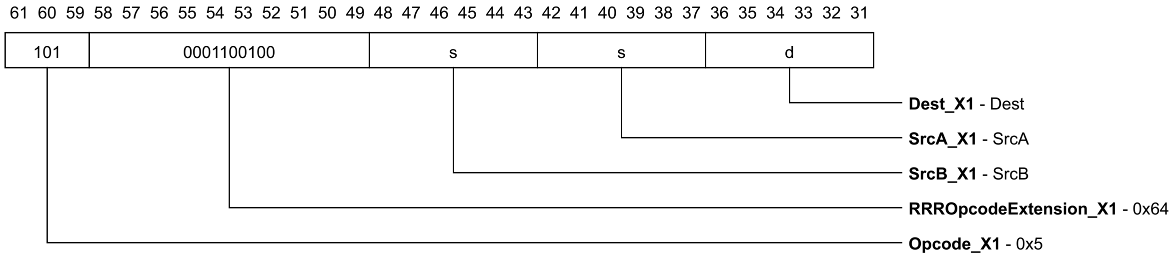

Figure 14-593: v4packsc in X1 Bits Encoding

v4shl

Vector Four Byte Shift Left

Syntax

```
v4shl Dest, SrcA, SrcB
```

Example

```
v4shl r5, r6, r7
```

Description

Logically shift each of the two 32-bit quantities in the first source operand to the left by the second source operand. The effective shift amount is the specified operand modulo 32. Logical left shift shifts zeros into the low ordered bits.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    output =
      set4Byte (output, counter,
       (get4Byte (rf[SrcA], counter) <<
        (((UnsignedMachineWord) rf[SrcB]) % 32)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

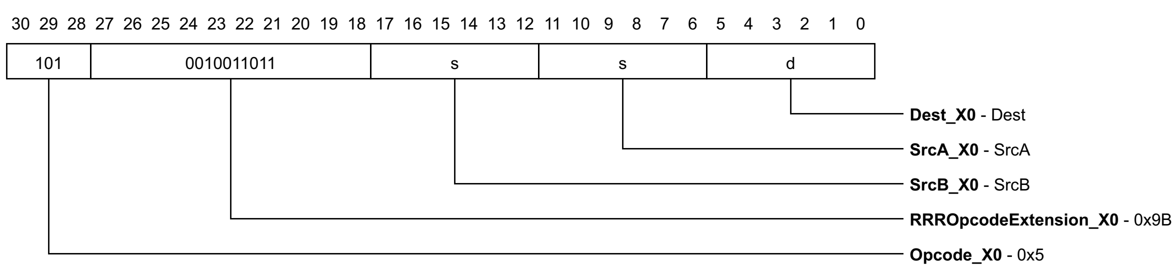

Figure 14-594: v4shl in X0 Bits Encoding

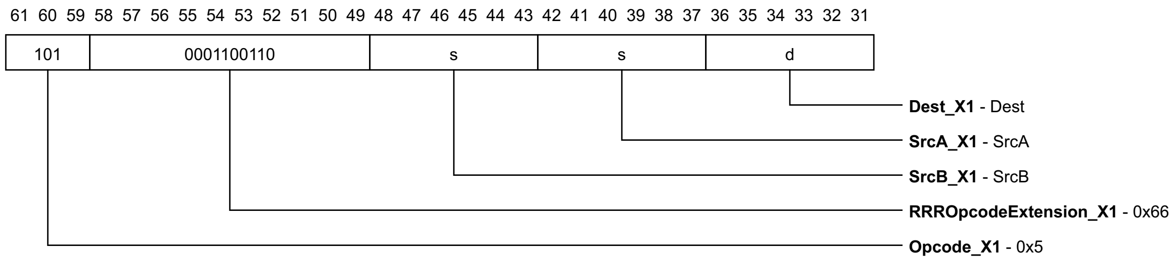

Figure 14-595: v4shl in X1 Bits Encoding

v4shlsc

Vector Four Byte Shift Left Signed Clamped

Syntax

```
v4shlsc Dest, SrcA, SrcB
```

Example

```
v4shlsc r5, r6, r7
```

Description

Logically shift each of the two 32-bit quantities in the first source operand to the left by the second source operand. The effective shift amount is the specified operand modulo 32. Logical left shift shifts zeros into the low ordered bits. If the left shift would arithmetically overflow a 32-bit quantity, the result is clamped to the minimum negative or maximum positive 32-bit value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
  {
    output =
      set4Byte (output, counter,
       signed_saturate32 (signExtend32 (get4Byte (rf[SrcA], counter))
                        << (((UnsignedMachineWord) rf[SrcB]) %
                            32)));
  }

rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

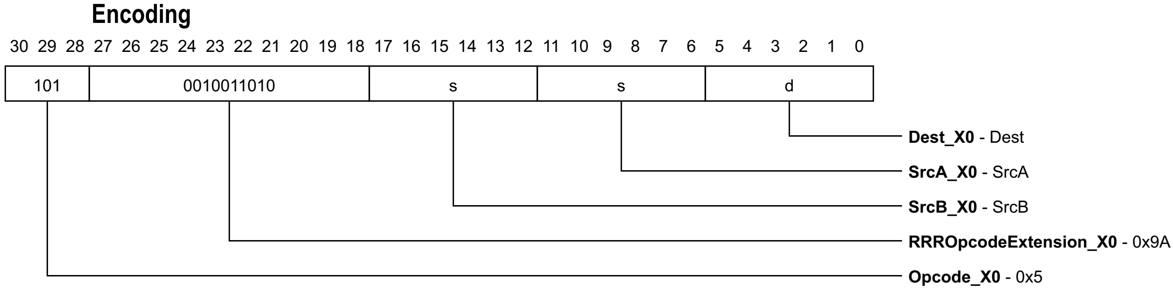

Figure 14-596: v4shlsc in X0 Bits Encoding

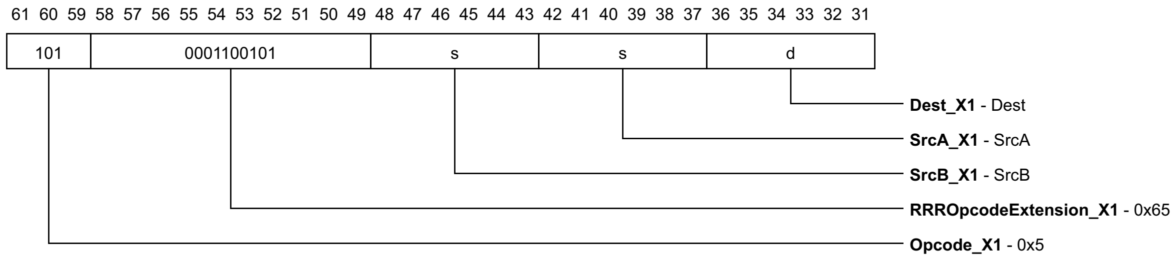

Figure 14-597: v4shlsc in X1 Bits Encoding

v4shrs

Vector Four Byte Shift Right Signed

Syntax

```
v4shrs Dest, SrcA, SrcB
```

Example

```
v4shrs r5, r6, r7
```

Description

Arithmetically shift each of the two 32-bit quantities in the first source operand to the right by the second source operand. If the shift amount is larger than 32, the effective shift amount is computed to be the specified shift amount modulo 32. Arithmetic right shift shifts the high ordered bit into the high ordered bits.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
   {
      output =
         set4Byte (output, counter,
            (signExtend32 (get4Byte (rf[SrcA], counter)) >>
               (((UnsignedMachineWord) rf[SrcB]) % 32)));
   }
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

