TILERA®

TILE PROCESSOR USER ARCHITECTURE MANUAL

REL. 2.4
DOC. NO. UG101
NOVEMBER 2011
TILERA CORPORATION

Publishing Information:

Document Number:	UG101
Document Release	2.4
Date	10 November 2011

Contact Information:

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

CONTENTS

CHAPTER 4 PROCESSOR ENGINE INSTRUCTION SET

CHAPTER 5 MEMORY AND CACHE ARCHITECTURE

CHAPTER 6 ON-CHIP NETWORK ARCHITECTURE

1 USER ARCHITECTURE INTRODUCTION

1.1 Introduction to the Tile Processor Architecture

The Tile Processor™ is a new class of multicore processor that delivers unprecedented levels of performance, flexibility, and power efficiency in a highly integrated device. The Tile Processor is programmable in standard ANSI C, and implements Tilera's iMesh Multicore technology, enabling application scaling across multiple cores (or tiles).

Each tile is a full-featured processor core, and is capable of running an entire operating system. Every tile implements a 32-bit, three-wide integer processor engine with an instruction fetch unit, execution units, a memory management unit including Translation Lookaside Buffers (TLBs), a 64-entry register file, and a two-level cache hierarchy. Hardware maintains cache coherence for processor and I/O memory accesses.

The tiles in the Tile Processor are connected to each other, to the on-chip memory controllers, and to the on-chip I/O controllers by multiple independent mesh networks. Tilera's iMesh™ multicore technology enables the Tile Processor to provide performance scalability and high bandwidth/low latency communication between all on-chip components.

1.2 About this Manual

This manual is organized as follows:

- Chapter 1: User Architecture Introduction (*this* chapter) provides an overview of this manual.
- Chapter 2: Basic Architecture provides hierarchical overview of the Tilera Tile Processor Architecture.
- Chapter 3: Processor Engine Architecture describes the processor engine (PE) in detail.
- Chapter 4: Processor Engine Instruction Set describes the instruction set architecture and bundling rules and formats.
- Chapter 5: Memory and Cache Architecture describes how memory is structured and accessed.
- Chapter 6: On-Chip Network Architecture describes the User Dynamic Network (UDN), which is used by applications to send messages between tiles.
- Chapter 7: Static Network describes the structure and functions of the Static Network (STN).
- Chapter 8: User-level System Concerns describes the system call architecture used to implement system interactions and interrupts, and communication with I/O devices.
- Appendix A: Special Purpose Registers defines special instructions (Special Purpose Registers, or SPRs) that access different portions of system level state.
- Glossary defines terms used in this document.

1.3 What's New In This Manual

This manual has been revised as follows:

- Introduction to the TILE*Pro* family of processors
- New cache and memory architecture
- New instructions to support the TILE*Pro*™ family of processors can be found in Chapter 4: Processor Engine Instruction Set. These are:
  - adds: Add Word Saturating
  - dword_align: Double Word Align
  - subs: Subtract Word Saturating
  - lbadd: Load Byte and Add
  - lbadd_u: Load Byte Unsigned and Add
  - lhadd: Load Half Word and Add
  - lhadd_u: Load Half Word Unsigned and Add
  - lw_na: Load Word No Alignment Trap
  - lwadd: Load Word and Add
  - lwadd_na: Load Word No Alignment Trap and Add
  - sbadd: Store Byte and Add
  - shadd: Store Half Word and Add
  - swadd: Store Word and Add
  - wh64: Write Hint 64 Bytes
  - addbs_u: Add Bytes Saturating Unsigned
  - addhs: Add Half Words Saturating
  - packbs_u: Pack Half Words Saturating
  - packhs: Pack Half Words Saturating
  - subbs_u: Subtract Bytes Saturating Unsigned
  - subhs: Subtract Half Words Saturating

1.4 Conventions

The following section describes the notational conventions used in this document.

1.4.1 Byte and Bit Order

The Tile Processor Architecture is little endian. More significant bytes are always numbered with a higher number than less significant bytes (LSBs). When data is stored in memory, bytes that are of greater significance are stored in higher numbered memory addresses than bytes of less significance.

When sets of bits are described or displayed in this document, bits of higher significance are displayed to the left of bits with lower significance. For instance, if 32 bits are to be displayed and are numbered from 0 to 31, bit 31 is displayed to the left of bit 0. Bits numbered with a higher number have greater significance than bits with a lower number.

1.4.2 Reserved Fields

Unused bits in control or I/O registers are considered reserved (reserved 0). When bits labeled as reserved are read they are not guaranteed to return 0. Bits denoted as reserved, must be written as 0. Bits that are ignored by the hardware are explicitly called out as being write-ignored. Writing a non-0 value to a reserved field will cause the processor to enter an undefined state.

1.4.3 Numbering

The default numeric base used in this document is base ten, or decimal representation. Any use of a numeric without an explicitly base identifier is considered to be a decimal number. Hexadecimal numbering is used widely in this document. When a numeric is to be interpreted as a hexadecimal (base sixteen) number, the prefix "0x" is prepended to the number. For example, the number 74 can also be expressed as 0x4A when written in hexadecimal.

When ranges of bits are numbered as a subset of a larger set of ordered bits a bracket notation is used. The notation contains one or two numbers separated by a colon. If only one number is specified, the numbered bit position is the bit referenced. In example, if "bus" is a 32-bit bus that is numbered 31 to 0 and the text describes bit 5, bus[5] is the nomenclature used to signify that bit. Bit ranges are specified as two numbers, with the left number being the higher-order bit locations and right number being the lower-ordered bit location. Bit ranges are inclusive of the specified higher- and lower-ordered bit locations. This nomenclature is consistent with the default manner in which little-endian bit ranges are denoted. For example, if `word` is a 32-bit word numbered 31 to 0 and the text describes the bits from bit 5 through bit 20, the appropriate manner to denote that is `word[20:5]`.

Figure 1-1 shows an example of how bitfields are graphically presented in this document. Bits[31:21] are shown as reserved bits.

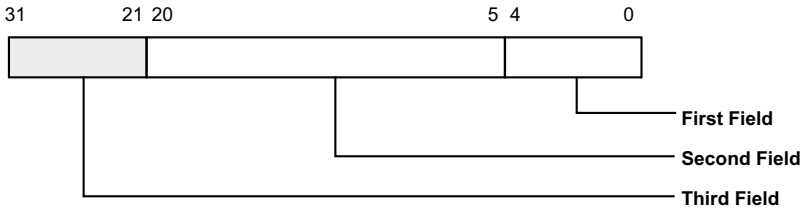

Figure 1-1: Bitfield Example

Figure 1-2 shows four bitfields that are logically represented along with a gap. The gap is not reserved, but is instead allocated for another use, and typically specified elsewhere.

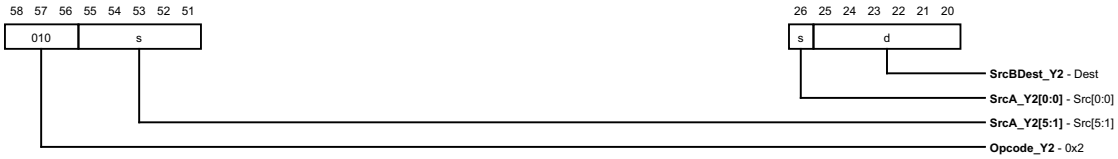

Figure 1-2: Bitfield Example with Fields Allocated by Other Functions

1.5 Implementation Dependence

This document describes the high-level Tile Processor Architecture and the microarchitecture of the Tile*Pro*64™ and Tile64™ implementations.

2 BASIC ARCHITECTURE

2.1 Architectural Overview

This section contains an overview of the Tilera Tile Processor™ Architecture.

The Tile Processor Architecture consists of tiles, input/output devices, and a communication fabric that connects them. Figure 2-3 shows the TILE64™/TILE*Pro*64™ Tile Processor with details of an individual tile's structure.

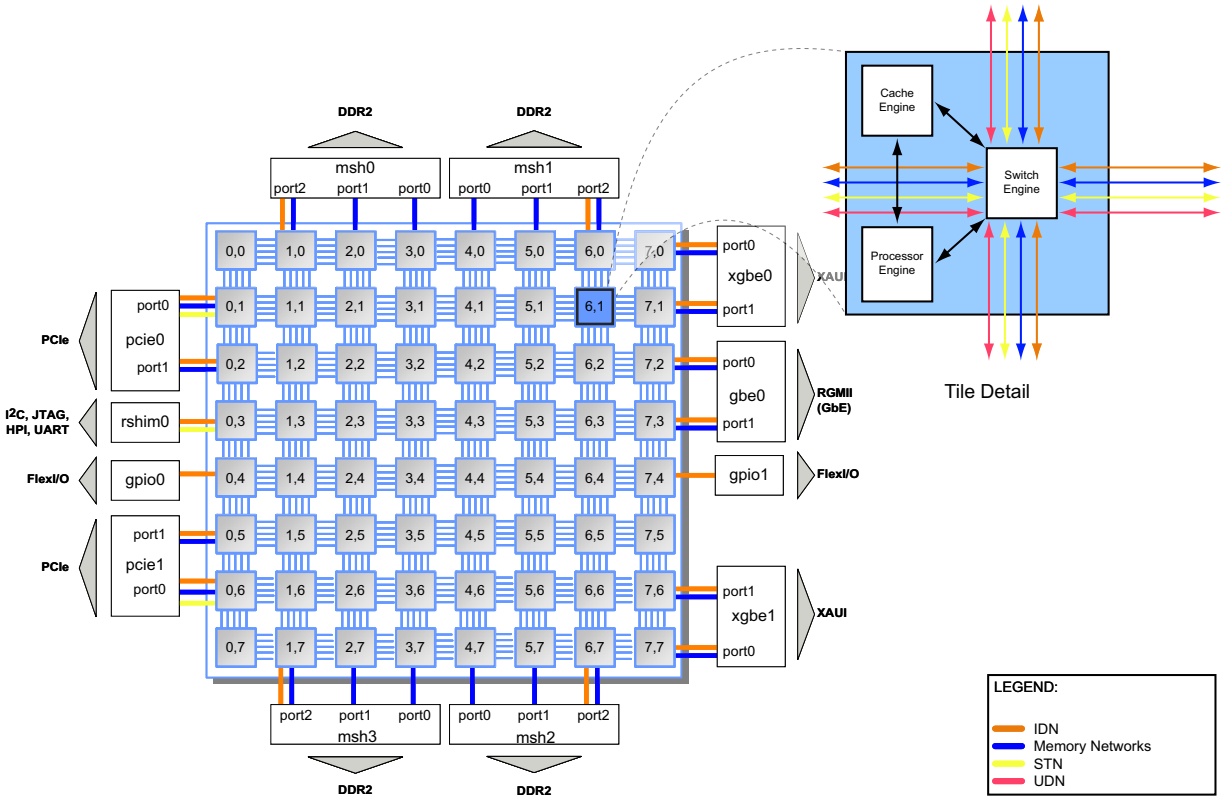

Figure 2-3: Tile Processor Hardware Architecture

2.1.1 Tile Architecture

The *tile* is the basic unit of replication in the Tile Processor Architecture. A key feature of a tile is that it is identical to all other tiles in a system. The fact that tiles are homogeneous eases automated mapping of programs to an array of tiles and allows for the arbitrary placement of programs across the homogeneous array. The tile is the main source of computational power within the Tile Processor Architecture. Each tile consists of a processor engine, a cache engine, and a switch engine. Figure 2-4 takes a view inside of a tile.

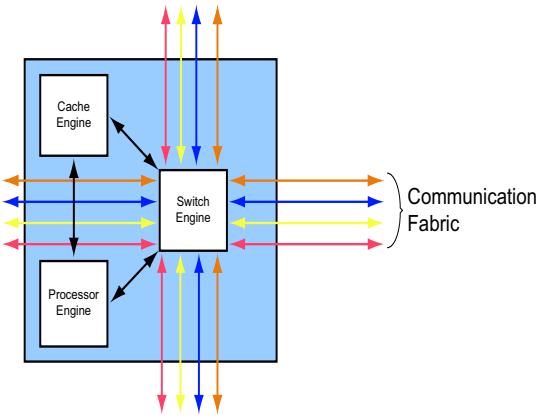

Figure 2-4: Basic Tile Architecture

2.1.1.1 Processor Engine

The processor engine consists of a fetch unit, instruction decoder, issue logic, general purpose register file, and special purpose registers. Figure 2-5 shows the basic architecture of the processor engine. The processor engine is a 32-bit, three instruction wide Very Long Instruction Word (VLIW) processor. Each VLIW bundle of instructions is 64 bits and is capable of encoding two or three instructions. The processor engine contains 56 general purpose registers, seven registers that interface to the on-chip iMesh networks, and one hard-wired zero register. While the stack pointer `sp` is included in the general purpose registers it is used only as a stack pointer by software convention.

The processor engine contains three instruction execution pipelines. The three pipelines that comprise the main processor are asymmetric, and are designated pipelines 0 through 2. Pipeline 0 is capable of executing any ALU operation, bit manipulation operations, select operations, multiply operations, and fused multiply-add operations. Pipeline 1 is capable of executing any ALU operation, special purpose register reads and writes, and control flow instructions (branches and jumps). Pipeline 2 is capable of executing load and store instructions and cache and memory maintenance instructions.

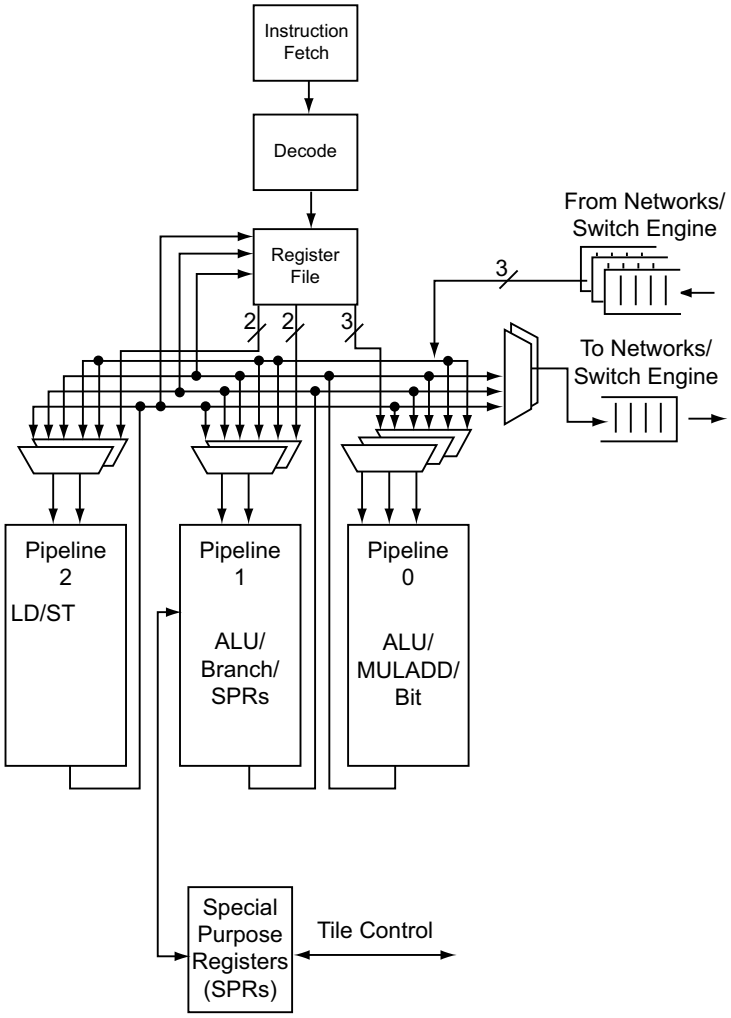

Figure 2-5: Processor Engine—Basic Architecture

2.1.1.2 Cache Engine

The tile's cache engine is responsible for handling caching of instructions and data, providing an interface to the memory system, translating memory addresses from virtual to physical addresses, and providing a coherent view of memory. The organization of the cache subsystem is implementation-dependent and the Tile Processor Architecture does not require a specific size or organization. For example, the cache organization found within the TILE64 processor provides an 8KB level 1 processor engine instruction cache, a two-way set associative 8KB level 1 data cache, and a two-way set associative 64KB unified level 2 cache. The TILE*Pro*64 processor provides a 16KB level 1 instruction cache, a two-way set associative 8KB level 1 data cache and a four-way set associative 64KB unified level 2 cache.

Figure 2-6 provides a conceptual block diagram of the processor/cache interface. When needed data is not found in the cache, the cache engine uses the on-chip networks to check for the data in other caches or in main memory.

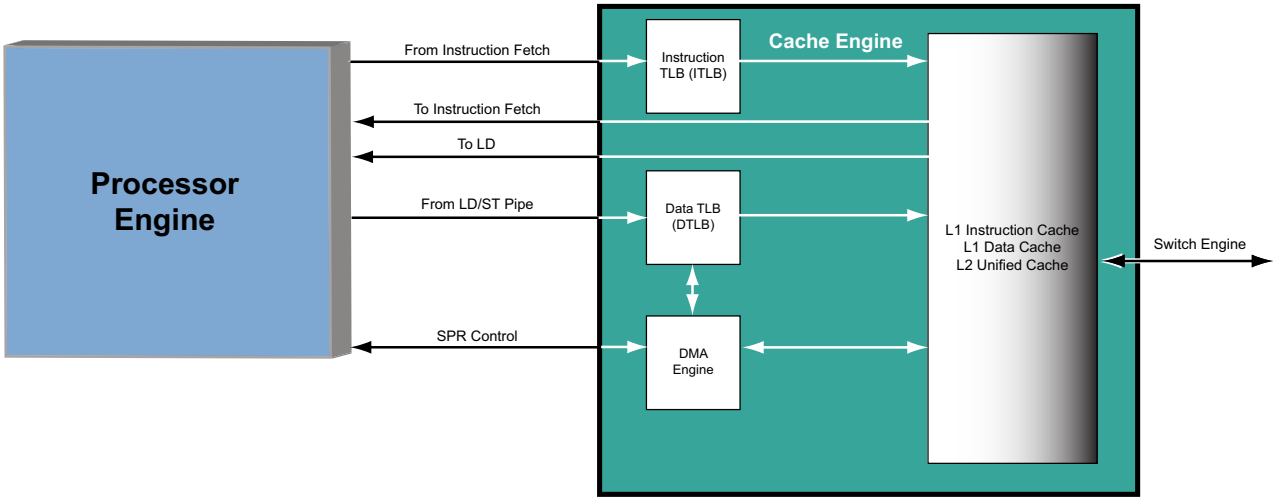

Figure 2-6: Cache Engine — Basic Architecture

The TILE64 and TILE*Pro* processors use two independent, dynamically-routed mesh networks to communicate with multiple memory controllers on the periphery of the chip and with other tiles. The Tile Processor Architecture supports virtual memory to supply protection and relocation of data structures stored in physical memory. The cache engine contains memory management units implemented via translation lookaside buffers (TLBs).

2.1.1.3 Switch Engine

Each tile contains a switch engine. The switch engine connects to neighboring tiles and I/Os (including the on-chip memory controllers) via the intra-tile iMesh. The tiles are laid out in a two dimensional grid, thus the switch engine connects to the neighbors to the north, south, east, and west. The switch engine connects directly to I/O devices if a tile is adjacent to an I/O device.

The switch engine is composed of multiple dynamic networks and a single static network. The Tile Processor Architecture contains three register-mapped architecturally-defined networks: the user dynamic network (UDN), the input/output dynamic network (IDN), and the static network (STN). In addition to the architecturally defined networks, TILE64 and TILE*Pro* contain hardware managed networks for communication with main memory and for inter-tile memory mapped communication.

Figure 2-7 shows an example network crosspoint with fully connected crossbar.