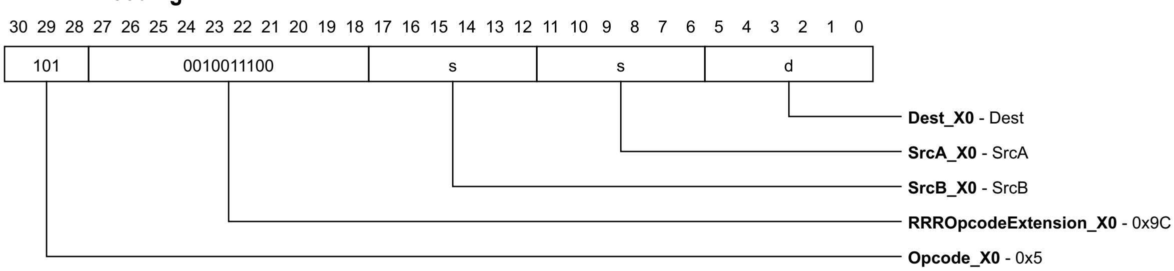

Figure 14-598: v4shrs in X0 Bits Encoding

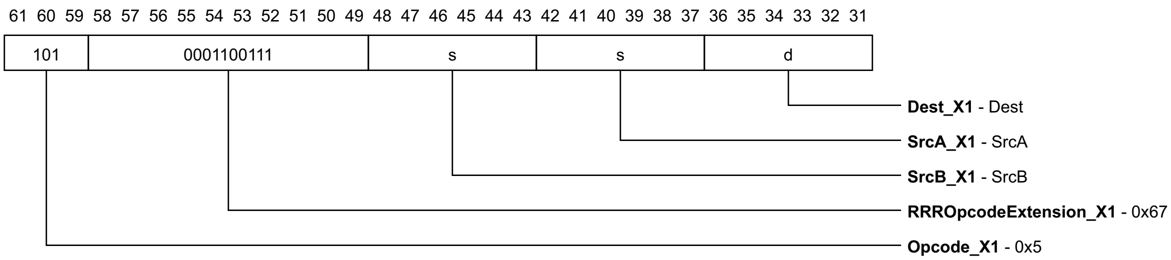

Figure 14-599: v4shrs in X1 Bits Encoding

v4shru

Vector Four Byte Shift Right Unsigned

Syntax

```
v4shru Dest, SrcA, SrcB
```

Example

```
v4shru r5, r6, r7
```

Description

Logically shift each of the two 32-bit quantities in the first source operand to the right by the second source operand. If the shift amount is larger than 32, the effective shift amount is computed to be the specified shift amount modulo 32. Logical right shift shifts zeros into the high ordered bits.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
   {
      output =
         set4Byte (output, counter,
            (get4Byte (rf[SrcA], counter) >>
               (((UnsignedMachineWord) rf[SrcB]) % 32)));
   }
   rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

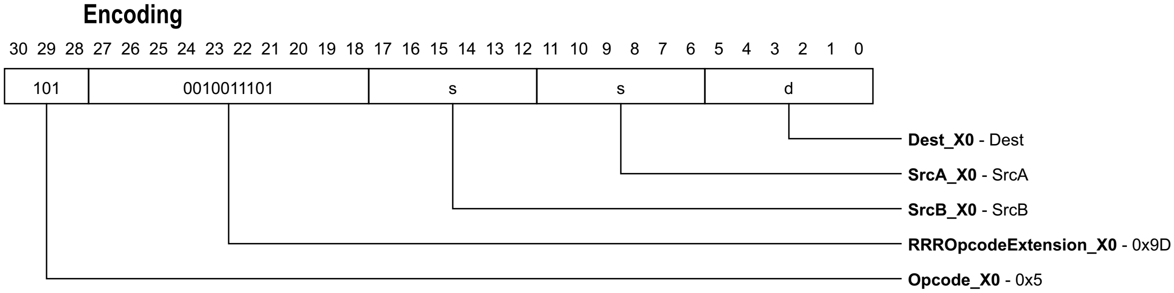

Figure 14-600: v4shru in X0 Bits Encoding

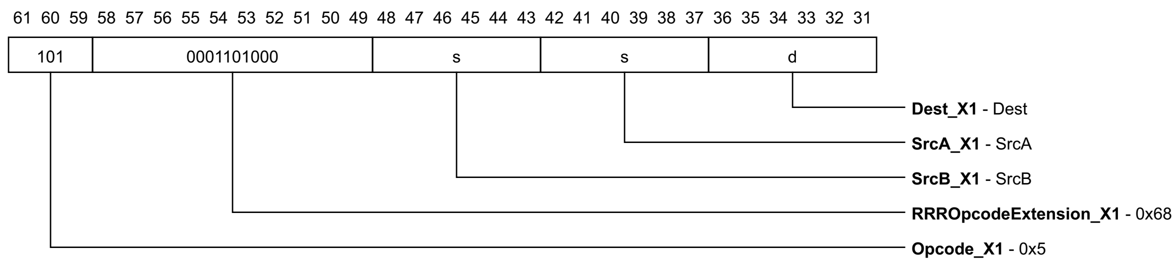

Figure 14-601: v4shru in X1 Bits Encoding

v4sub

Vector Four Byte Subtract

Syntax

```
v4sub Dest, SrcA, SrcB
```

Example

```
v4sub r5, r6, r7
```

Description

Subtract the two 32-bit quantities in the second source operand from the two 32-bit quantities in the first source operand.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
    {
        output =
            set4Byte (output, counter,
                (get4Byte (rf[SrcA], counter) -
                 get4Byte (rf[SrcB], counter)));
    }
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

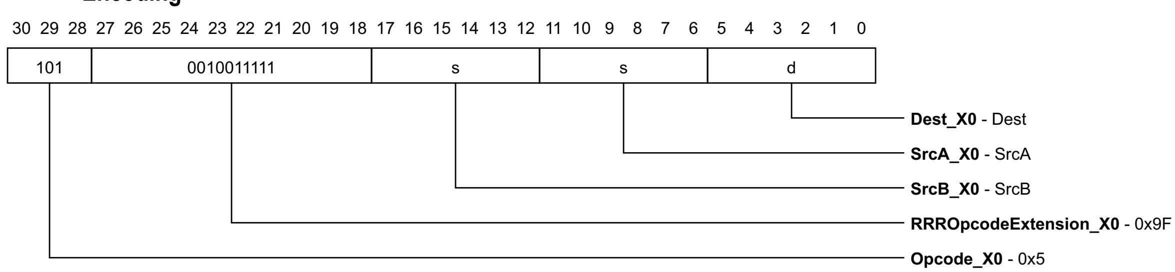

Figure 14-602: v4sub in X0 Bits Encoding

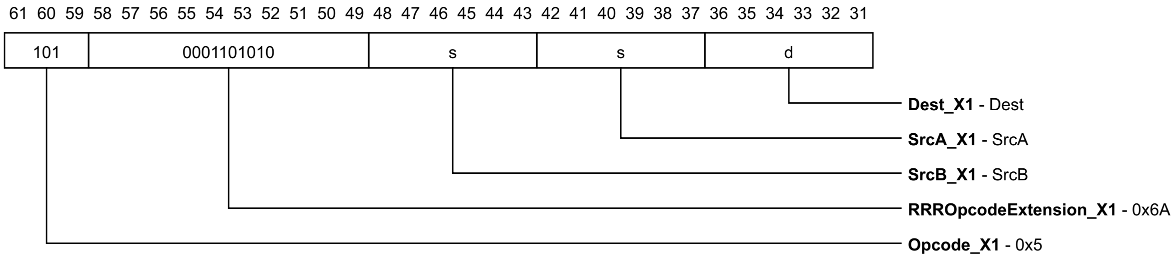

Figure 14-603: v4sub in X1 Bits Encoding

v4subsc

Vector Four Byte Subtract Signed Clamped

Syntax

```
v4subsc Dest, SrcA, SrcB
```

Example

```
v4subsc r5, r6, r7
```

Description

Subtract the two 32-bit quantities in the second source operand from the two 32-bit quantities in the first source operand and clamp each result to the minimum negative value or maximum positive value.

Functional Description

```
uint64_t output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / 32); counter++)
   {
      output =
         set4Byte (output, counter,
            signed_saturate32 (signExtend32 (get4Byte (rf[SrcA], counter))
               -
               signExtend32 (get4Byte
                           (rf[SrcB], counter))));
   }
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

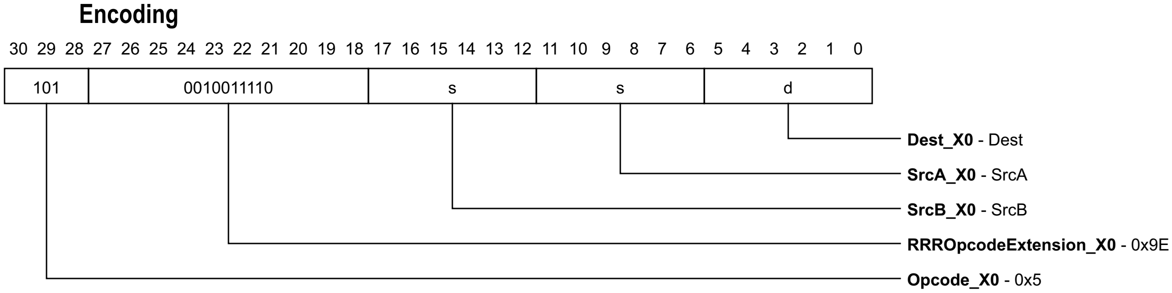

Figure 14-604: v4subsc in X0 Bits Encoding

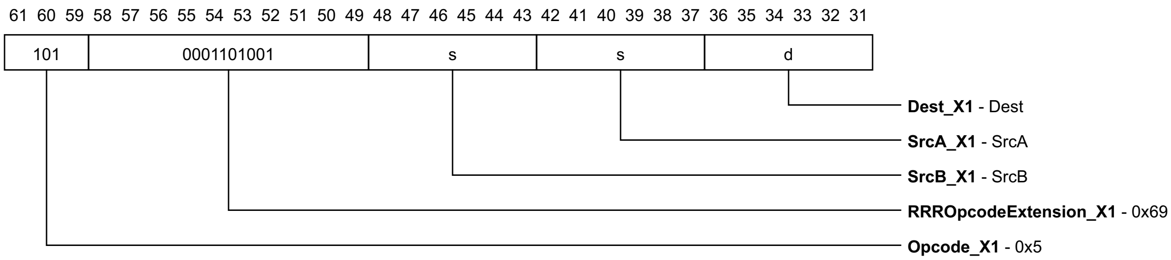

Figure 14-605: v4subsc in X1 Bits Encoding

CHAPTER 15 SYSTEM INSTRUCTIONS

15.1 Overview

The following sections provide detailed descriptions of system instructions listed alphabetically.

drain	Drain Instruction
icoh	Instruction Stream Coherence
ill	Illegal Instruction
iret	Interrupt Return
mfspr	Move from Special Purpose Register Word
mtspr	Move from Special Purpose Register Word
nap	Nap
swint0	Software Interrupt 0
swint1	Software Interrupt 1
swint2	Software Interrupt 2
swint3	Software Interrupt 3

15.2 Instructions

System instructions are described in the sections that follow.

drain

Drain Instruction

Syntax