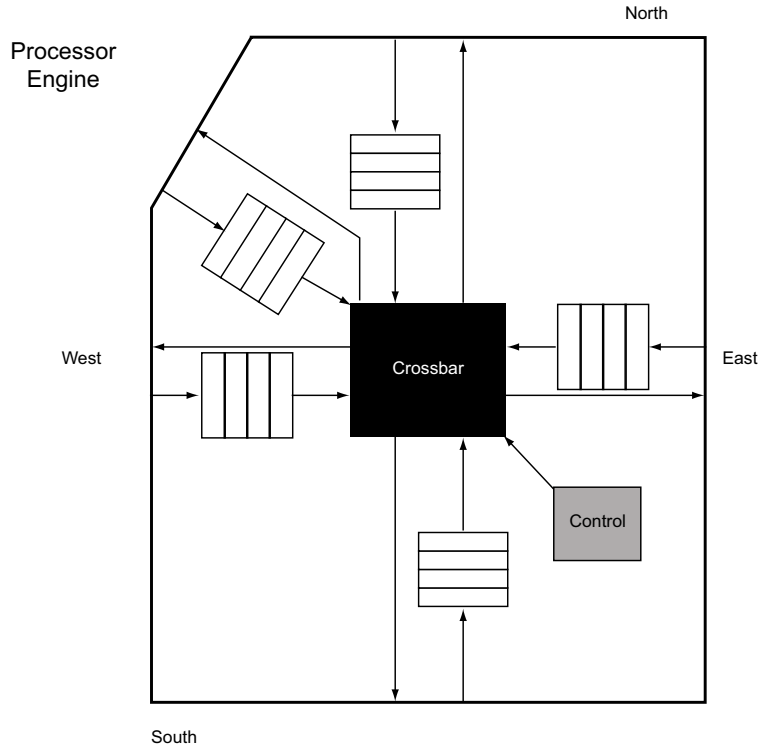

Figure 2-7: Switch Engine Architecture Implementing a Single Network

The UDN is primarily used by *user-level processes* to communicate with fast low-latency explicit messages. The Tilera software suite provides libraries to the developer to facilitate accessing the UDN with differing programming paradigms. The IDN is used by the system software to communicate with I/O devices and for tile-to-tile communication at the system level.

The static network is a scalar operand network designed to transport scalar values efficiently from one tile to another tile across the iMesh. The routing of the static network crossbar is controlled by the processor engine. The static network passes data to and from the main processor, and connects to the tiles to the north, south, east and west.

The static network is configured in a hard-coded routing mode, which specifies how data is to be routed from one port to another. Routing in the static network is atomic—a transfer that routes data stalls until input data is available and all the targeted output ports are free. The routing in the static network encodes the input direction that supplies data for each output direction, thereby allowing multicasting of data.

2.1.2 I/O Devices

The iMesh interconnect fabric extends from the periphery of the array of tiles to connect to I/O interfaces, which translate messaging packets into operations on the inputs and outputs of the chip. An I/O device can be connected to any iMesh network, but typically I/O devices are connected to the IO Dynamic Network (IDN) and memory networks. The arrangement of I/O devices and the way in which they are connected to tile-array ports is specific to a particular implementation.

The TILE*Pro*64 and TILE64 processors have the following on-chip interfaces: the TILE*Pro* processor implementation has the following on-chip interfaces: two 10Gbps XAUI, two PCIe 4x, two 10/100/1000 Gbps Ethernet, four DDR-2 64-bit memory interfaces, 64 general purpose I/Os, two-wire interface (I^2C-compatible), SPI, HPI, and a UART.

The typical makeup of an I/O interface can be seen in Figure 2-8. This example shows an I/O device being connected to a dynamic network port. On the left side of this figure, the I/O device is connected to buffering and a finite state machine for control. The finite state machine is the key portion of the I/O interface. The finite state machine receives dynamic messages from the fabric and parses the messages. In response to messages that it parses, the finite state machine acts accordingly by controlling the I/O device. Likewise, the finite state machine receives data and control requests from the I/O device and constructs messages destined for the tiles, memory, or other I/O interfaces. I/O shims typically contain buffering in order to provide end-to-end flow control.

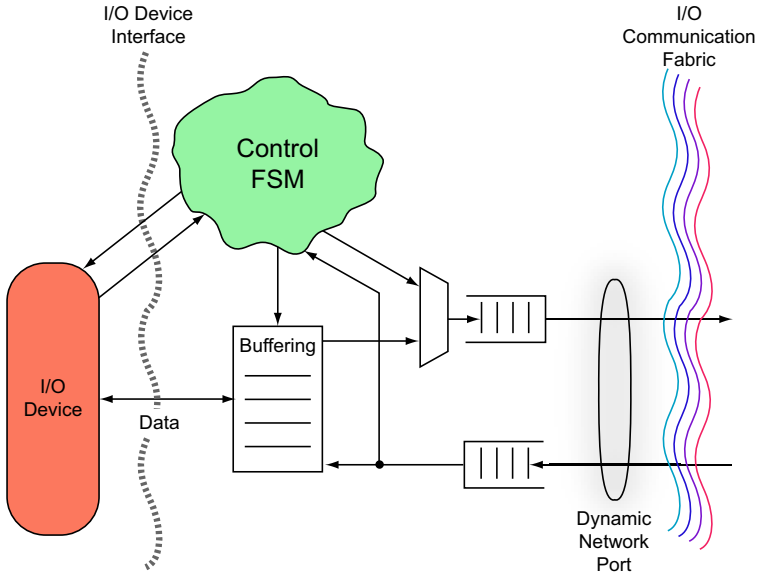

Figure 2-8: The Anatomy of a Basic I/O Interface

2.1.3 iMesh

An instantiation of the Tile Processor Architecture consists of a rectangular array of tiles and I/O devices. In order for the tiles to communicate with each other and to I/O devices, the Tile Processor Architecture provides a communication fabric called the iMesh™. The iMesh consists of the array of the switch engines, which are embedded inside each tile of the array, and the two-dimensional network that interconnects the engines and IO devices.

There are three types of networks: the static networks, architecturally-defined dynamic networks, and implementation-specific dynamic networks. The TILE64 has the two architecturally-defined dynamic networks (IDN/UDN), the static network (STN) and two implementation specific networks to interconnect the tile's cache engines and memory controllers. TILE*Pro*64 adds a third memory network for coherence traffic.

Each of these networks is logically 32-bits (one word) in width. Each switch engine contains multiple independent crossbars that each contain five connections. The five connections are north, south, east, west, and one connecting to the processor engine. Each connection consists of two uni-

directional links. For example, a UDN connection from a tile's switch engine to a neighboring tile's switch engine is logically 64-bits wide. Thirty-two bits are used for traffic leaving a tile for the tile to the east and 32-bits are used for traffic entering the tile from the easterly side.

I/O devices directly connect to switch engines of tiles on the periphery of the array via the iMesh. In Tile Processor Architecture implementations, the tiles are typically arranged in a rectangular two-dimensional array surrounded by I/O devices. The I/O devices connect on the periphery of the tile array to a set of networks that extend out of the tile array. I/O devices typically use a single connection to the IDN, but may be connected to any of the on-chip networks and may be connected to multiple tiles' networks in order to increase I/O to tile array bandwidth. I/O devices can also use the iMesh and tile switch engines to route traffic between one I/O device and another I/O device.

2.2 Data Types

Differing sized data types can be used on the Tile Processor Architecture. Data composed of 8 bits is considered a *byte*. Datum composed of 16 bits is considered a *Half Word*. Data composed of 32 bits is considered a *Word*, and data composed of 64 bits is considered a *Double Word*. In addition to these basic data types, the processor engine supports two packed data formats to be used with the SIMD instructions. These formats pack a number of smaller elements into a single word. The SIMD instructions support a *packed byte* format, which consists of four bytes packed into a *word*. The SIMD instructions also support a *Packed Half Word* format, which consists of two half words packed into a word. See "SIMD Instructions" on page 218 for more details.

2.3 Addressing

The Tile Processor architecture defines a flat, globally shared 64-bit physical address space and a 32-bit virtual address space. The TILE64 and TILE*Pro* family of processors implement a 36-bit physical address space. The globally shared physical address space provides the mechanism by which processes and threads can share instructions and data.

3 PROCESSOR ENGINE ARCHITECTURE

This section 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.

3.1 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. “Processor Engine Instruction Set” on page 19 discusses the encoding format and mix of instructions in greater detail.

3.2 Atomicity of Bundles

The Tile 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.

3.3 Register Set

The Tile Processor Architecture contains 64 architected registers. Each register is 32-bits wide. Of the 64 registers, some are general purpose registers and others allow access to the on-chip networks.

Table 3-1 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. Register `sn` allows access the static network. 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 `0` (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 3-1 presents the register identifier mapping.

Table 3-1. Register Numbers

Register Numbers	Short Name	Purpose
0 - 53	r0-r53	General Purpose Registers
54	sp	Stack Pointer
55	lr	Link Register
56	sn	Static Network
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 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, IDN, and STN. 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, all three networks (UDN, IDN, and STN) 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 mul-

tiple 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 3-1. contains example code for network reads and writes.

Listing 3-1. Network Reads and Writes

```
// add writes to udn0, sub reads
// idn0 and idn1 and writes to sn
{addi udn0, r5, 10; sub sn, 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}
```

The Tile Processor Architecture provides two methods of writing to the static network: by specifying `sn` as the destination register, and by setting the `S` bit within an encoded instruction. To set the `S` bit in an assembly program, add the suffix `.sn` to the mnemonic. The advantage of this method is that a GPR or another network-mapped register can be specified as the destination register, allowing both to be written simultaneously. This saves the programmer from having to add an extra instruction that explicitly writes to `sn`. The result of specifying `sn` as the destination register of an instruction with its `S` bit set is undefined. The `S`-bit only appears on a subset of the instructions, most notably, arithmetic instructions that execute in two-wide mode. With respect to obeying the above rules regarding bundle atomicity and network flow control, setting the `S` bit is identical to specifying sn as the destination register. When the `S`-bit is used, a move instruction to the network can be saved. Listing 3-2. contains example code for an instruction that writes the `STN` with an `S`-bit.

Listing 3-2. Writing an Instruction to the STN with an S-Bit

```
// Instruction adds r6 and r7 and
// deposits result in r5 and enqueues the
// result in the static network.
{add.sn r5, r6, r7}
```

3.4 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 95.

3.5 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 Processor Architecture. The SPR space is designed so that groups of SPRs require different protection levels to access it.

For more information, see “Special Purpose Registers” on page 391.

3.6 TILE64 and TILE*Pro* Processing Engine Pipeline

The Tile 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`.

3.6.1 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.

3.6.2 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.

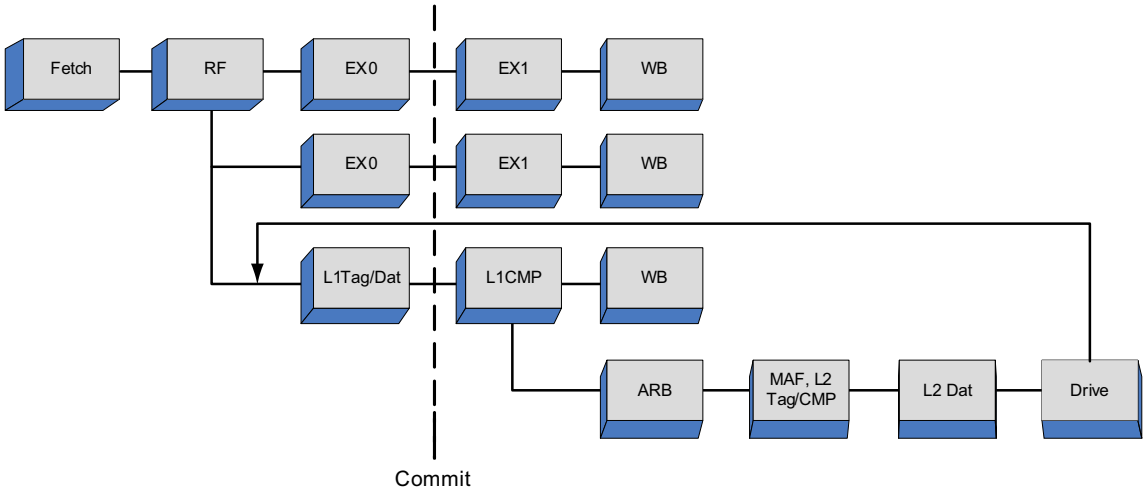

Figure 3-9: Processor Pipeline

3.6.3 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.

3.6.4 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.

3.6.5 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 3-2. TILEPro Pipeline Latencies

Operation	Latency
Branch Mispredict	2 cycles
Load to Use - L1 hit	2 cycles
Load to Use - L1 miss, L2 hit	8 cycles

Table 3-2. TILEPro Pipeline Latencies (continued)

Operation	Latency
Load to Use - L1/L2 Miss, adjacent Dynamic Distributed Cache (DDC™) hit	35 cycles
Load to Use - L1/L2 Miss, DDR2 page open, typical	69 cycles
Load to Use - L1/L2 Miss, DDR2 page miss, typical	88 cycles
MUL*, SAD*, ADIFF instructions	2 cycles
All other instructions	1 cycle

4 PROCESSOR ENGINE INSTRUCTION SET

4.1 Overview

This chapter describes the Instruction Set Architecture (ISA), the formats used to specify instructions, definitions and semantics, constants, and pipeline latencies. For a complete list of instructions, refer to "Master List of Main Processor Instructions" on page 35.

4.1 Instruction Set Architecture

The Tile Processor Architecture instructions can be categorized into 11 major groups:

- Arithmetic Instructions
- Bit Manipulation Instructions
- Compare Instructions
- Control Instructions
- Logical Instructions
- Memory Instructions
- Memory Maintenance Instructions
- Multiply Instructions
- NOP Instructions
- SIMD Instructions
- System Instructions

4.1.1 Instruction Organization and Format

The Tile 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 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` bit, bit 63. When the `Mode` bit is one (1), the bundle format is a Y bundle and when the `Mode` bit is zero (0), the bundle is an X bundle.

Instruction formats are described in the sections that follow.

4.1.1.1 X Instruction Formats

Figure 4-10 and Figure 4-11 show the basic X format instruction encodings.

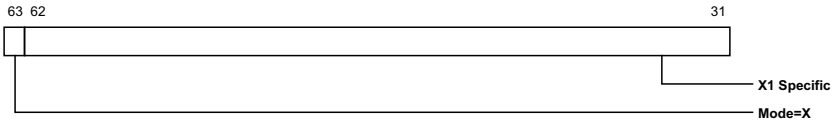

Figure 4-10: X1 Specific Format

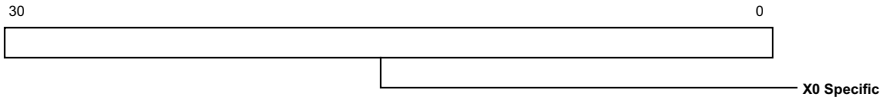

Figure 4-11: X0 Specific Format

Bundles that are in the Y format can encode three simultaneous operations where one is a memory operation, one is a arithmetic 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 and these instructions are not capable of writing to both the static network and a register in a single instruction (Y mode instructions lack `s` bits). 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. Also, many instructions in X mode bundles have `s` bits that indicate that the instruction writes to the static network in addition to the destination register specified in the instruction. For more information on the `s`-bit, refer to page 15.

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 and logical instructions, and Y0 is capable of executing multiply, arithmetic, and logical instructions. Figure 4-30 through Figure 4-38 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 4-12 through Figure 4-27 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.

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
```

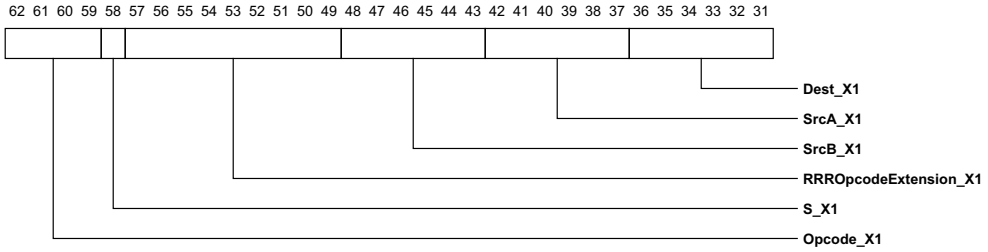

Figure 4-12: X1 RRR Format (X1_RRR)

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
```

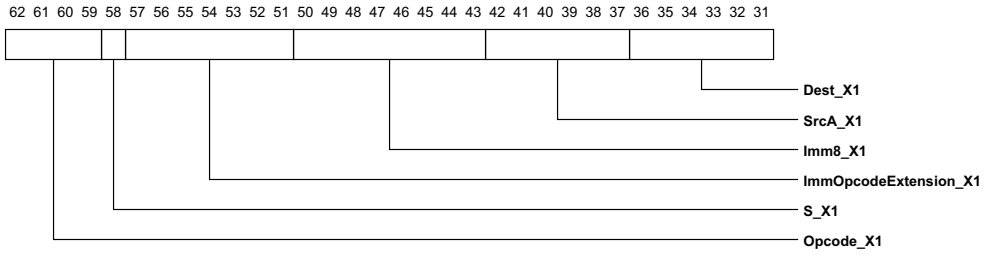

Figure 4-13: X1 Immediate Format (X1_Imm8)

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_SNSTATIC
{ mtspr SPR_SNSTATIC, r0 }
```

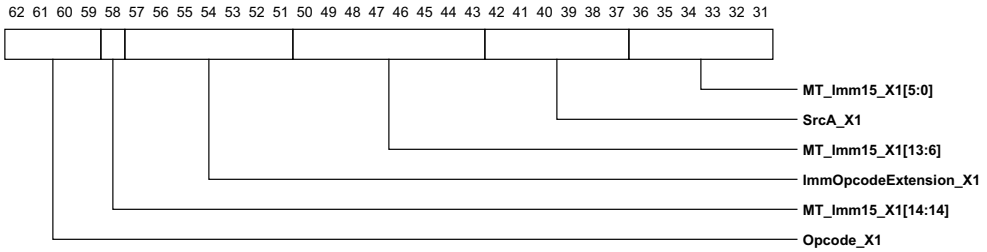

Figure 4-14: X1 Immediate MTSPR Format (X1_MT_Imm15)

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

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

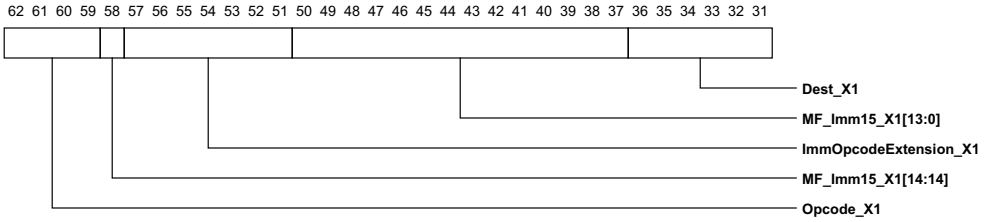

Figure 4-15: X1 Immediate MFSPR Format (X1_MF_Imm15)

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 }
```

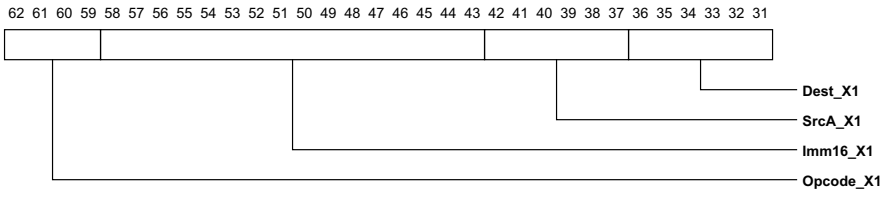

Figure 4-16: X1 Long Immediate Format (X1_Imm16)

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

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

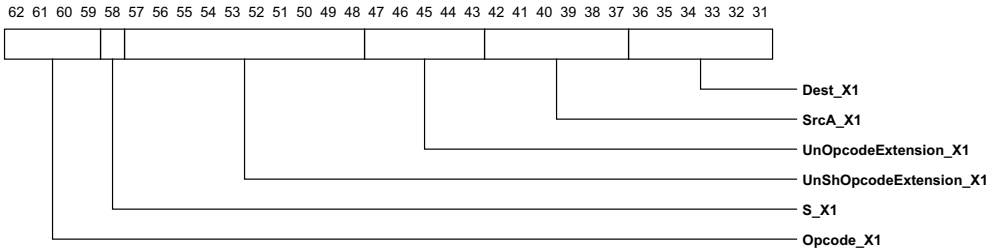

Figure 4-17: X1 Unary Format (X1_Unary)