```
drain
```

Example

```
drain
```

Description

Acts as a barrier that requires all previous instructions to complete before any subsequent instructions are executed. A `drain` instruction is dependent on all, program order, previous instructions. All, program order subsequent instructions are dependent on the `drain` instruction. Instructions in the same bundle as the `drain` instruction will produce unspecified results. The `drain` instruction also traverses the full length of any processor pipelining before subsequent instructions are executed. By traversing the length of any processor pipelining, the `drain` Instruction can be used to make state modifications to portions of the processor pipeline earlier than where the state modification takes place. The `drain` instruction does not post memory operations or serve as a Memory Fence. In order to guarantee memory ordering, a `mf` instruction is required.

Functional Description

```
drain ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

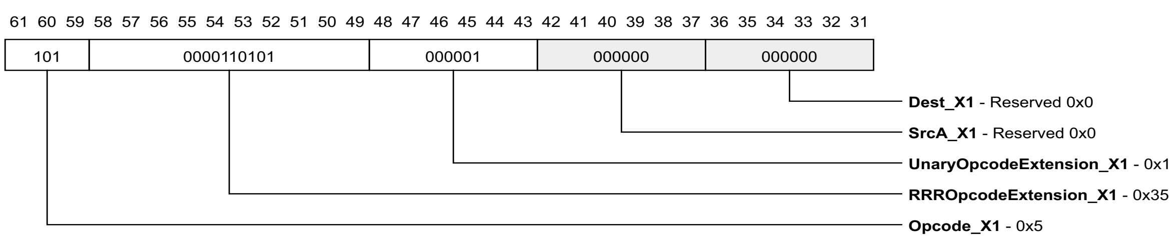

Figure 15-606: drain in X1 Bits Encoding

icoh

Instruction Stream Coherence

Syntax

```
icoh SrcA
```

Example

```
icoh r5
```

Description

Make the instruction stream coherent with the data stream for a particular cache index. Removes possible stale instructions from the instruction stream caching system. The source operand names a particular indexed set in the instruction cache. All of the blocks associated with the indexed set are removed from the icache. The `icoh` instruction minimally flushes words, but may operate on cache lines depending on the instruction cache implementation. One icoh instruction is guaranteed to minimally flush an aligned word of data from the instruction cache. The indexing of the instruction cache is the same as if the parameter of the instruction is interpreted as a 64-bit zero-extended physical address. If `icoh` is used in a loop that increments any address by words and loops `icoh` instructions over an address range up to the size of the implementation specific instruction cache size, then the entire instruction cache is cleared with the exception of the flushing loop. The `icoh` instruction needs to be used when data stores are made to a memory location which is to be executed later. Examples of this include self modifying code and physical page invalidates.

Functional Description

```
iCoherent (rf[SrcA]);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

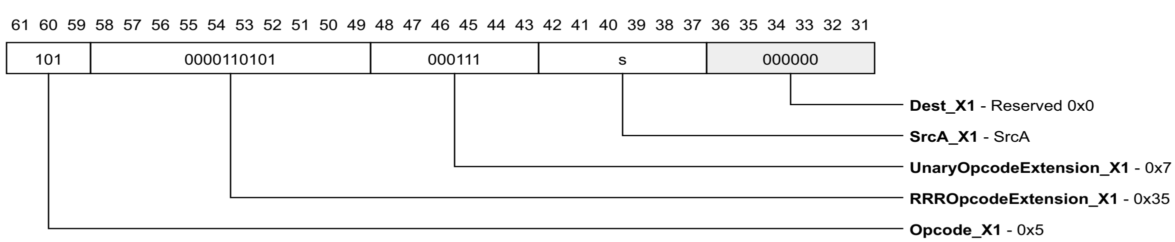

Figure 15-607: icoh in X1 Bits Encoding

ill

Illegal Instruction

Syntax

```
ill
```

Example

```
ill
```

Description

Causes an illegal instruction interrupt to occur. The `ill` Instruction is guaranteed to always cause an illegal instruction interrupt for all current and future derivations of the architecture.

Functional Description