The X1 Shift Format is used for instructions that require a destination register, a source register, and a 5-bit shift count. For example:

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

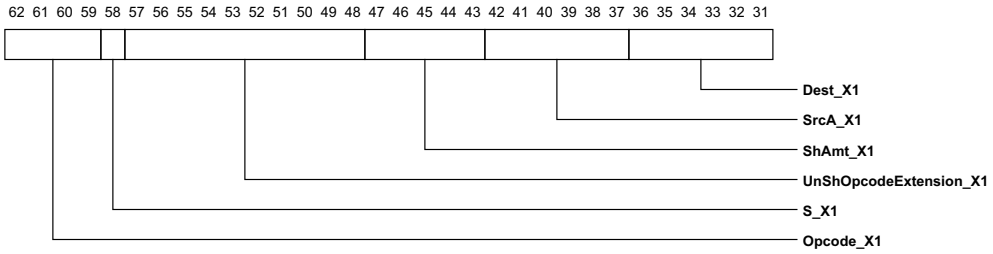

Figure 4-18: X1 Shift Format (X1_Shift)

The X1 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 }
```

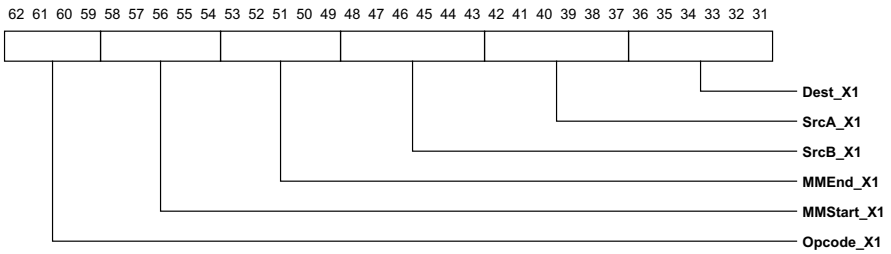

Figure 4-19: X1 Masked Merge Format (X1_MM)

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

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

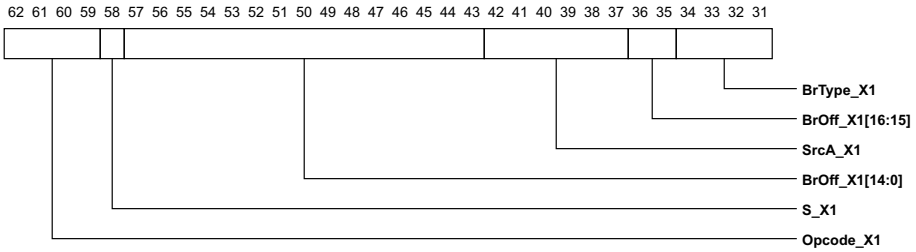

Figure 4-20: X1 Branch Format (X1_Br)

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
```

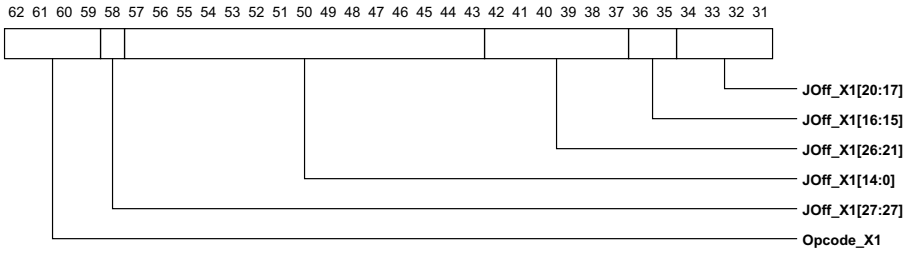

Figure 4-21: X1 Jump Format (X1_J)

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
```

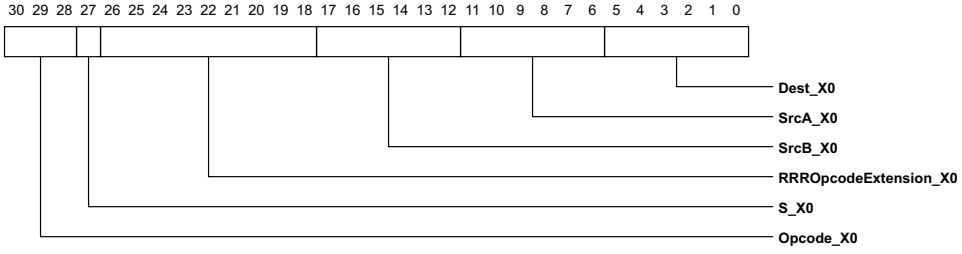

Figure 4-22: X0 RRR Format (X0_RRR)

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
```

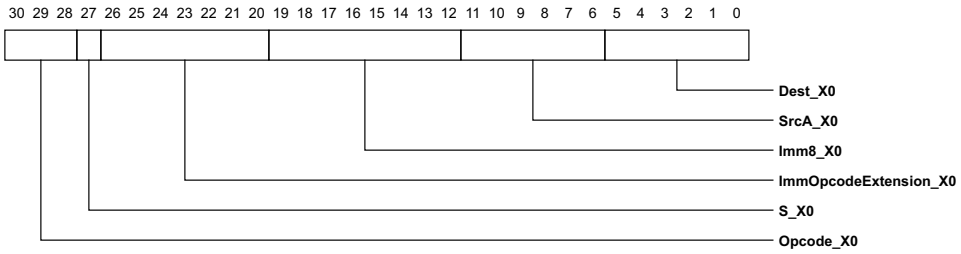

Figure 4-23: X0 Immediate Format (X0_Imm8)

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 }
```

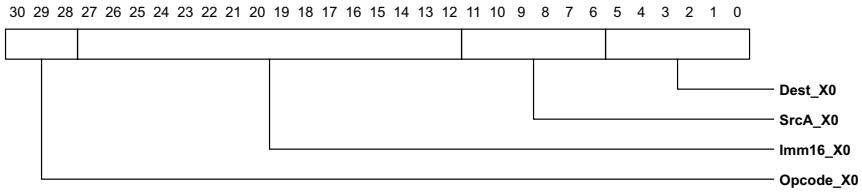

Figure 4-24: X0 Long Immediate Format (X0_Imm16)

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

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

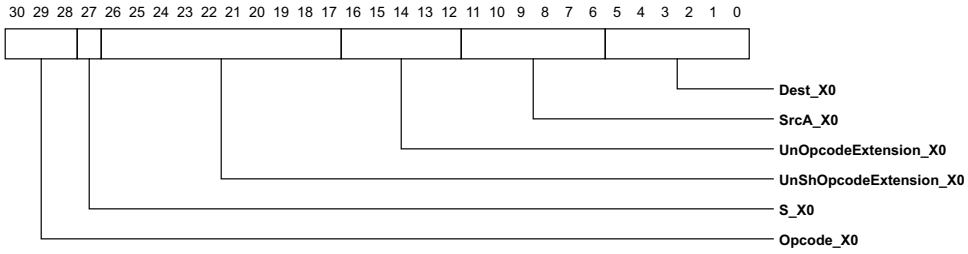

Figure 4-25: X0 Unary Format (X0_Unary)

The X0 Shift Format is used for instructions that require a destination register, a source register, and a 5-bit shift count. For example:

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

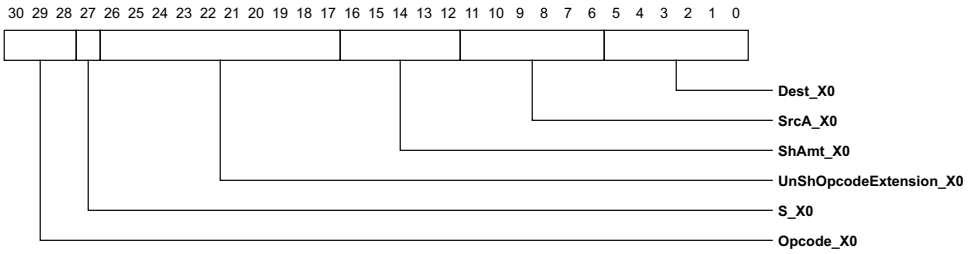

Figure 4-26: X0 Shift Format (X0_Shift)

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 }
```

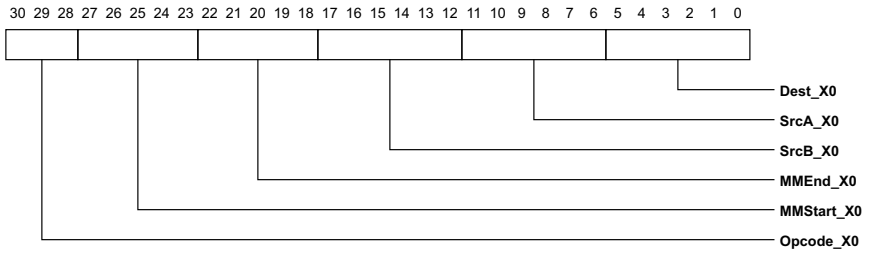

Figure 4-27: X0 Masked Merge Format (X0_MM)

4.1.1.2 Y Instruction Formats

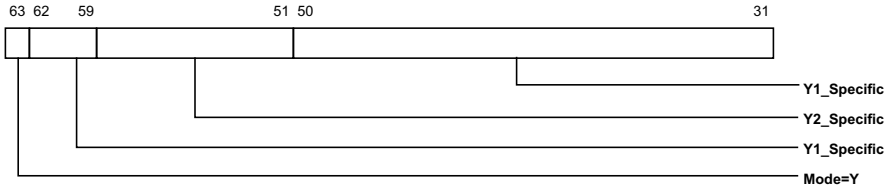

Figure 4-28: Y1 Specific Format

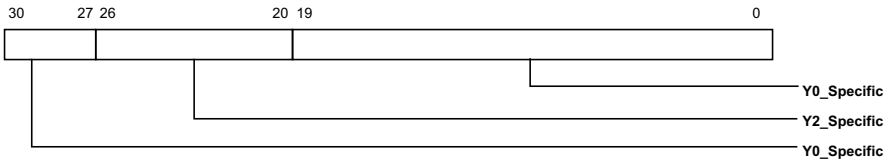

Figure 4-29: Y0 Specific Format

Y2 Instruction Formats

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

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

Figure 4-30: 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
```

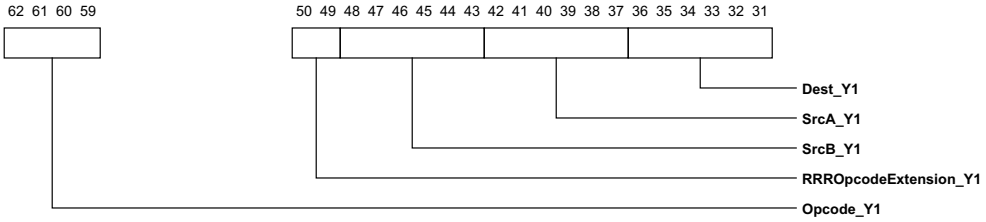

Figure 4-31: Y1 RRR Format (Y1_RRR)

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
```

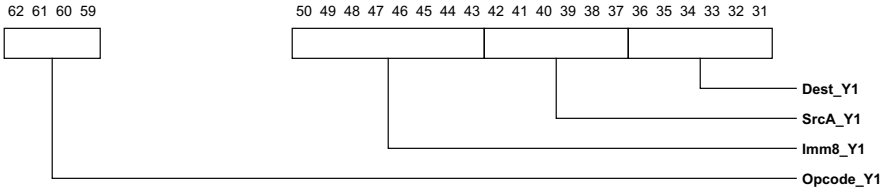

Figure 4-32: Y1 Immediate Format (Y1_Imm8)

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

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

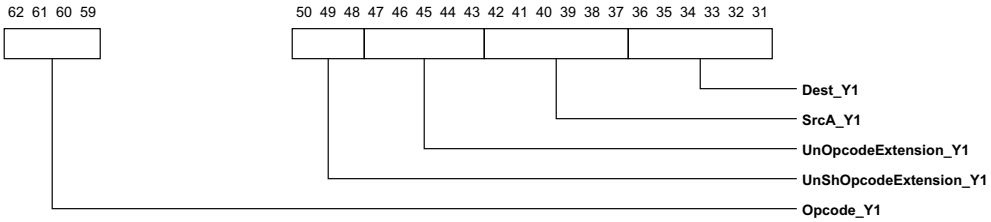

Figure 4-33: Y1 Unary Format (Y1_Unary)

The Y1 Shift Format is used for instructions that require a destination register, a source register, and a 5-bit shift count. For example:

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

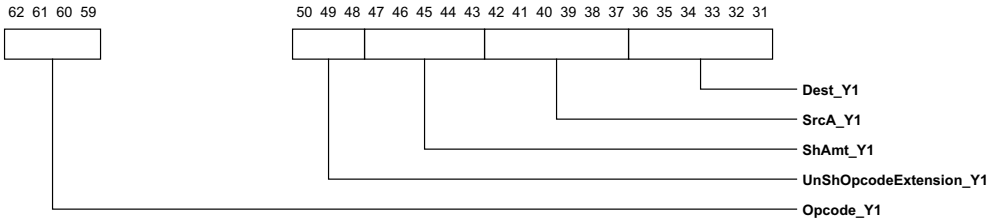

Figure 4-34: Y1 Shift Format (Y1_Shift)

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
```

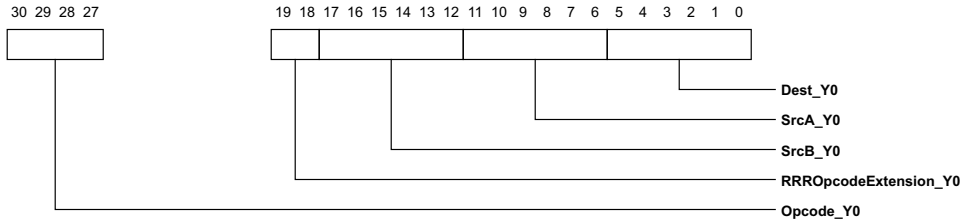

Figure 4-35: Y0 RRR Format (Y0_RRR)

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
```

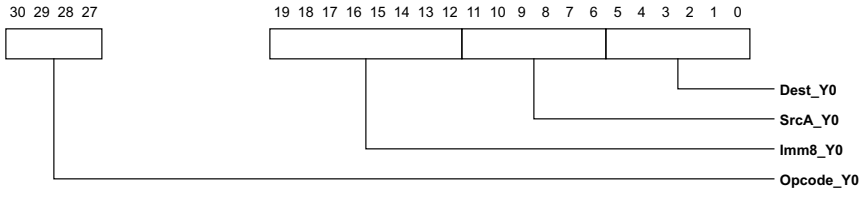

Figure 4-36: Y0 Immediate Format (Y0_Imm8)

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

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

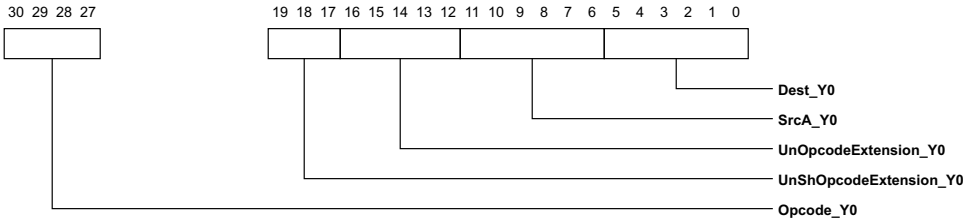

Figure 4-37: Y0 Unary Format (Y0_Unary)

The Y0 Shift Format is used for instructions that require a destination register, a source register, and a 5-bit shift count. For example:

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

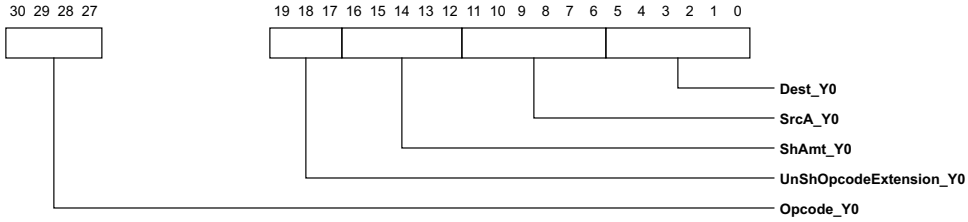

Figure 4-38: Y0 Shift Format (Y0_Shift)

4.1.2 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.

4.1.2.1 Constants

`WORD_SIZE 32`	The size of a machine word in bits. The Tile Processor is a 32-bit machine.
`WORD_MASK 0xFFFFffff`	A mask to represent all of the bits in a word.
`WORD_ADDR_MASK 0xFFFFfffc`	A mask that represents the portion of an address that forms a word aligned mask.
`HALF_WORD_SIZE 16`	The size of half of a machine word in bits. The Tile Processor is a 32-bit machine thus half the word length is 16.

HALF_WORD_ADDR_MASK 0xFFFFfffe	A mask that represents the portion of an address that forms a half 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.
BACKWARD_OFFSET 0x80000000	A constant address offset added to the instruction specified offset in backwards jump instructions. For more information, refer to "Control Instructions" on page 95.
INSTRUCTION_SIZE 64	The length in bits of an instruction (bundle) in the Tile Processor architecture.
INSTRUCTION_SIZE_LOG_2 6	The logarithm base 2 of the length in bits of an instruction (bundle) in the Tile Processor.
ALIGNED_INSTRUCTION_MASK 0xFFFFfff8	A mask that selects the relevant bits for the address of an aligned instruction.
BYTE_16_ADDR_MASK 0xFFFFfff0	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.

4.1.2.2 Types

`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`.

4.1.2.3 Functions

`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;`
`memoryReadWord`	Returns the value stored in memory of length `WORD_SIZE` at the address passed to this function. The value is not actually extended since it is already the same as `WORD_SIZE/UnsignedMachineWord`. The address passed as a parameter to this function is processed depending on the memory mode and contents of the TLB. The Tile Processor is a little endian machine.
`memoryReadHalfWord`	Returns the value stored in memory of length `HALF_WORD_SIZE` at the address passed to this function. This function returns the value 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 Processor is a little endian machine.
`memoryReadByte`	Returns the value stored in memory of length `BYTE_SIZE` at the address passed to this function. This function returns the value zero 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 Processor is a little endian machine.

`memoryWriteWord`	Writes to memory `WORD_SIZE` bits 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 Processor is a little endian machine.
`memoryWriteHalfWord`	Writes to memory `HALF_WORD_SIZE` bits 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 Processor is a little endian machine.
`memoryWriteByte`	Writes to memory `BYTE_SIZE` bits 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 Processor is a little endian machine.
`setNextPC`	Set the program counter to this function's parameter.
`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 that 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.
`dtlbProbe`	See "dtlbpr: Data TLB Probe" on page 184.
`memoryFence`	See "mf: Memory Fence" on page 188.
`getHighHalfWordUnsigned`	Returns the high-order half word of the parameter.
`getLowHalfWordUnsigned`	Returns the low-order half word of the parameter.
`iCoherent`	See "icoh: Instruction Stream Coherence" on page 349.
`fnop`	See "fnop: Filler No Operation" on page 214.
`nop`	See "nop: Architectural No Operation" on page 216.
`drain`	See "drain: Drain Instruction" on page 348.
`illegalInstruction`	Denotes that an illegal instruction has occurred.
`nap`	See "nap: Nap" on page 354.
`softwareInterrupt`	Denotes that a software interrupt has occurred. The parameter specifies which software interrupt will be generated.

4.1.3 Master List of Main Processor Instructions

Table 4-3 provides a complete list instructions in alphabetic order. Pseudo Instructions are listed on page 359.

Table 4-3. Master List of Main Processor Instructions

Register	Type	Description
add	Arithmetic	Add Word (Refer to page 44.)
addb	SIMD	Add Bytes (Refer to page 220.)
addbs_u	SIMD	Add Bytes Saturating Unsigned (Refer to page 222.)
addh	SIMD	Add Half Words (Refer to page 224.)
addhs	SIMD	Add Half Words (Refer to page 226.)
addi	Arithmetic	Add Immediate Word (Refer to page 46.)
addib	SIMD	Add Immediate Bytes (Refer to page 228.)
addih	SIMD	Add Immediate Half Words (Refer to page 229.)
addli	Arithmetic	Add Long Immediate Word (Refer to page 48.)
addlis	Arithmetic	Add Long Immediate Static Write Word (Refer to page 49.)
adds	Arithmetic	Add Word Saturating (Refer to page 50.)
adiffb_u	SIMD	Absolute Difference Unsigned Bytes (Refer to page 231.)
adiffh	SIMD	Absolute Difference Half Words (Refer to page 232.)
and	Logical	And Word (Refer to page 122.)
andi	Logical	And Immediate Word (Refer to page 124.)
auli	Arithmetic	Add Upper Long Immediate Word (Refer to page 52.)
avgb_u	SIMD	Average Byte Unsigned (Refer to page 233.)
avgh	SIMD	Average Half Words (Refer to page 234.)
bbns	Control	Branch Bit Not Set Word (Refer to page 96.)
bbnst	Control	Branch Bit Not Set Taken Word (Refer to page 97.)
bbs	Control	Branch Bit Set Word (Refer to page 98.)
bbst	Control	Branch Bit Set Taken Word (Refer to page 99.)
bgez	Control	Branch Greater Than or Equal to Zero Word (Refer to page 100.)
bgezt	Control	Branch Greater Than or Equal to Zero Predict Taken Word (Refer to page 101.)
bgz	Control	Branch Greater Than Zero Word (Refer to page 102.)

Table 4-3. Master List of Main Processor Instructions (continued)

Register	Type	Description
bgzt	Control	Branch Greater Than Zero Predict Taken Word (Refer to page 103.)
bitx	Bit Manipulation	Bit Exchange Word (Refer to page 64.)
blez	Control	Branch Less Than or Equal to Zero Word (Refer to page 104.)
blezt	Control	Branch Less Than or Equal to Zero Taken Word (Refer to page 105.)
blz	Control	Branch Less Than Zero Word (Refer to page 106.)
blzt	Control	Branch Less Than Zero Taken Word (Refer to page 107.)
bnz	Control	Branch Not Zero Word (Refer to page 108.)
bnzt	Control	Branch Not Zero Predict Taken Word (Refer to page 109.)
bytex	Bit Manipulation	Byte Exchange Word (Refer to page 66.)
bz	Control	Branch Zero Word (Refer to page 110.)
bzt	Control	Branch Zero Predict Taken Word (Refer to page 111.)
clz	Bit Manipulation	Count Leading Zeros Word (Refer to page 68.)
crc32_32	Bit Manipulation	CRC32 32-bit Step (Refer to page 70.)
crc32_8	Bit Manipulation	CRC32 8-bit Step (Refer to page 71.)
ctz	Bit Manipulation	Count Trailing Zeros Word (Refer to page 72.)
align	Bit Manipulation	Double Word Align (Refer to page 74.)
drain	System	Drain Instruction (Refer to page 348.)
dtlbpr	Memory Maintenance	Data TLB Probe (Refer to page 184.)
finv	Memory Maintenance	Flush and Invalidate Cache Line (Refer to page 185.)
flush	Memory Maintenance	Flush Cache Line (Refer to page 186.)
fnop	NOP	Filler No Operation (Refer to page 214.)
icoh	System	Instruction Stream Coherence (Refer to page 349.)
ill	System	Illegal Instruction (Refer to page 350.)
inthb	SIMD	Interleave High Byte (Refer to page 235.)
inthh	SIMD	Interleave High Half Words (Refer to page 237.)
intlb	SIMD	Interleave Low Byte (Refer to page 239.)
intlh	SIMD	Interleave Low Half Words (Refer to page 241.)
inv	Memory Maintenance	Invalidate Cache Line (Refer to page 187.)
iret	System	Interrupt Return (Refer to page 351.)

Table 4-3. Master List of Main Processor Instructions (continued)

Register	Type	Description
jalb	Control	Jump and Link Backward (Refer to page 112.)
jalf	Control	Jump and Link Forward (Refer to page 113.)
jalr	Control	Jump and Link Register (Refer to page 114.)
jalrp	Control	Jump and Link Register Predict (Refer to page 115.)
jb	Control	Jump Backward (Refer to page 116.)
jf	Control	Jump Forward (Refer to page 117.)
jr	Control	Jump Register (Refer to page 118.)
jrp	Control	Jump Register Predict (Refer to page 119.)
lb	Memory	Load Byte (Refer to page 164.)
lb_u	Memory	Load Byte Unsigned (Refer to page 165.)
lbadd	Memory	Load Byte and Add (Refer to page 166.)
lbadd_u	Memory	Load Byte Unsigned and Add (Refer to page 167.)
lh	Memory	Load Half Word (Refer to page 168.)
lh_u	Memory	Load Half Word Unsigned (Refer to page 169.)
lhadd	Memory	Load Half Word and Add (Refer to page 170.)
lhadd_u	Memory	Load Half Word Unsigned and Add (Refer to page 171.)
lnk	Control	Link (Refer to page 120.)
lw	Memory	Load Word (Refer to page 172.)
lw_na	Memory	Load Word No Alignment Trap (Refer to page 173.)
lwadd	Memory	Load Word and Add (Refer to page 174.)
lwadd_na	Memory	Load Word No Alignment Trap and Add (Refer to page 175.)
maxb_u	SIMD	Maximum Byte Unsigned (Refer to page 243.)
maxh	SIMD	Maximum Half Words (Refer to page 245.)
maxib_u	SIMD	Maximum Immediate Byte Unsigned (Refer to page 247.)
maxih	SIMD	Maximum Immediate Half Words (Refer to page 249.)
mf	Memory Maintenance	Memory Fence (Refer to page 188.)
mfspr	System	Move from Special Purpose Register Word (Refer to page 352.)
minb_u	SIMD	Minimum Byte Unsigned (Refer to page 251.)
minh	SIMD	Minimum Half Words (Refer to page 253.)

Table 4-3. Master List of Main Processor Instructions (continued)

Register	Type	Description
minib_u	SIMD	Minimum Immediate Byte Unsigned (Refer to page 255.)
minih	SIMD	Minimum Immediate Half Words (Refer to page 257.)
mm	Logical	Masked Merge Word (Refer to page 126.)
mnz	Logical	Mask Not Zero Word (Refer to page 128.)
mnzb	SIMD	Mask Not Zero Byte (Refer to page 259.)
mnzh	SIMD	Mask Not Zero Half Words (Refer to page 261.)
mtspr	System	Move to Special Purpose Register Word (Refer to page 353.)
mulhh_ss	Multiply	Multiply High Signed High Signed Half Word (Refer to page 191.)
mulhh_su	Multiply	Multiply High Signed High Unsigned Half Word (Refer to page 192.)
mulhh_uu	Multiply	Multiply High Unsigned High Unsigned Half Word (Refer to page 193.)
mulhha_ss	Multiply	Multiply Accumulate High Signed High Signed Half Word (Refer to page 194.)
mulhha_su	Multiply	Multiply Accumulate High Signed High Unsigned Half Word (Refer to page 195.)
mulhha_uu	Multiply	Multiply Accumulate High Unsigned High Unsigned Half Word (Refer to page 196.)
mulhhsa_uu	Multiply	Multiply Shift Accumulate High Unsigned High Unsigned Half Word (Refer to page 197.)
mulhl_ss	Multiply	Multiply High Signed Low Signed Half Word (Refer to page 198.)
mulhl_su	Multiply	Multiply High Signed Low Unsigned Half Word (Refer to page 199.)
mulhl_us	Multiply	Multiply High Unsigned Low Signed Half Word (Refer to page 200.)
mulhl_uu	Multiply	Multiply High Unsigned Low Unsigned Half Word (Refer to page 201.)
mulhla_ss	Multiply	Multiply Accumulate High Signed Low Signed Half Word (Refer to page 202.)
mulhla_su	Multiply	Multiply Accumulate High Signed Low Unsigned Half Word (Refer to page 203.)
mulhla_us	Multiply	Multiply Accumulate High Unsigned Low Signed Half Word (Refer to page 204.)
mulhla_uu	Multiply	Multiply Accumulate High Unsigned Low Unsigned Half Word (Refer to page 205.)
mulhlsa_uu	Multiply	Multiply Shift Accumulate High Unsigned Low Unsigned Half Word (Refer to page 206.)
mulll_ss	Multiply	Multiply Low Signed Low Signed Half Word (Refer to page 207.)
mulll_su	Multiply	Multiply Low Signed Low Unsigned Half Word (Refer to page 208.)

Table 4-3. Master List of Main Processor Instructions (continued)

Register	Type	Description
mulll_uu	Multiply	Multiply Low Unsigned Low Unsigned Half Word (Refer to page 209.)
mullla_ss	Multiply	Multiply Accumulate Low Signed Low Signed Half Word (Refer to page 210.)
mullla_su	Multiply	Multiply Accumulate Low Signed Low Unsigned Half Word (Refer to page 211.)
mullla_uu	Multiply	Multiply Accumulate Low Unsigned Low Unsigned Half Word (Refer to page 212.)
mulllsa_uu	Multiply	Multiply Shift Accumulate Low Unsigned Low Unsigned Half Word (Refer to page 212.)
mvnz	Logical	Move Not Zero Word (Refer to page 130.)
mvz	Logical	Move Zero Word (Refer to page 131.)
mz	Logical	Mask Zero Word (Refer to page 132.)
mzb	SIMD	Mask Zero Byte (Refer to page 263.)
mzh	SIMD	Mask Zero Half Words (Refer to page 265.)
nap	System	Nap (Refer to page 354.)
nop	NOP	Architectural No Operation (Refer to page 216.)
nor	Logical	Nor Word (Refer to page 134.)
or	Logical	Or Word (Refer to page 136.)
ori	Logical	Or Immediate Word (Refer to page 138.)
packhb	SIMD	Pack Low Byte (Refer to page 269.)
packhs	SIMD	Pack High Half Words Saturating (Refer to page 271.)
packlb	SIMD	Pack Low Byte (Refer to page 273.)
packbs_u	SIMD	Pack Half Words Saturating (Refer to page 267.)
pcnt	Bit Manipulation	Population Count Word (Refer to page 75.)
rl	Logical	Rotate Left Word (Refer to page 140.)
rli	Logical	Rotate Left Immediate Word (Refer to page 142.)
s1a	Arithmetic	Shift Left One Add Word (Refer to page 53.)
s2a	Arithmetic	Shift Left Two Add Word (Refer to page 55.)
s3a	Arithmetic	Shift Left Three Add Word (Refer to page 57.)
sadab_u	SIMD	Sum of Absolute Difference Accumulate Unsigned Bytes (Refer to page 267.)

Table 4-3. Master List of Main Processor Instructions (continued)

Register	Type	Description
sadah	SIMD	Sum of Absolute Difference Accumulate Half Words (Refer to page 276.)
sadah_u	SIMD	Sum of Absolute Difference Accumulate Unsigned Half Words (Refer to page 277.)
sadb_u	SIMD	Sum of Absolute Difference Unsigned Bytes (Refer to page 278.)
sadh	SIMD	Sum of Absolute Difference Half Words (Refer to page 279.)
sadh_u	SIMD	Sum of Absolute Difference Unsigned Half Words (Refer to page 280.)
sb	Memory	Store Byte (Refer to page 176.)
sbadd	Memory	Store Byte and Add (Refer to page 177.)
seq	Compare	Set Equal Word (Refer to page 77.)
seqb	SIMD	Set Equal to Byte (Refer to page 281.)
seqh	SIMD	Set Equal To Half Words (Refer to page 283.)
seqi	Compare	Set Equal Immediate Word (Refer to page 79.)
seqib	SIMD	Set Equal To Immediate Byte (Refer to page 285.)
seqih	SIMD	Set Equal To Immediate Half Words (Refer to page 287.)
sh	Memory	Store Half Word (Refer to page 178.)
shadd	Memory	Store Half Word and Add (Refer to page 179.)
shl	Logical	Logical Shift Left Word (Refer to page 144.)
shlb	SIMD	Logical Shift Left Bytes (Refer to page 289.)
shlh	SIMD	Logical Shift Left Half Words (Refer to page 291.)
shli	Logical	Logical Shift Left Immediate Word (Refer to page 146.)
shlib	SIMD	Logical Shift Left Immediate Bytes (Refer to page 292.)
shlih	SIMD	Logical Shift Left Immediate Half Words (Refer to page 294.)
shr	Logical	Logical Shift Right Word (Refer to page 148.)
shrb	SIMD	Logical Shift Right Bytes (Refer to page 296.)
shrh	SIMD	Logical Shift Right Half Words (Refer to page 298.)
shri	Logical	Logical Shift Right Immediate Word (Refer to page 150.)
shrib	SIMD	Logical Shift Right Immediate Bytes (Refer to page 300.)
shrih	SIMD	Logical Shift Right Immediate Half Words (Refer to page 302.)
slt	Compare	Set Less Than Word (Refer to page 81.)

Table 4-3. Master List of Main Processor Instructions (continued)

Register	Type	Description
slt_u	Compare	Set Less Than Unsigned Word (Refer to page 83.)
sltb	SIMD	Set Less Than Byte (Refer to page 304.)
sltb_u	SIMD	Set Less Than Unsigned Byte (Refer to page 306.)
slte	Compare	Set Less Than or Equal Word (Refer to page 85.)
slte_u	Compare	Set Less Than or Equal Unsigned Word (Refer to page 87.)
slteb	SIMD	Set Less Than or Equal Byte (Refer to page 308.)
slteb_u	SIMD	Set Less Than or Equal Unsigned Byte (Refer to page 310.)
slteh	SIMD	Set Less Than or Equal Half Words (Refer to page 312.)
slteh_u	SIMD	Set Less Than or Equal Unsigned Half Words (Refer to page 314.)
slth	SIMD	Set Less Than Half Words (Refer to page 316.)
slth_u	SIMD	Set Less Than Unsigned Half Words (Refer to page 318.)
slti	Compare	Set Less Than Immediate Word (Refer to page 89.)
slti_u	Compare	Set Less Than Unsigned Immediate Word (Refer to page 91.)
sltib	SIMD	Set Less Than Immediate Byte (Refer to page 320.)
sltib_u	SIMD	Set Less Than Unsigned Immediate Byte (Refer to page 322.)
sltih	SIMD	Set Less Than Immediate Half Words (Refer to page 324.)
sltih_u	SIMD	Set Less Than Unsigned Immediate Half Words (Refer to page 326.)
sne	Compare	Set Not Equal Word (Refer to page 93.)
sneb	SIMD	Set Not Equal To Byte (Refer to page 328.)
sneh	SIMD	Set Not Equal To Half Words (Refer to page 330.)
sra	Logical	Arithmetic Shift Right Word (Refer to page 152.)
srab	SIMD	Arithmetic Shift Right Bytes (Refer to page 332.)
srah	SIMD	Arithmetic Shift Right Half Words (Refer to page 334.)
srai	Logical	Arithmetic Shift Right Immediate Word (Refer to page 154.)
sraib	SIMD	Arithmetic Shift Right Immediate Bytes (Refer to page 336.)
sraih	SIMD	Arithmetic Shift Right Immediate Half Words (Refer to page 338.)
sub	Arithmetic	Subtract Word (Refer to page 59.)
subs	Arithmetic	Subtract Word Saturating (Refer to page 61.)
subb	SIMD	Subtract Bytes (Refer to page 340.)

Table 4-3. Master List of Main Processor Instructions (continued)

Register	Type	Description
subb_u	SIMD	Subtract Bytes Saturating Unsigned (Refer to page 342.)
subh	SIMD	Subtract Half Words (Refer to page 344.)
subhs	SIMD	Subtract Half Words Saturating (Refer to page 345.)
sw	Memory	Store Word (Refer to page 180.)
swadd	Memory	Store Word and Add (Refer to page 181.)
swint0	System	Software Interrupt 0 (Refer to page 355.)
swint1	System	Software Interrupt 1 (Refer to page 356.)
swint2	System	Software Interrupt 2 (Refer to page 357.)
swint3	System	Software Interrupt 3 (Refer to page 358.)
tblidxb0	Logical	Table Index Byte 0 (Refer to page 156.)
tblidxb1	Logical	Table Index Byte 1 (Refer to page 157.)
tblidxb2	Logical	Table Index Byte 2 (Refer to page 158.)
tblidxb3	Logical	Table Index Byte 3 (Refer to page 159.)
tns	Memory	Test and Set Word (Refer to page 182.)
wh64	Memory	Write Hint 64 Bytes (Refer to page 190.)
xor	Logical	Exclusive Or Word (Refer to page 160.)
xori	Logical	Exclusive Or Immediate Word (Refer to page 162.)

4.1.4 Arithmetic Instructions

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

- add: Add Word
- addi: Add Immediate Word
- addli: Add Long Immediate Word
- addlis: Add Long Immediate Static Write Word
- adds: Add Word Saturating
- auli: Add Upper Long Immediate Word
- s1a: Shift Left One Add Word
- s2a: Shift Left Two Add Word
- s3a: Shift Left Three Add Word
- sub: Subtract Word
- subs: Subtract Word Saturating