```
illegalInstruction ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X		X	

Encoding

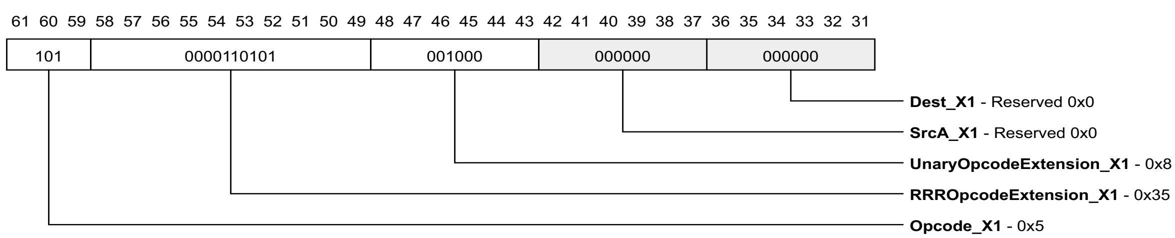

Figure 15-608: ill in X1 Bits Encoding

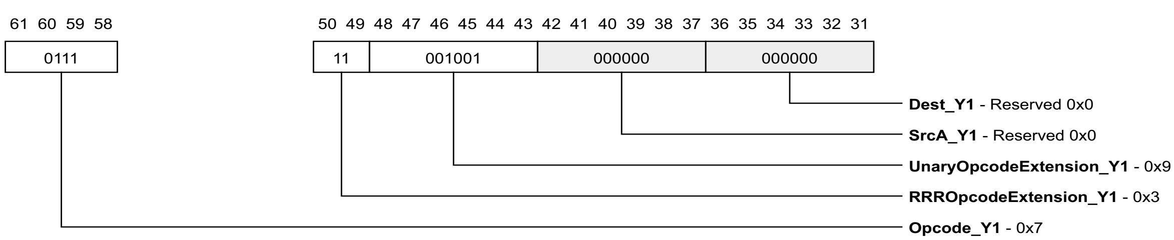

Figure 15-609: ill in Y1 Bits Encoding

iret

Interrupt Return

Syntax

```
iret
```

Example

```
iret
```

Description

Returns from an interrupt. Transfers control flow to the program counter location and protection level contained in the current PL's `EX_CONTEXT` registers, and restores the interrupt critical section bit to the value contained in those registers.

Functional Description

```
setNextPC (sprf
   [EX_CONTEXT_SPRF_OFFSET +
    (getCurrentProtectionLevel () * EX_CONTEXT_SIZE) +
    PC_EX_CONTEXT_OFFSET]);
branchPredictedIncorrect ();
setProtectionLevel ((sprf
            [EX_CONTEXT_SPRF_OFFSET +
             (getCurrentProtectionLevel () * EX_CONTEXT_SIZE) +
             PROTECTION_LEVEL_EX_CONTEXT_OFFSET]);
           setInterruptCriticalSection ((sprf[EX_CONTEXT_SPRF_OFFSET +
(getCurrentProtectionLevel () * EX_CONTEXT_SIZE) +
INTERRUPT_CRITICAL_SECTION_EX_CONTEXT_OFFSET]);/* besides the PC we need to set our
new protection level, and set the interrupt critical section bit atomically inside
of this instruction */
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

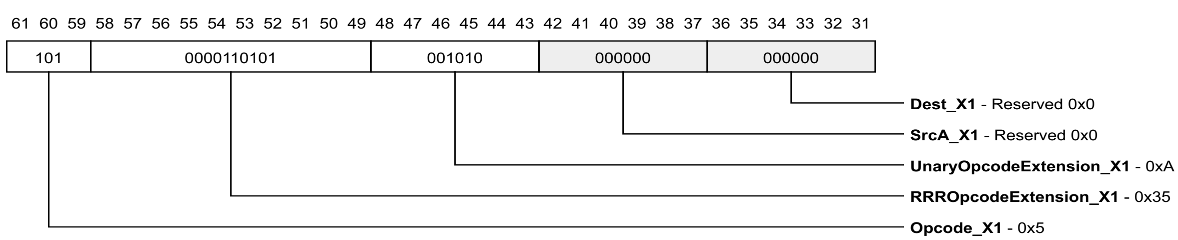

Figure 15-610: iret in X1 Bits Encoding

mfspr

Move from Special Purpose Register Word

Syntax

```
mfspr Dest, Imm14
```

Example

```
mfspr r6, 0x5
```

Description

Moves a word from a special purpose register. The special purpose register number is contained as an immediate and allows for the addressing of 2^{14} possible special purpose registers.

Functional Description

```
rf[Dest] = sprf[Imm14];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

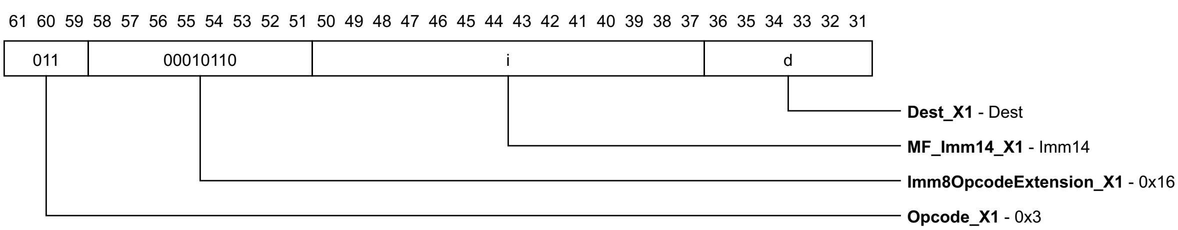

Figure 15-611: mfspr in X1 Bits Encoding

mtspr

Move to Special Purpose Register Word

Syntax

```
mtspr Imm14, SrcA
```

Example

```
mtspr 0x5, r6
```

Description

Moves a word to a special purpose register. The special purpose register number is contained as an immediate and allows for the addressing of 2^{14} possible special purpose registers.