add: Add Word

Syntax

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

Example

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

Description

Adds two words together.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

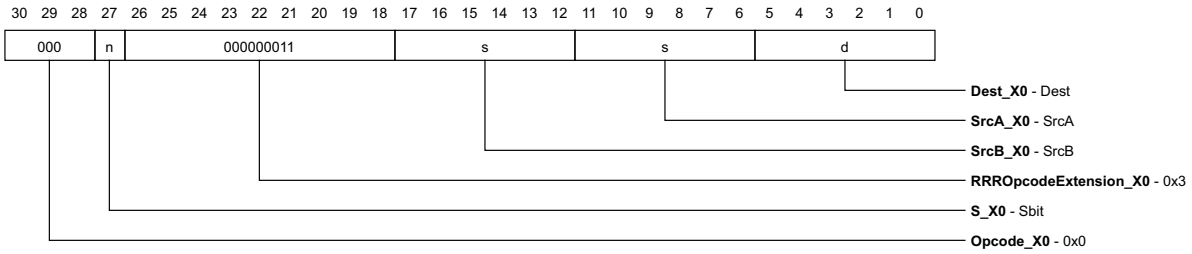

Figure 4-39: add in X0 Bit Descriptions

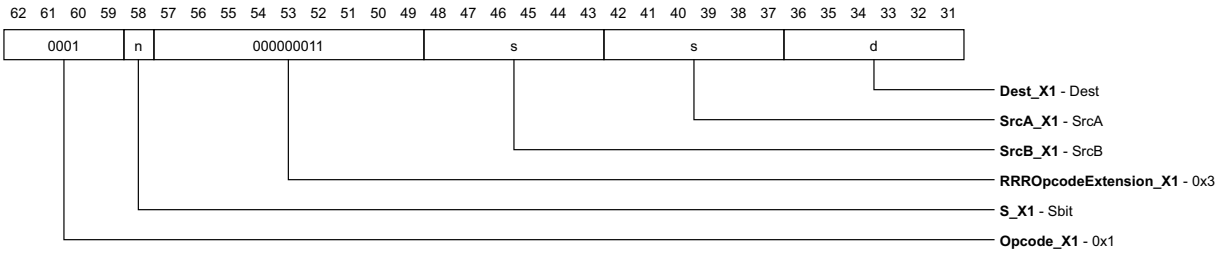

Figure 4-40: add in X1 Bit Descriptions

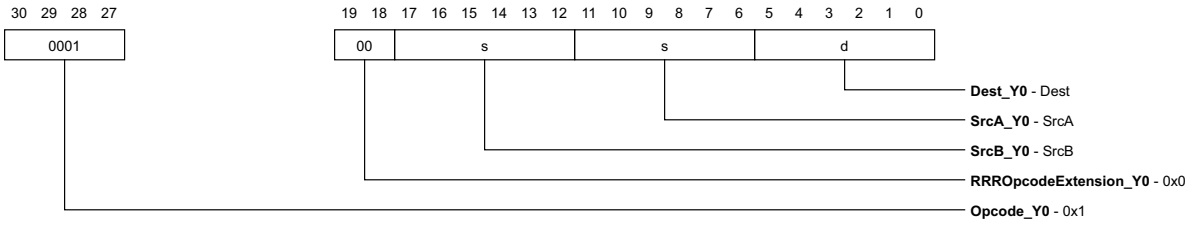

Figure 4-41: add in Y0 Bit Descriptions

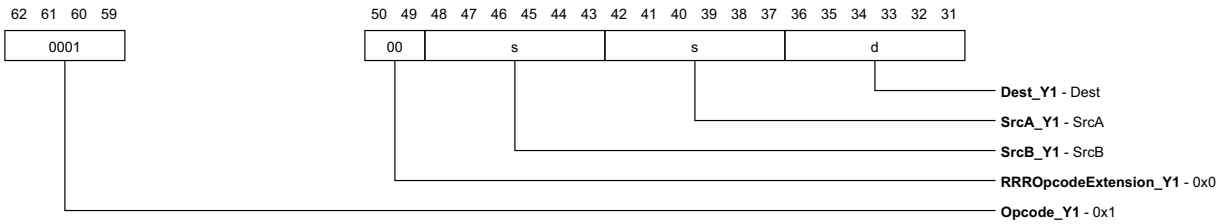

Figure 4-42: add in Y1 Bit Descriptions

addi: Add Immediate Word

Syntax

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

Example

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

Description

Adds one word 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

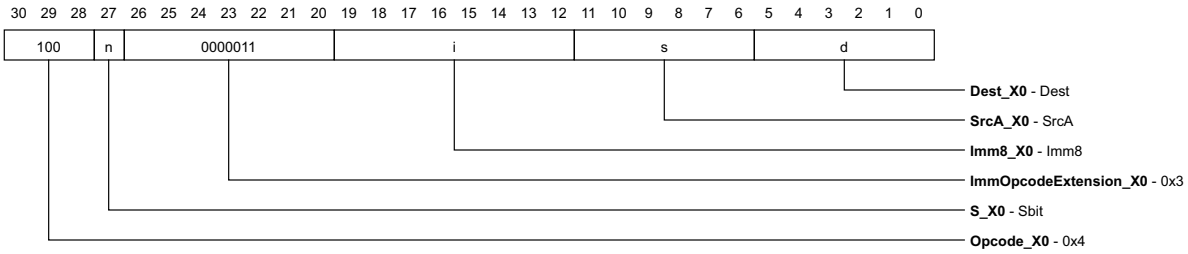

Figure 4-43: addi in X0 Bit Descriptions

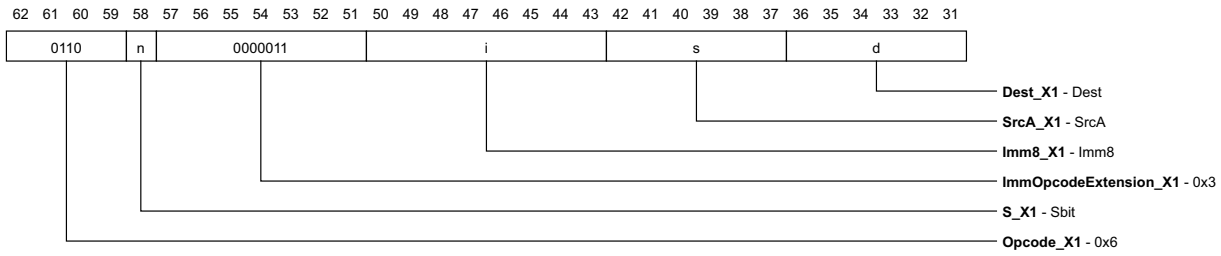

Figure 4-44: addi in X1 Bit Descriptions

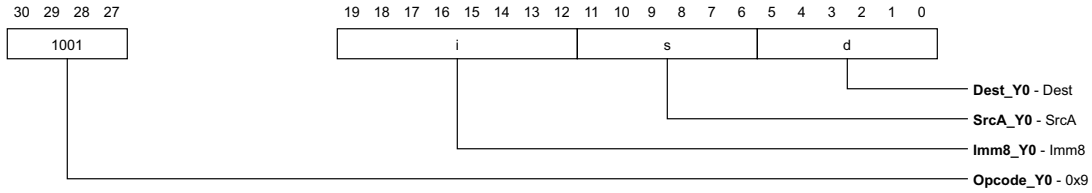

Figure 4-45: addi in Y0 Bit Descriptions

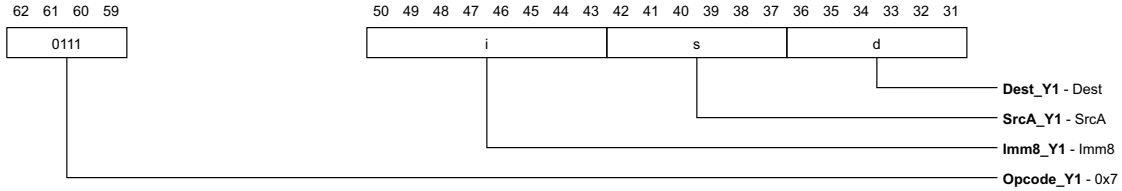

Figure 4-46: addi in Y1 Bit Descriptions

addli: Add Long Immediate Word

Syntax

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

Example

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

Description

Adds one word with a sign extended long immediate.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

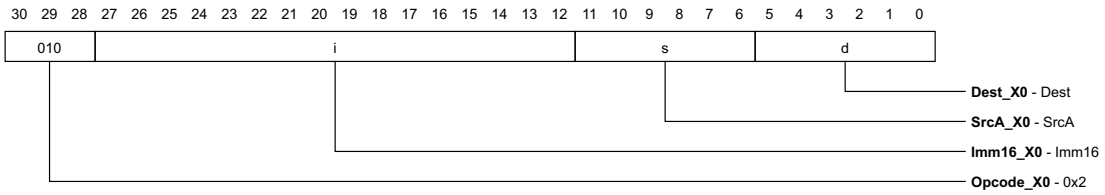

Figure 4-47: addli in X0 Bit Descriptions

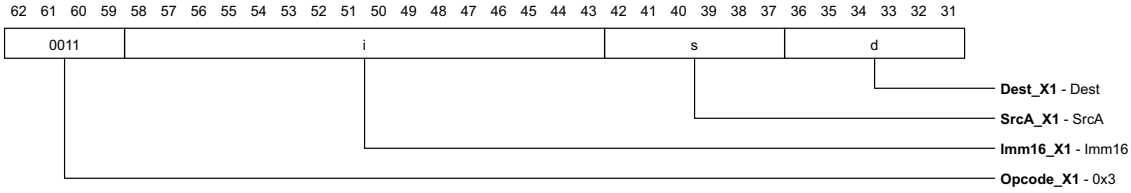

Figure 4-48: addli in X1 Bit Descriptions

addlis: Add Long Immediate Static Write Word

Syntax

```
addlis Dest, SrcA, Imm16
```

Example

```
addlis r5, r6, 0x1234
```

Description

Adds one word with a sign extended long immediate. The result is placed in the destination register and enqueued in the static network output port.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

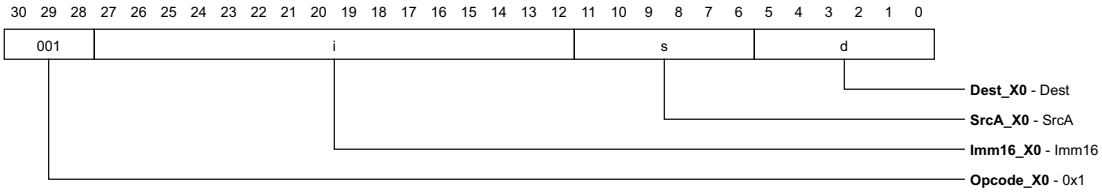

Figure 4-49: addlis in X0 Bit Descriptions

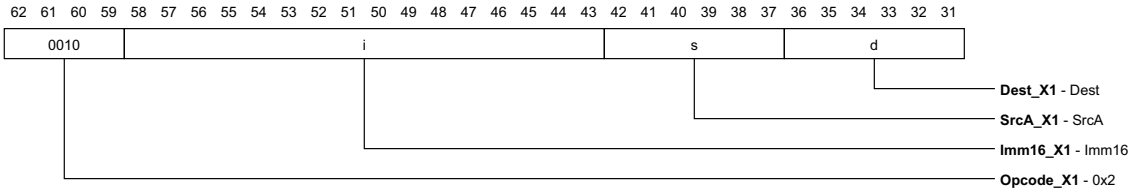

Figure 4-50: addlis in X1 Bit Descriptions

adds: Add Word Saturating

Syntax

```
adds Dest, SrcA, SrcB
```

Example

```
adds r5, r6, r7
```

Description

Adds two words together saturating the result at the minimum negative value or the maximum positive value.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

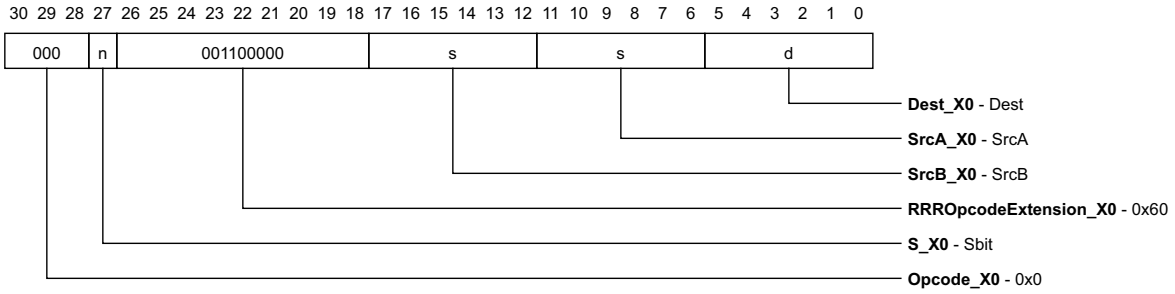

Figure 4-51: adds in X0 Bit Descriptions

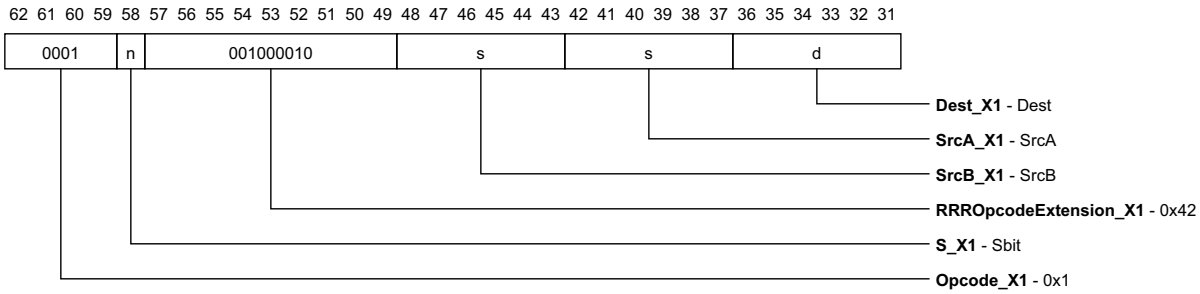

Figure 4-52: adds in X1 Bit Descriptions

auli: Add Upper Long Immediate Word

Syntax

```
auli Dest, SrcA, Imm16
```

Example

```
auli r5, r6, 0x1234
```

Description

Returns the addition of the first source operand and a sign extended long immediate loaded into the 16 most significant bits of a word. This instruction only contains an immediate form.

Functional Description

```
rf[Dest] = rf[SrcA] + (signExtend16( Imm16 ) << 16);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

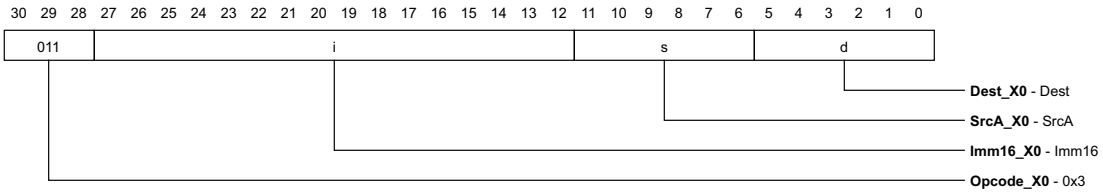

Figure 4-53: auli in X0 Bit Descriptions

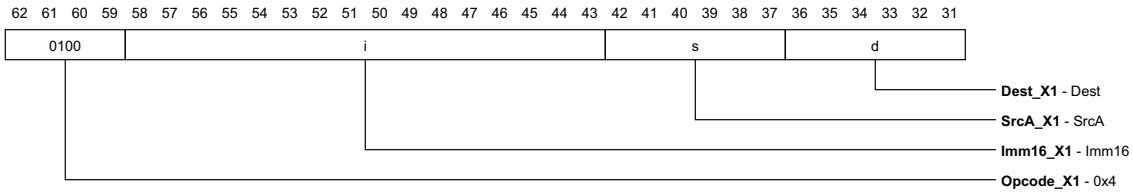

Figure 4-54: auli in X1 Bit Descriptions

s1a: Shift Left One Add Word

Syntax

```
s1a Dest, SrcA, SrcB
```

Example

```
s1a r5, r6, r7
```

Description

Shifts the first input 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

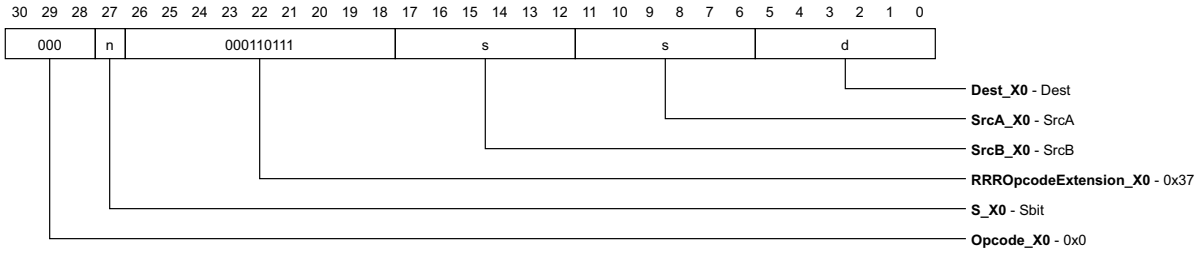

Figure 4-55: s1a in X0 Bit Descriptions

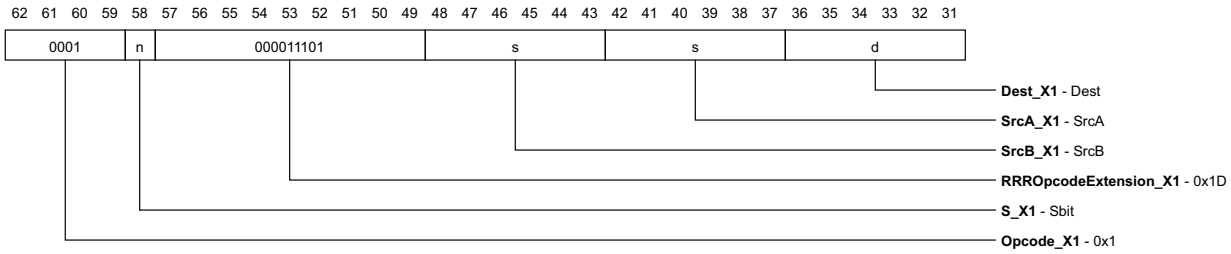

Figure 4-56: s1a in X1 Bit Descriptions

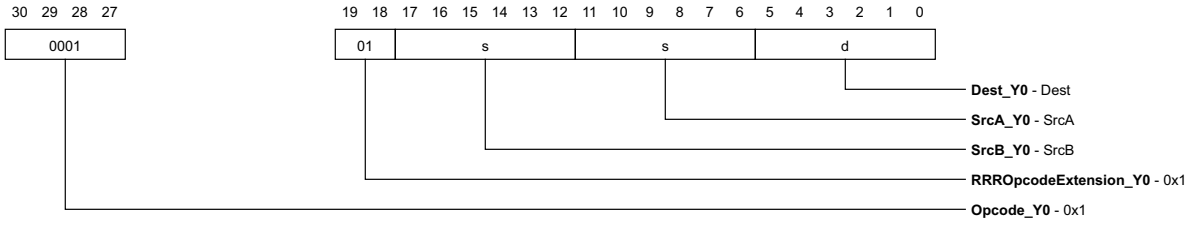

Figure 4-57: s1a in Y0 Bit Descriptions

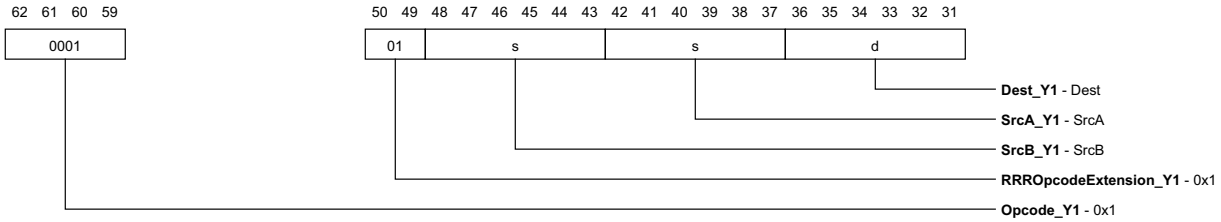

Figure 4-58: s1a in Y1 Bit Descriptions

s2a: Shift Left Two Add Word

Syntax

```
s2a Dest, SrcA, SrcB
```

Example

```
s2a r5, r6, r7
```

Description

Shifts the first input 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

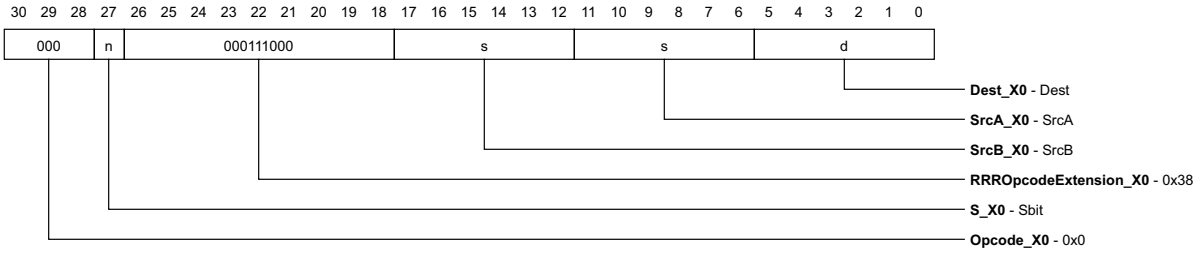

Figure 4-59: s2a in X0 Bit Descriptions

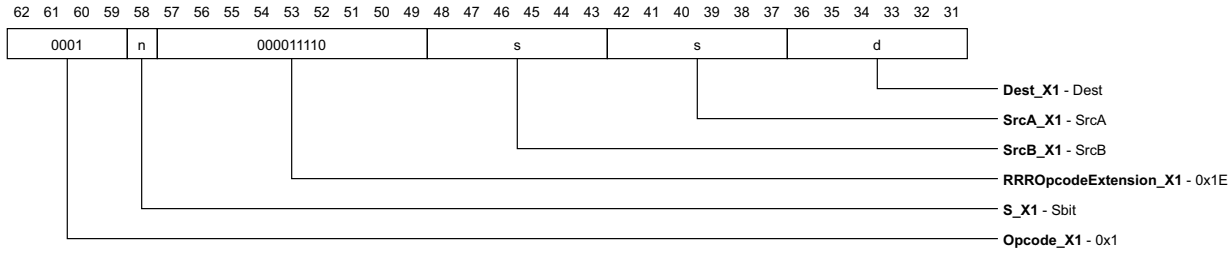

Figure 4-60: s2a in X1 Bit Descriptions

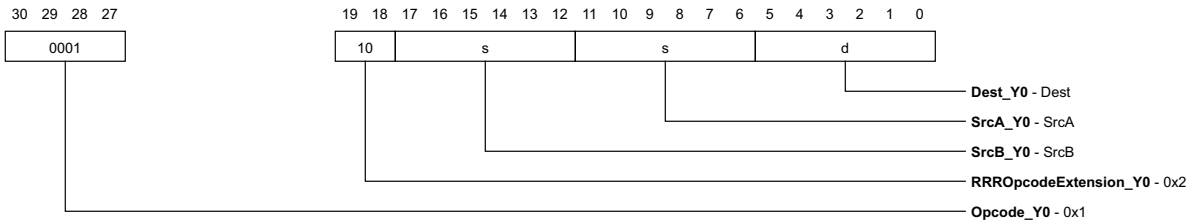

Figure 4-61: s2a in Y0 Bit Descriptions

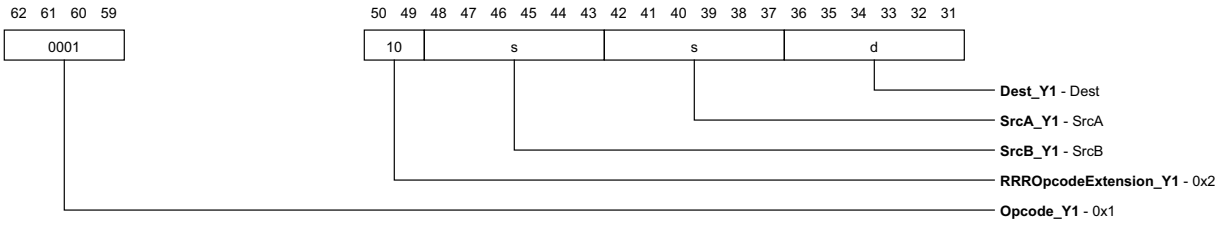

Figure 4-62: s2a in Y1 Bit Descriptions

s3a: Shift Left Three Add Word

Syntax

```
s3a Dest, SrcA, SrcB
```

Example

```
s3a r5, r6, r7
```

Description

Shifts the first input 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

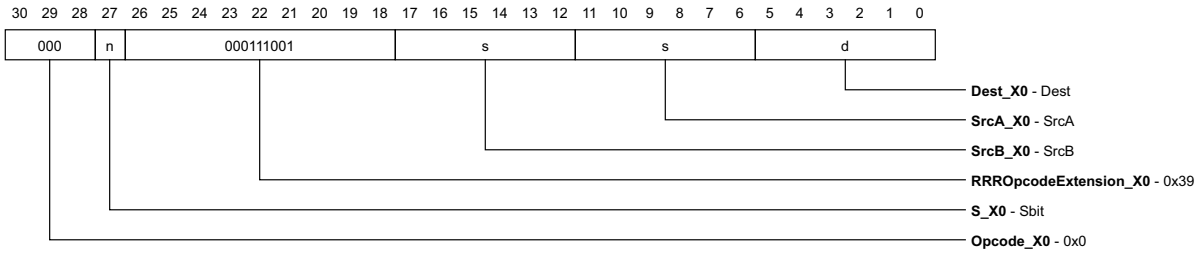

Figure 4-63: s3a in X0 Bit Descriptions

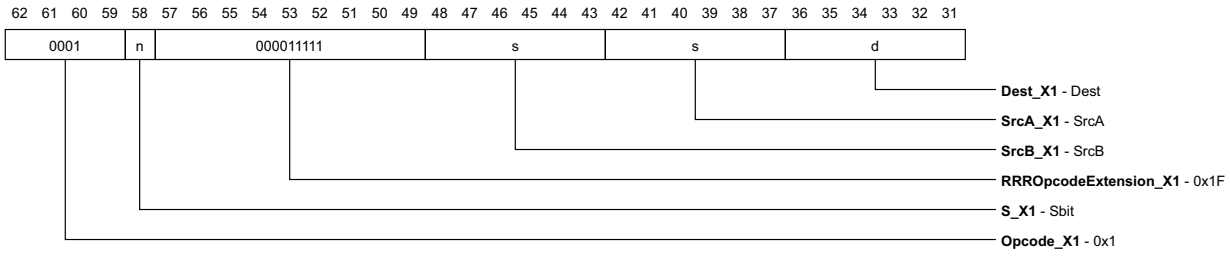

Figure 4-64: s3a in X1 Bit Descriptions

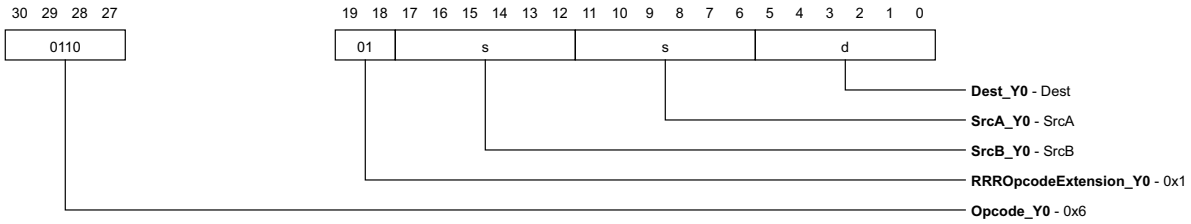

Figure 4-65: s3a in Y0 Bit Descriptions

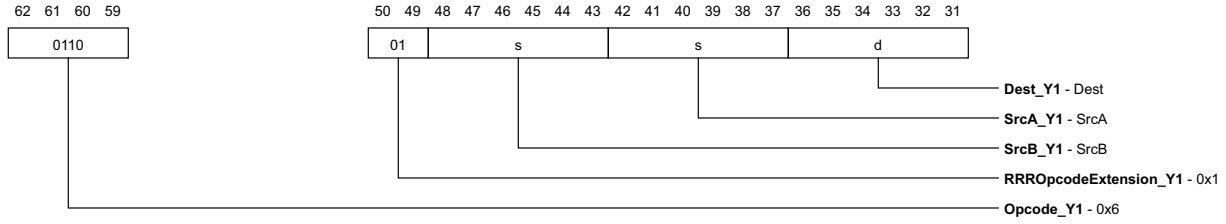

Figure 4-66: s3a in Y0 Bit Descriptions

sub: Subtract Word

Syntax

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

Example

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

Description

Subtracts one word from another.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

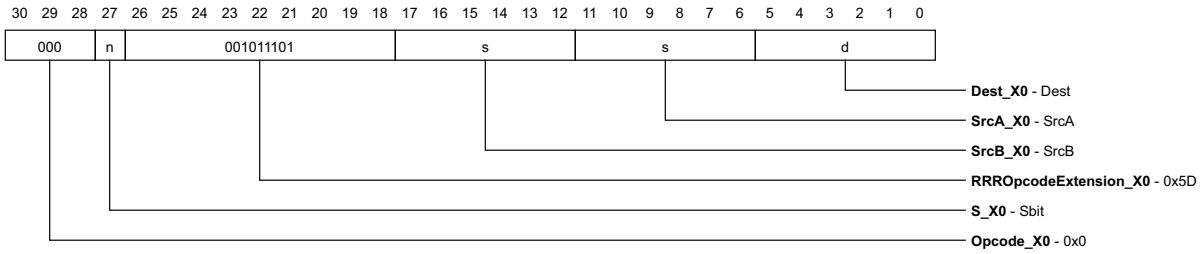

Figure 4-67: sub in X0 Bit Descriptions

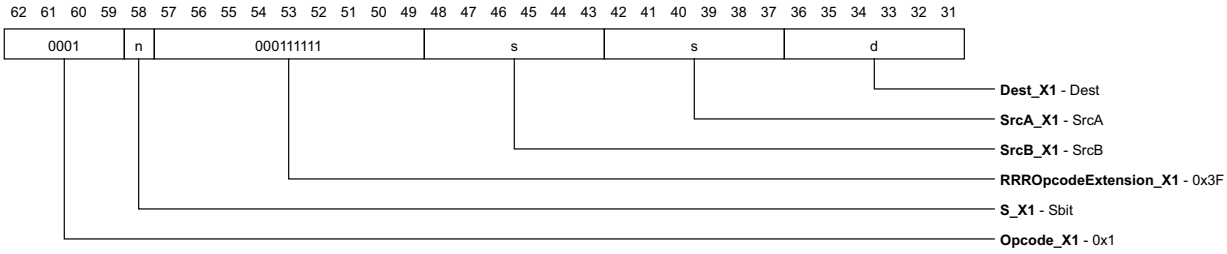

Figure 4-68: sub in X1 Bit Descriptions

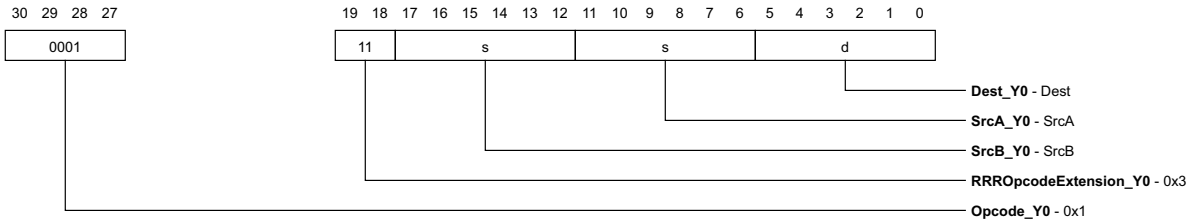

Figure 4-69: sub in Y0 Bit Descriptions

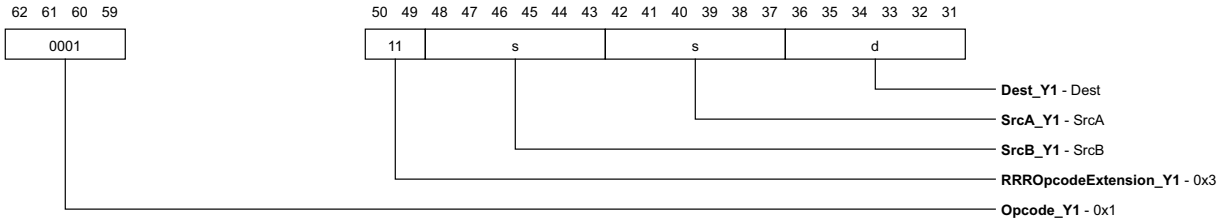

Figure 4-70: sub in Y0 Bit Descriptions

subs: Subtract Word Saturating

Syntax

```
subs Dest, SrcA, SrcB
```

Example

```
subs r5, r6, r7
```

Description

Subtracts one word from another, saturating the result at the minimum negative value or the maximum positive value.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

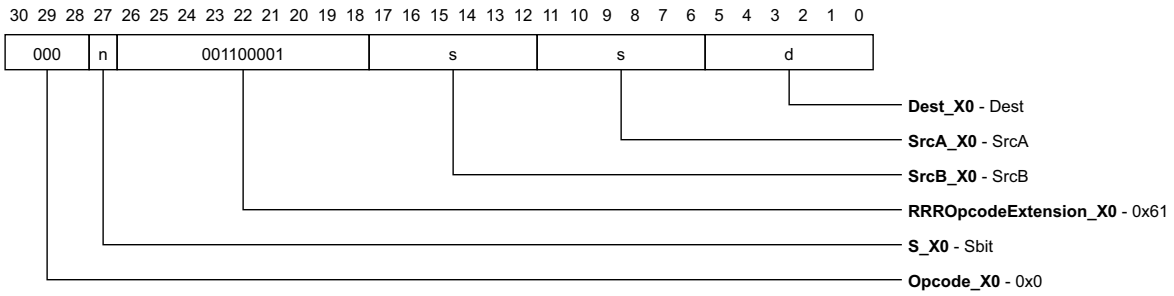

Figure 4-71: subs in X0 Bit Descriptions

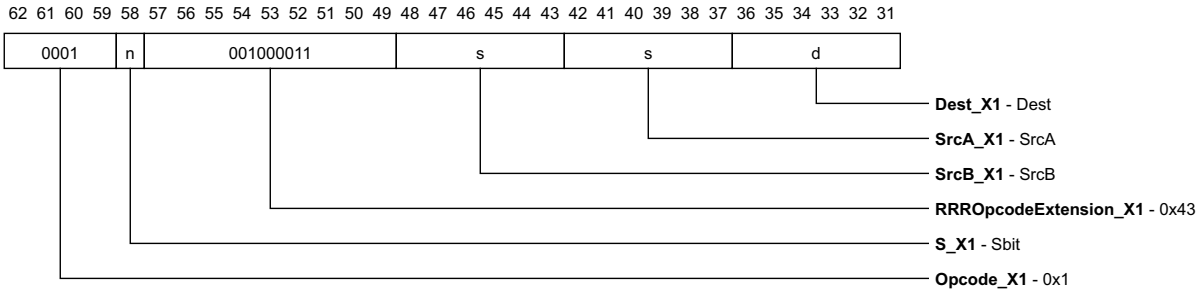

Figure 4-72: subs in X1 Bit Descriptions

4.1.5 Bit Manipulation Instructions

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

bitx: Bit Exchange Word

Syntax

```
bitx Dest, SrcA
```

Example

```
bitx 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 nth most significant bit becomes nth least significant bit in the output.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t 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

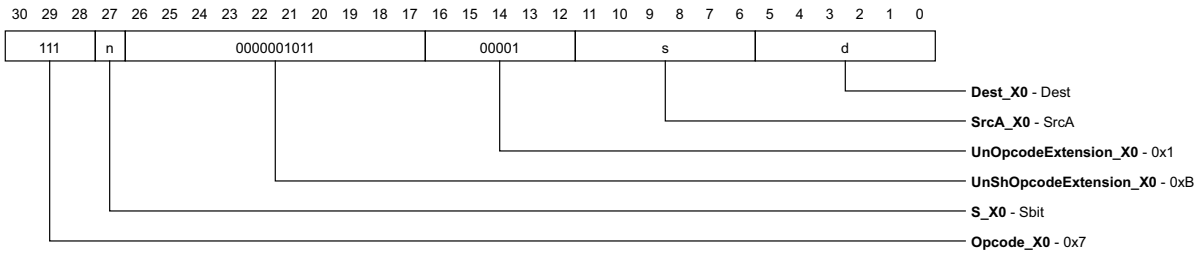

Figure 4-73: bitx in X0 Bit Descriptions

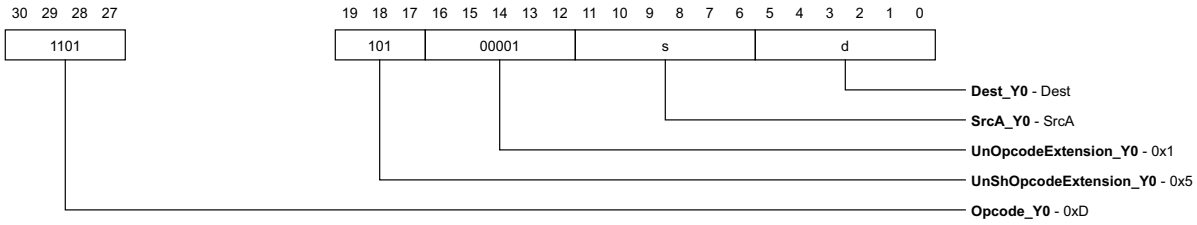

Figure 4-74: bitx in Y0 Bit Descriptions

bytex: Byte Exchange Word

Syntax

```
bytex Dest, SrcA
```

Example

```
bytex 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. This instruction changes endianness.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t 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

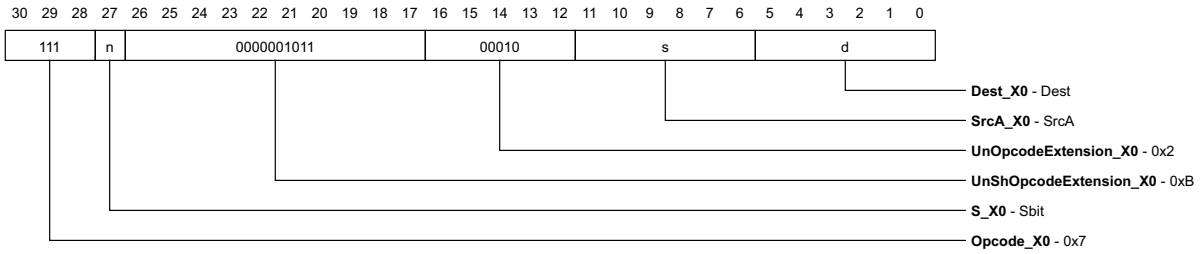

Figure 4-75: bytex in XO Bit Descriptions

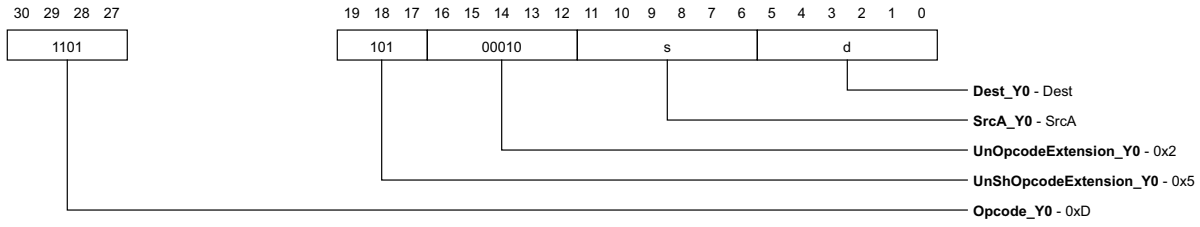

Figure 4-76: bytex in YO Bit Descriptions

clz: Count Leading Zeros Word

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

```
uint32_t 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

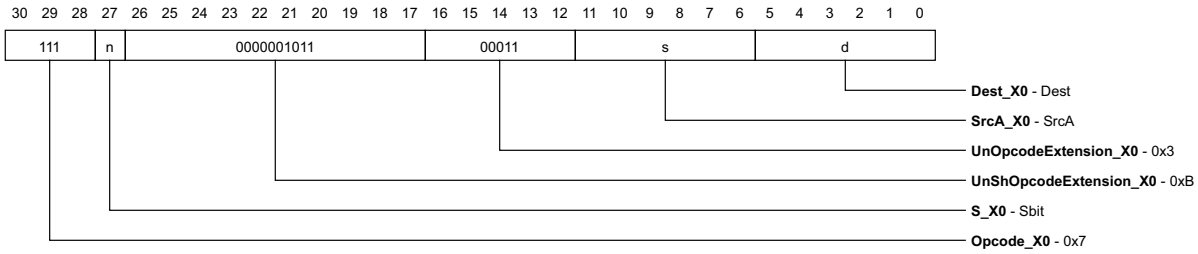

Figure 4-77: clz in XO Bit Descriptions

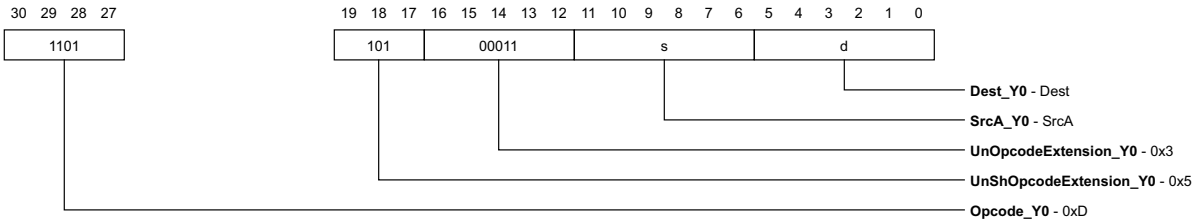

Figure 4-78: clz in YO Bit Descriptions

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 (uint32_t 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

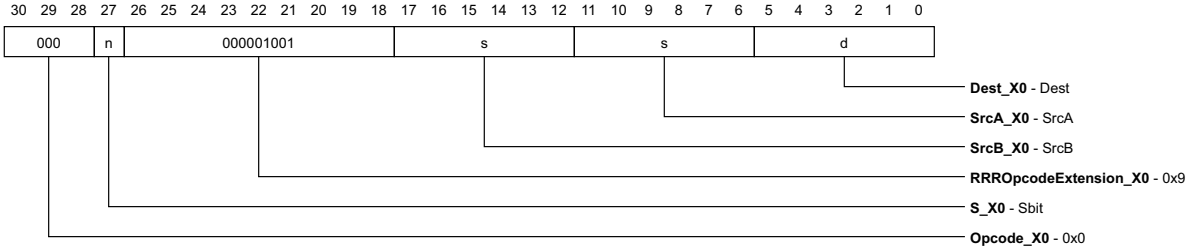

Figure 4-79: crc32_32 in XO Bit Descriptions

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 (uint32_t 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