Functional Description

```
sprf[Imm14] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

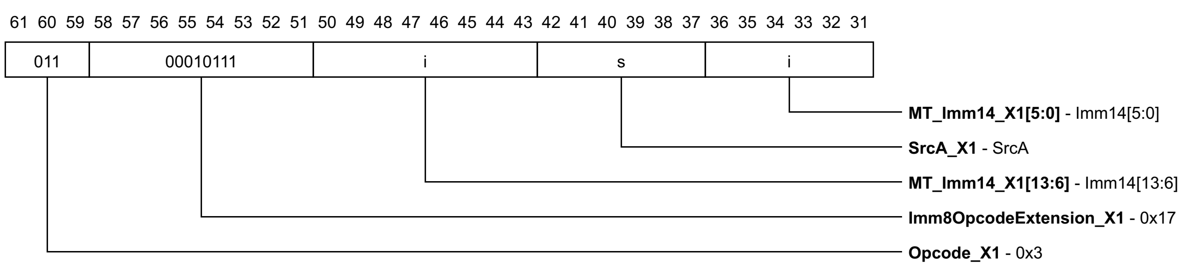

Figure 15-612: mtspr in X1 Bits Encoding

nap

Nap

Syntax

```
nap
```

Example

```
nap
```

Description

Enters a lower power state. This instruction may or may not complete. To guarentee continued naping on all implementations, this instruction should be used in a loop. Instructions in the same bundle as the `nap` instruction will produce unspecified results. If this instruction completes, this operation does not modify architectural state.

Functional Description

```
nap ();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

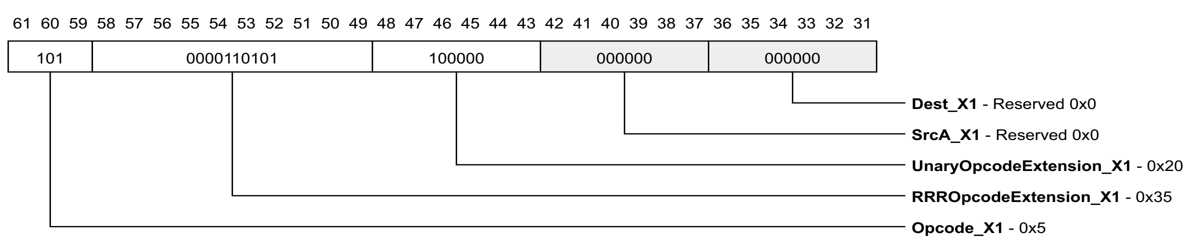

Figure 15-613: nap in X1 Bits Encoding

swint0

Software Interrupt 0

Syntax

```
swint0
```

Example

```
swint0
```

Description

Signals that a precise software interrupt should occur on this instruction to the Software Interrupt 0 interrupt handler. Instructions in the same bundle as the `swint0` instruction will produce unspecified results.

Functional Description

```
softwareInterrupt (0);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

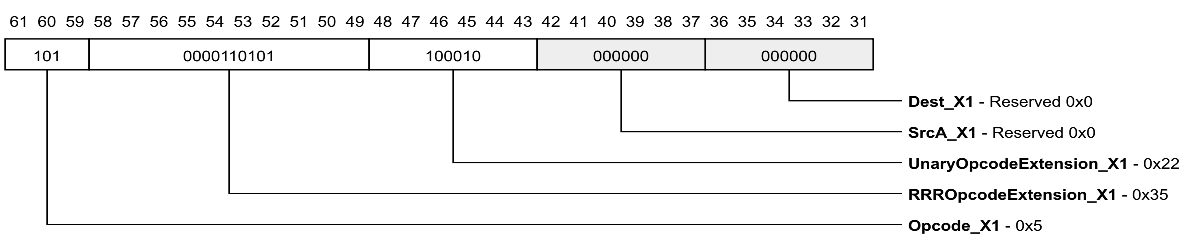

Figure 15-614: swint0 in X1 Bits Encoding

swint1

Software Interrupt 1

Syntax

`swint1`

Example

`swint1`

Description

Signals that a precise software interrupt should occur on this instruction to the Software Interrupt 1 interrupt handler. Instructions in the same bundle as the `swint1` instruction will produce unspecified results.

Functional Description

```
softwareInterrupt (1);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

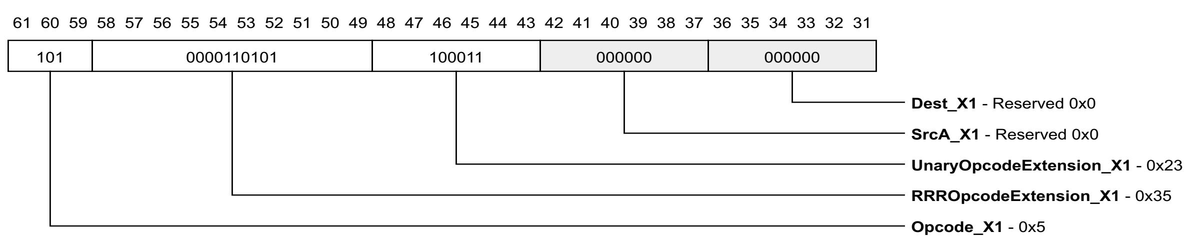

Figure 15-615: swint1 in X1 Bits Encoding

swint2

Software Interrupt 2

Syntax

`swint2`

Example

`swint2`

Description

Signals that a precise software interrupt should occur on this instruction to the Software Interrupt 2 interrupt handler. Instructions in the same bundle as the `swint2` instruction will produce unspecified results.