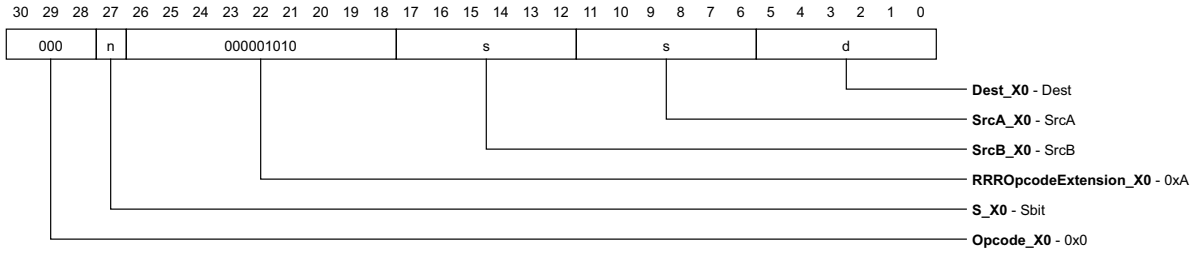

Figure 4-80: crc32_8 in XO Bit Descriptions

ctz: Count Trailing Zeros Word

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

```
uint32_t 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

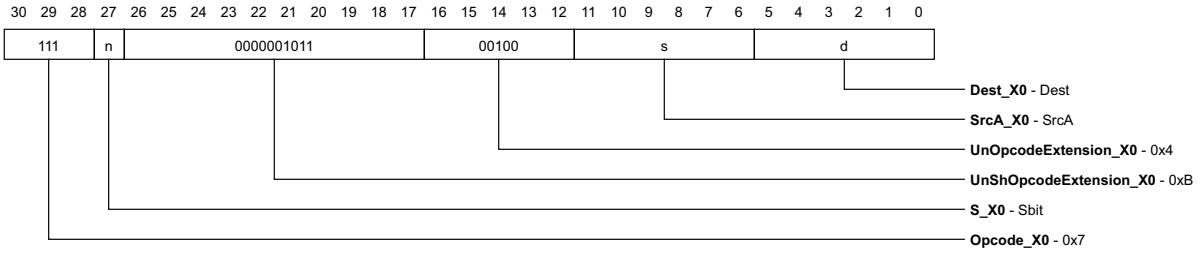

Figure 4-81: ctz in XO Bit Descriptions

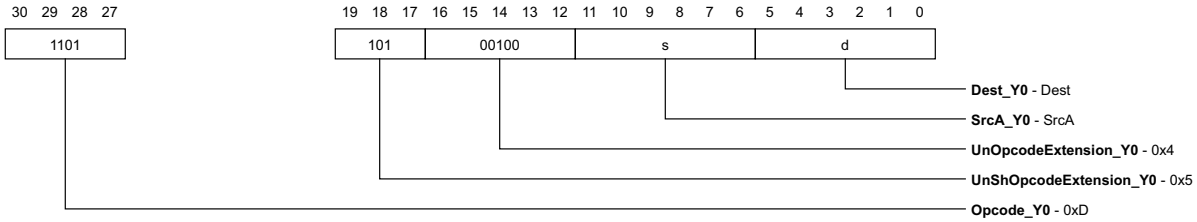

Figure 4-82: ctz in YO Bit Descriptions

dword_align: Double Word Align

Syntax

```
dword_align Dest, SrcA, SrcB
```

Example

```
dword_align r5, r6, r7
```

Description

Shift a double word by the number of bytes specified by the bottom two 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 source double word is constructed from the concatenation of the first source operand and the destination register.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

```
rf[Dest] =
    (UnsignedMachineWord) (little_endian()?
                           ((((UnsignedDoubleMachineWord)
                              ((UnsignedMachineWord) rf[SrcA]) <<
                              WORD_SIZE) |
                              (UnsignedDoubleMachineWord) ((UnsignedMachineWord) rf[Dest]))
                              >> (BYTE_SIZE * (rf[SrcB] & 3))) :
                           (((((UnsignedDoubleMachineWord) ((UnsignedMachineWord) rf[Dest])
                           << WORD_SIZE) |
                              (UnsignedDoubleMachineWord) ((UnsignedMachineWord) rf[SrcA]))
                           << (BYTE_SIZE * (rf[SrcB] & 3))) >> WORD_SIZE));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

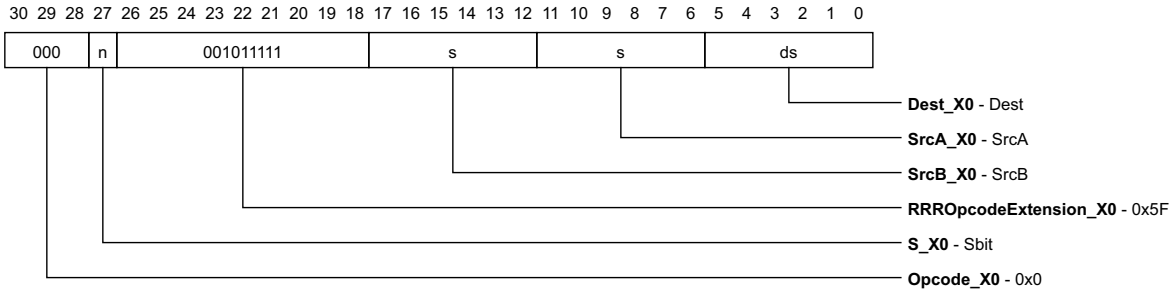

Figure 4-83: dword_align in X0 Bit Descriptions

pcnt: Population Count Word

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

```
uint32_t 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

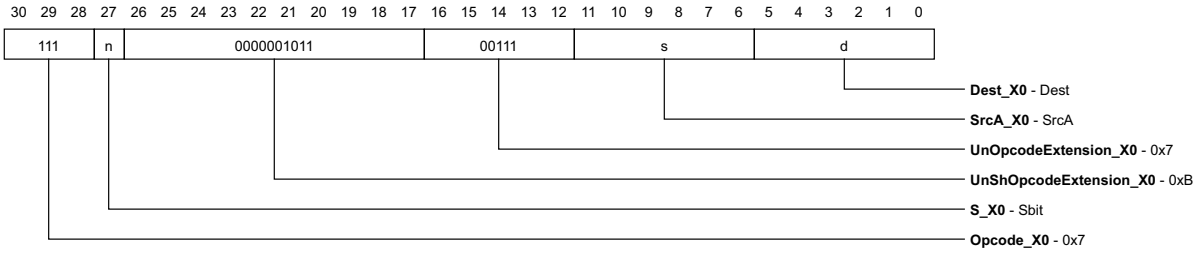

Figure 4-84: pcnt in XO Bit Descriptions

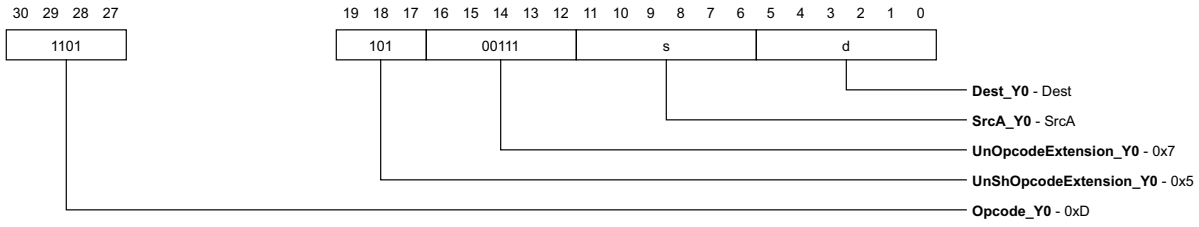

Figure 4-85: pcnt in YO Bit Descriptions

4.1.6 Compare Instructions

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

- seq: Set Equal Word
- seqi: Set Equal Immediate Word
- slt: Set Less Than Word
- slt_u: Set Less Than Unsigned Word
- slte: Set Less Than or Equal Word
- slte_u: Set Less Than or Equal Unsigned Word
- slti: Set Less Than Immediate Word
- slti_u: Set Less Than Unsigned Immediate Word
- sne: Set Not Equal Word

seq: Set Equal Word

Syntax

```
seq Dest, SrcA, SrcB
```

Example

```
seq r5, r6, r7
```

Description

Sets each 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] =
    ((UnsignedMachineWord) rf[SrcA] ==
     (UnsignedMachineWord) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

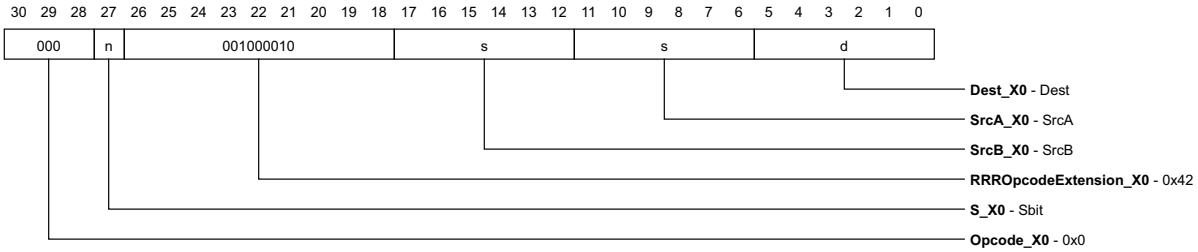

Figure 4-86: seq in XO Bit Descriptions

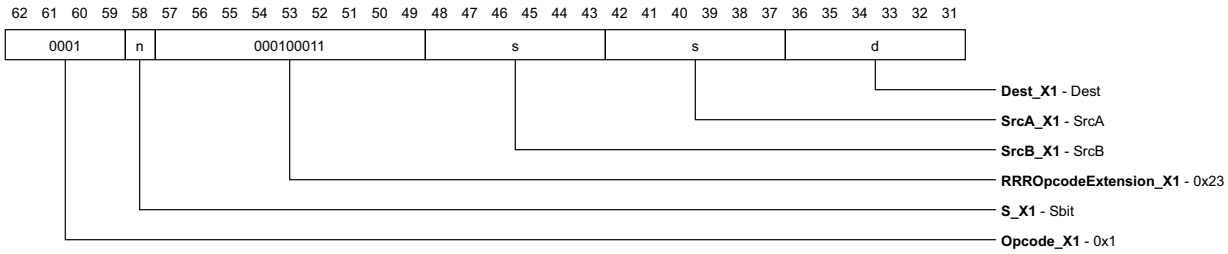

Figure 4-87: seq in X1 Bit Descriptions

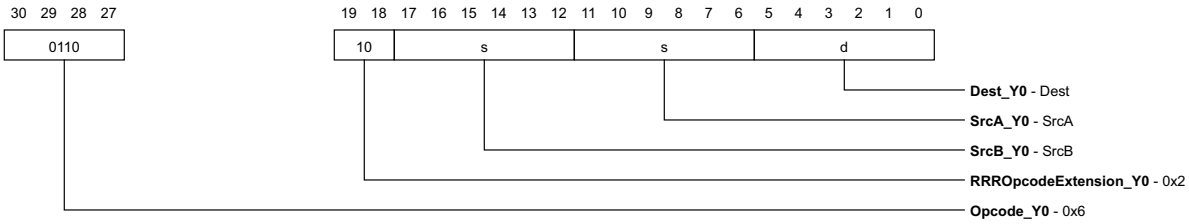

Figure 4-88: seq in YO Bit Descriptions

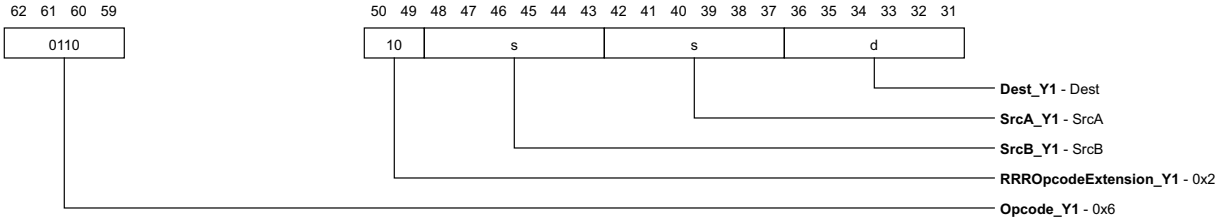

Figure 4-89: seq in Y1 Bit Descriptions

seqi: Set Equal Immediate Word

Syntax

```
seqi Dest, SrcA, Imm8
```

Example

```
seqi r5, r6, 5
```

Description

Sets each 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] =
    ((UnsignedMachineWord) rf[SrcA] ==
     (UnsignedMachineWord) signExtend8(Imm8)) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

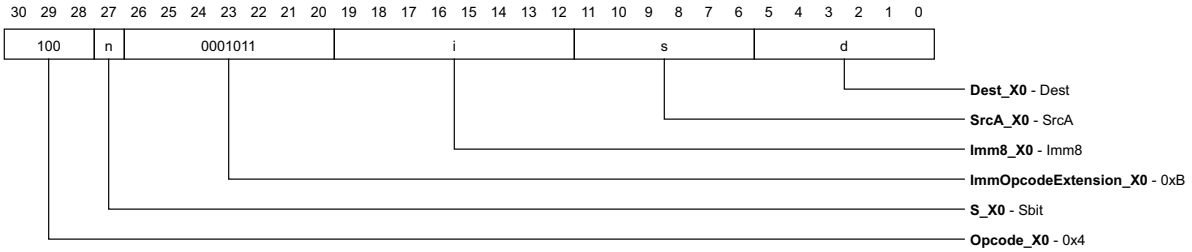

Figure 4-90: seqi in XO Bit Descriptions

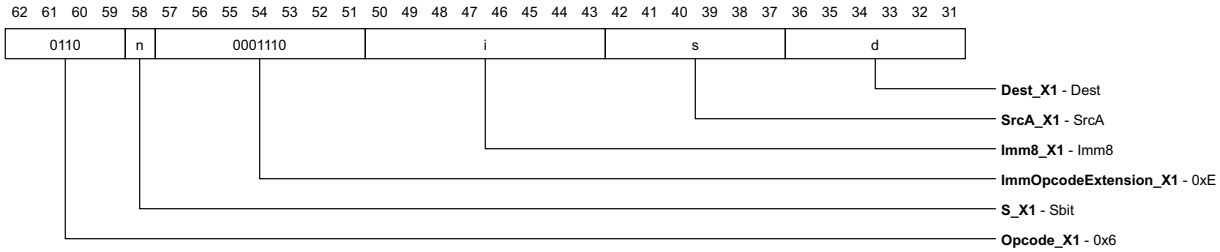

Figure 4-91: seqi in X1 Bit Descriptions

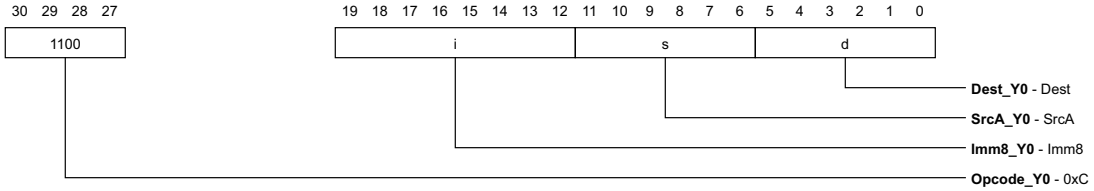

Figure 4-92: seqi in YO Bit Descriptions

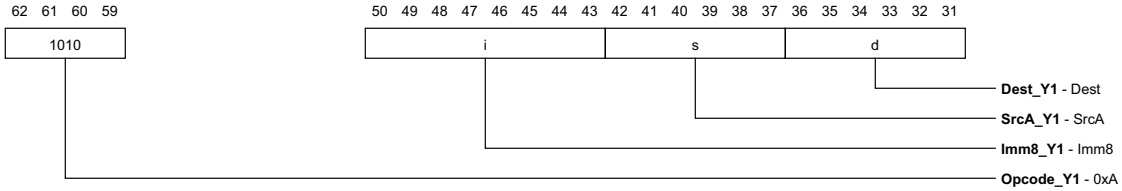

Figure 4-93: seqi in Y1 Bit Descriptions

slt: Set Less Than Word

Syntax

```
slt Dest, SrcA, SrcB
```

Example

```
slt r5, r6, r7
```

Description

Sets each 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] =
    ((SignedMachineWord) rf[SrcA] <
     (SignedMachineWord) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

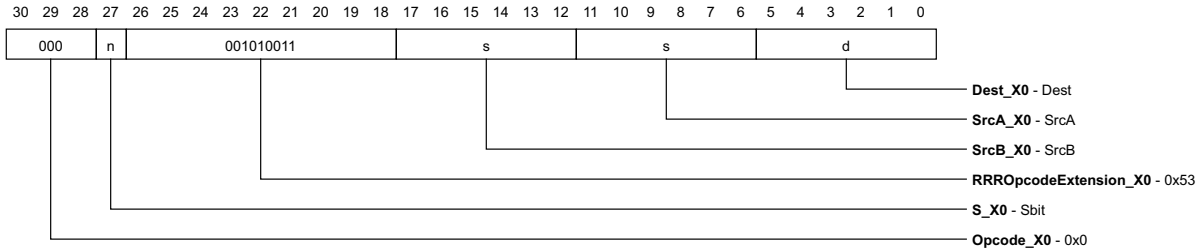

Figure 4-94: slt in XO Bit Descriptions

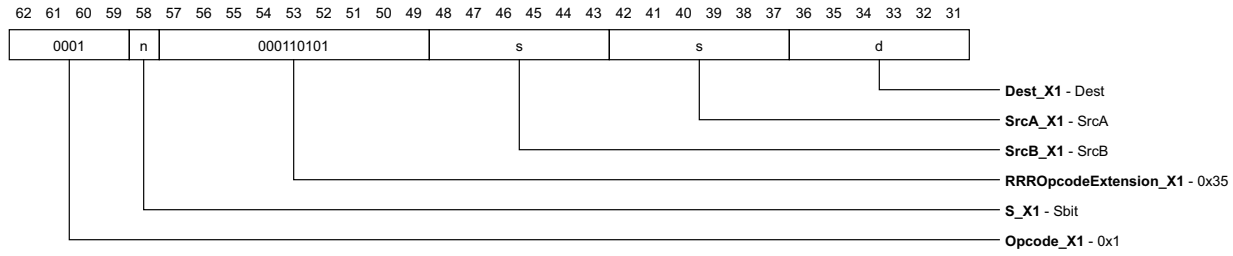

Figure 4-95: slt in X1 Bit Descriptions

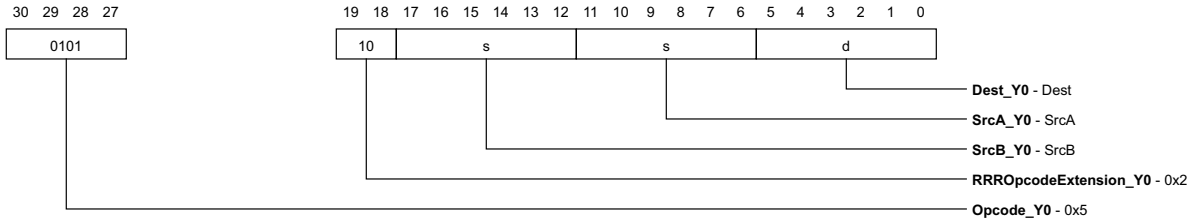

Figure 4-96: slt in YO Bit Descriptions

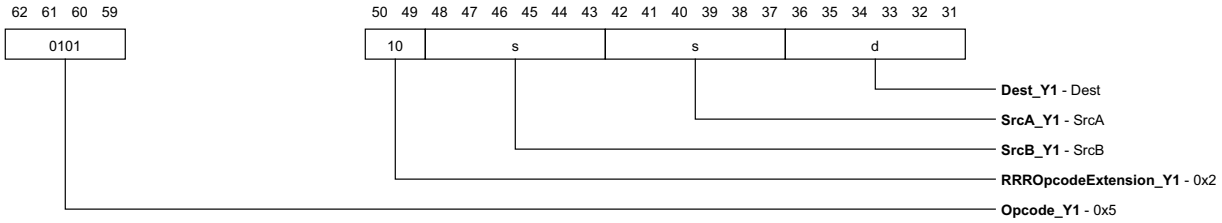

Figure 4-97: slt in Y1 Bit Descriptions

slt_u: Set Less Than Unsigned Word

Syntax