Functional Description

```
softwareInterrupt (2);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

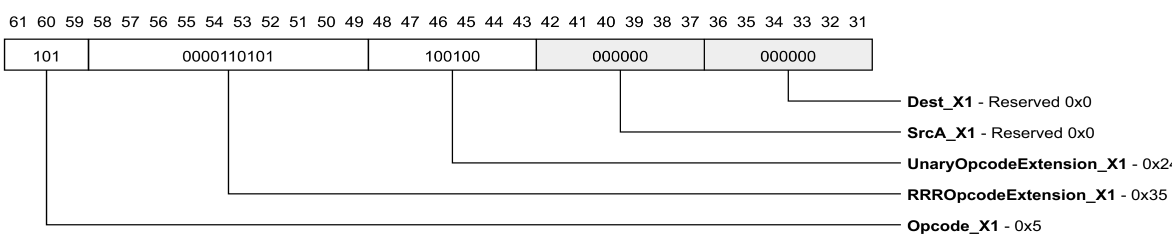

Figure 15-616: swint2 in X1 Bits Encoding

swint3

Software Interrupt 3

Syntax

```
swint3
```

Example

```
swint3
```

Description

Signals that a precise software interrupt should occur on this instruction to the Software Interrupt 3 interrupt handler. Instructions in the same bundle as the `swint3` instruction will produce unspecified results.

Functional Description

```
softwareInterrupt (3);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

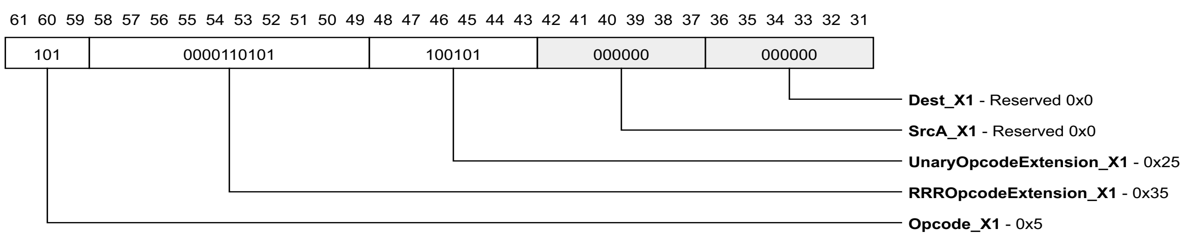

Figure 15-617: swint3 in X1 Bits Encoding

G GLOSSARY

Term	Definition
DDC™	Dynamic Distributed Cache. A system for accelerating multicore coherent cache subsystem performance. Based on the concept of a distributed L3 cache, a portion of which exists on each tile and is accessible to other tiles through the iMesh. A TLB directory structure exists on each tile — eliminating bottlenecks of centralized coherency management — mapping the locations of pages among the other tiles.
Dynamic Network	A network where the path of each message is determined at each switch point. The path of each message may be different, based on the contents of the message.
Hypervisor services	Provided to support two basic operations: install a new page table (performed on context switch), and flush the TLB (performed after invalidating or changing a page table entry). On a page fault, the client receives an interrupt, and is responsible for taking appropriate action (such as making the necessary data available via appropriate changes to the page table, or terminating a user program which has used an invalid address).
Little-endian byte ordering	More significant bytes are numbered with a higher byte address or byte number than less significant bytes (LSBs).
Multicore Development Environment™ (MDE™)	Multicore programming environment.
RAW Dependence	Read-after-Write dependence, or true dependence. RAW dependencies arise when a read operation on a location follows in program order a write operation to the same location. The read operation must receive the value from the most recent write operation, and must wait for the write operation to complete if the processor executes the operations simultaneously or out of order.
SIMD	Single Instruction Multiple Data. An architecture that allows a single instruction to apply to multiple sets of data. In the TILE-Gx Processor™, SIMD instructions allow a single instruction to operate on registers containing four bytes or two half-words.
VLIW architecture	VLIW (Very Long Instruction Word). A microprocessor design technology. A chip with VLIW technology is capable of executing many operations within one clock cycle. Essentially, a compiler reduces program instructions into basic operations that the processor can perform simultaneously. The operations are put into a very long instruction word that the processor then takes apart and passes the operations off to the appropriate devices.

I INDEX

E

F

G

I