```
slt_u Dest, SrcA, SrcB
```

Example

```
slt_u r5, r6, r7
```

Description

Sets each 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] =
    ((UnsignedMachineWord) rf[SrcA] <
     (UnsignedMachineWord) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

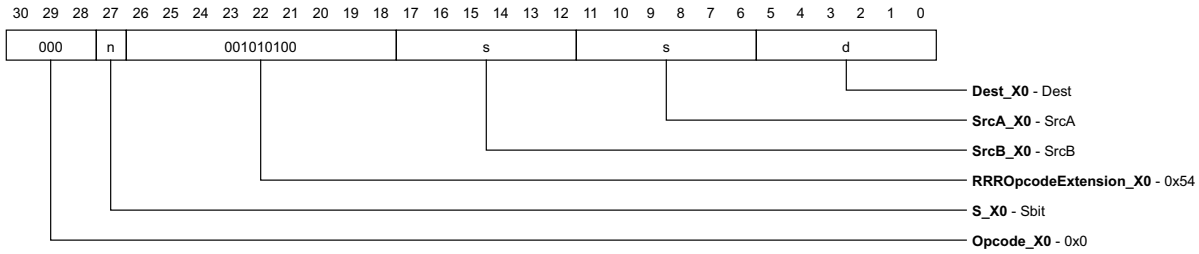

Figure 4-98: slt_u in XO Bit Descriptions

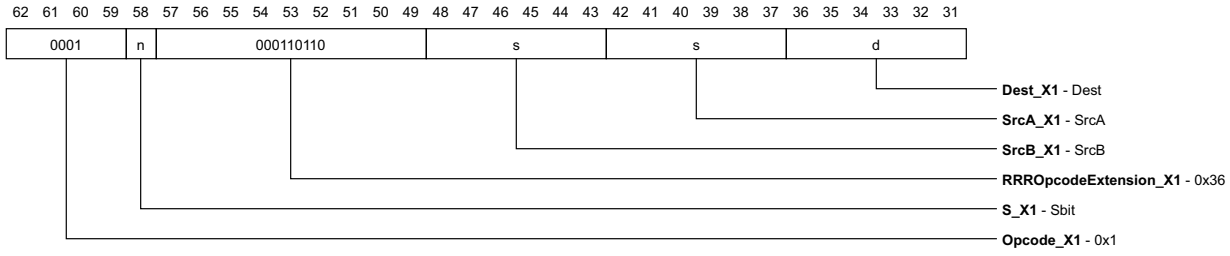

Figure 4-99: slt_u in X1 Bit Descriptions

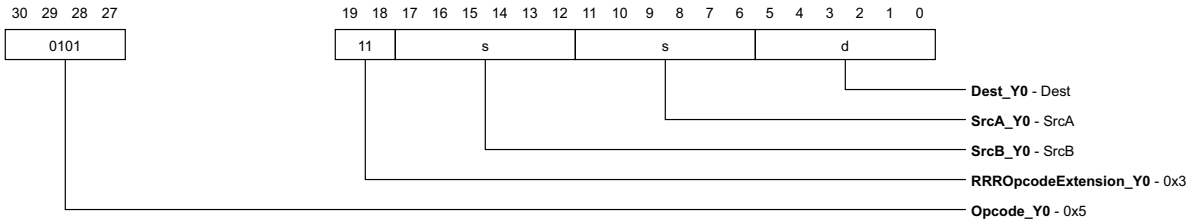

Figure 4-100: slt_u in YO Bit Descriptions

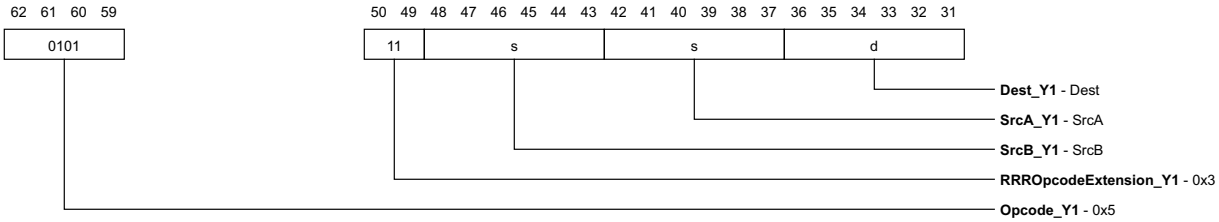

Figure 4-101: slt_u in Y1 Bit Descriptions

slte: Set Less Than or Equal Word

Syntax

```
slte Dest, SrcA, SrcB
```

Example

```
slte r5, r6, r7
```

Description

Sets each 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] =
    ((SignedMachineWord) rf[SrcA] <=
     (SignedMachineWord) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

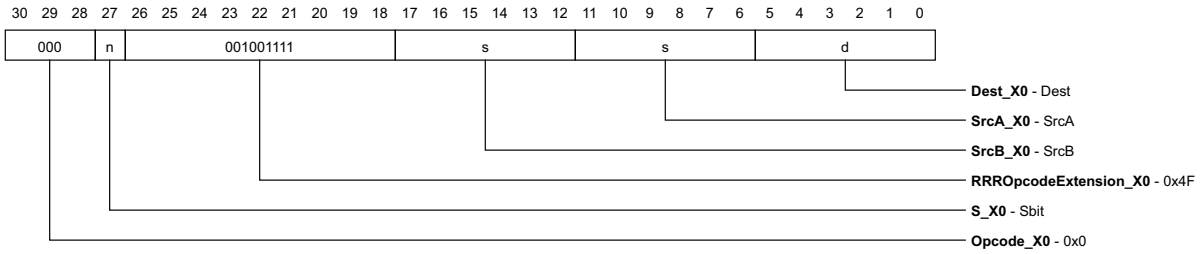

Figure 4-102: slte in XO Bit Descriptions

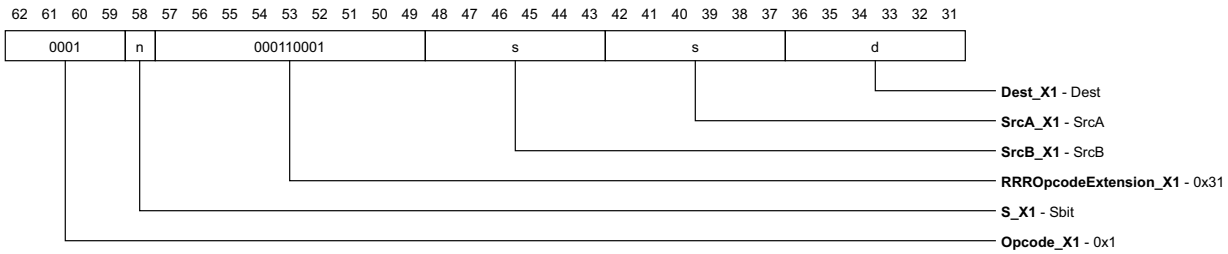

Figure 4-103: slte in X1 Bit Descriptions

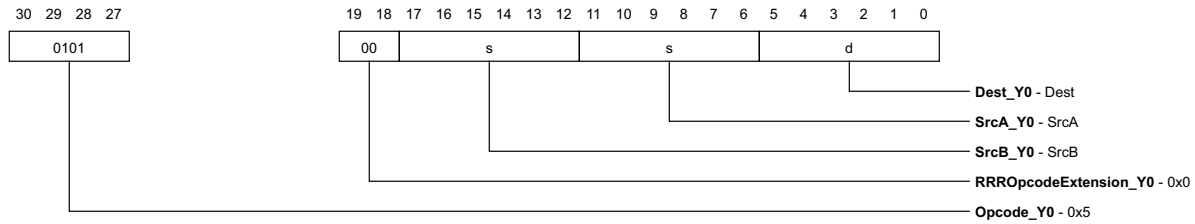

Figure 4-104: sIte in YO Bit Descriptions

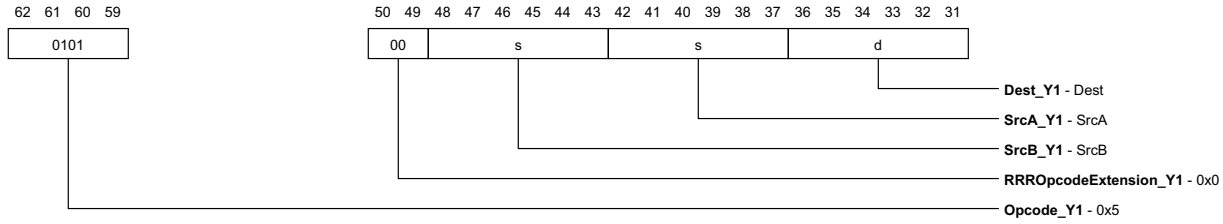

Figure 4-105: sIte in Y1 Bit Descriptions

slte_u: Set Less Than or Equal Unsigned Word

Syntax

```
slte_u Dest, SrcA, SrcB
```

Example

```
slte_u r5, r6, r7
```

Description

Sets each 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] =
    ((UnsignedMachineWord) rf[SrcA] <=
     (UnsignedMachineWord) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

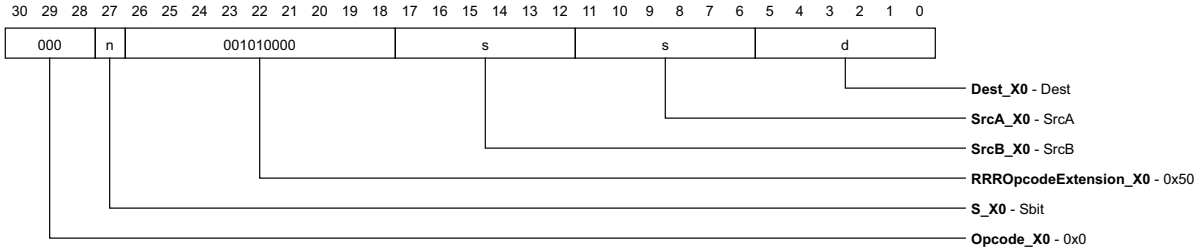

Figure 4-106: slte_u in XO Bit Descriptions

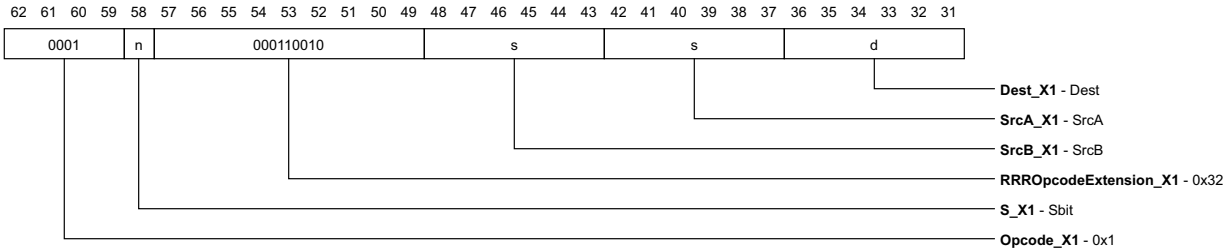

Figure 4-107: slte_u in X1 Bit Descriptions

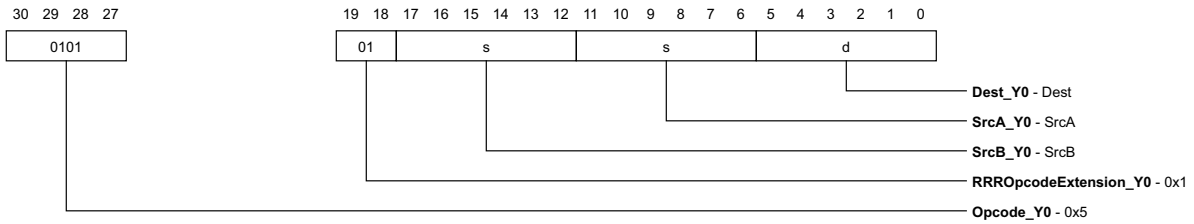

Figure 4-108: slte_u in YO Bit Descriptions

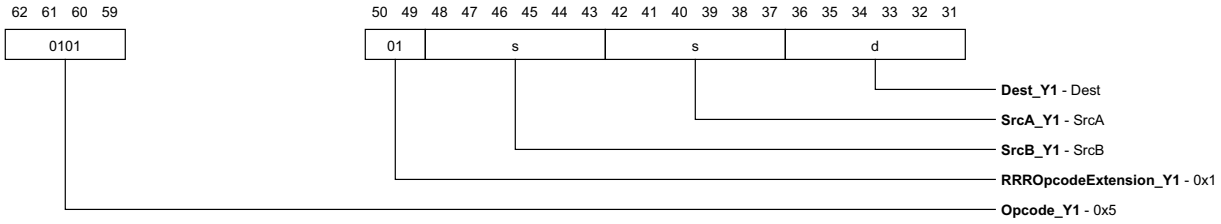

Figure 4-109: slte_u in Y1 Bit Descriptions

slti: Set Less Than Immediate Word

Syntax

```
slti Dest, SrcA, Imm8
```

Example

```
slti r5, r6, 5
```

Description

Sets each 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] =
    ((SignedMachineWord) rf[SrcA] <
     ((SignedMachineWord) signExtend8(Imm8))) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

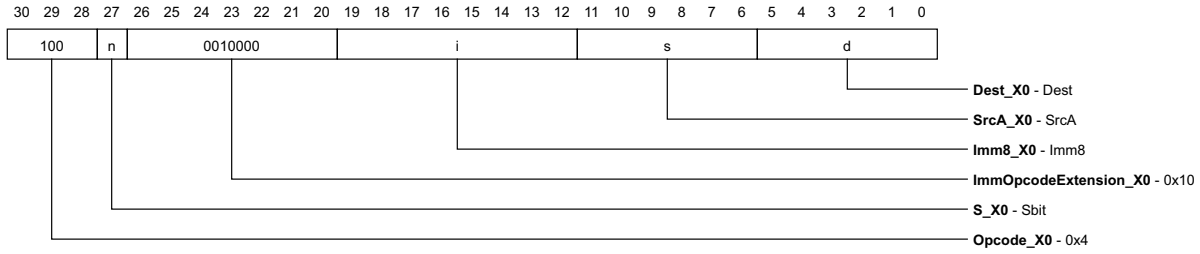

Figure 4-110: slti in XO Bit Descriptions

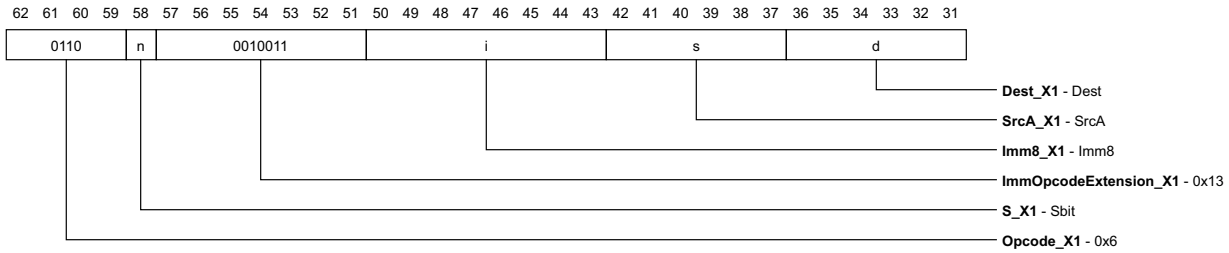

Figure 4-111: slti in X1 Bit Descriptions

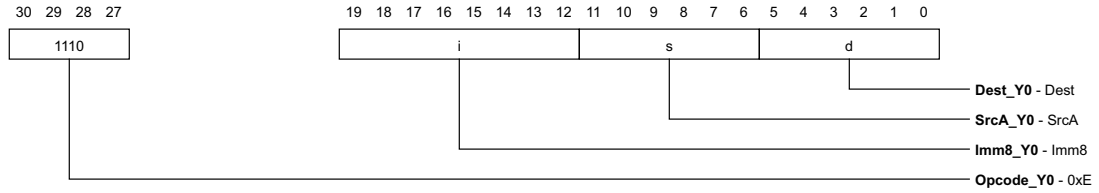

Figure 4-112: slti in YO Bit Descriptions

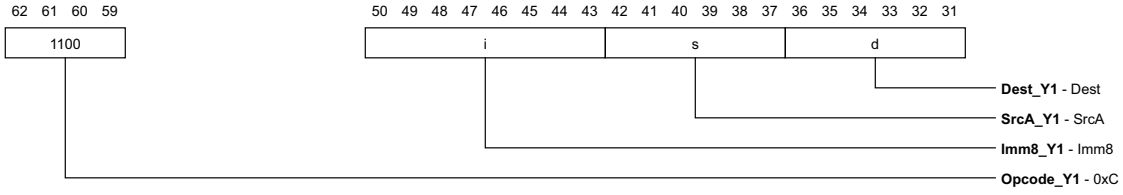

Figure 4-113: slti in Y1 Bit Descriptions

slti_u: Set Less Than Unsigned Immediate Word

Syntax

```
slti_u Dest, SrcA, Imm8
```

Example

```
slti_u r5, r6, 5
```

Description

Sets each 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] =
    ((UnsignedMachineWord) rf[SrcA] <
     ((UnsignedMachineWord) signExtend8(Imm8))) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

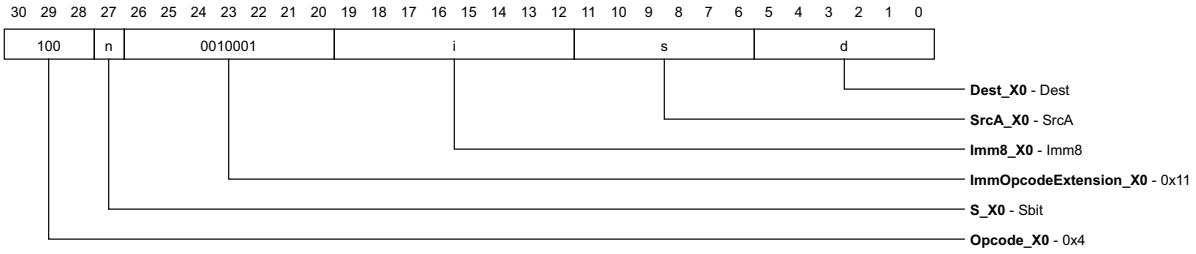

Figure 4-114: slti_u in XO Bit Descriptions

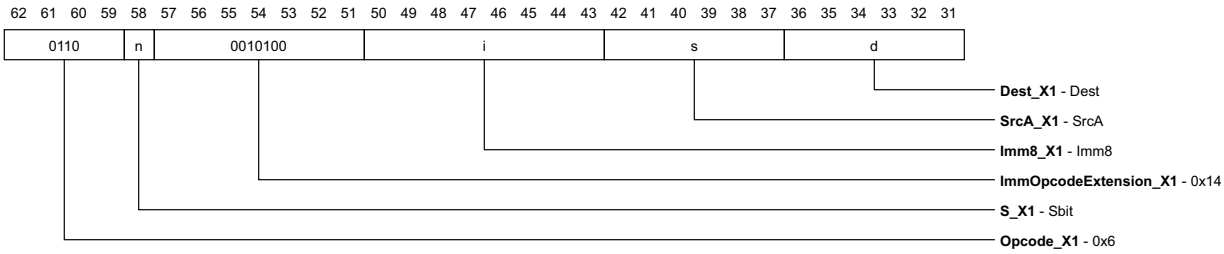

Figure 4-115: slti_u in X1 Bit Descriptions

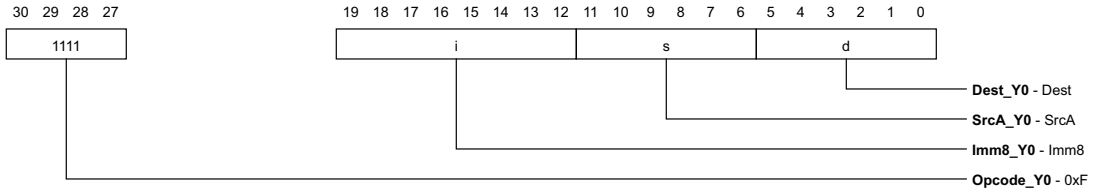

Figure 4-116: slti_u in YO Bit Descriptions

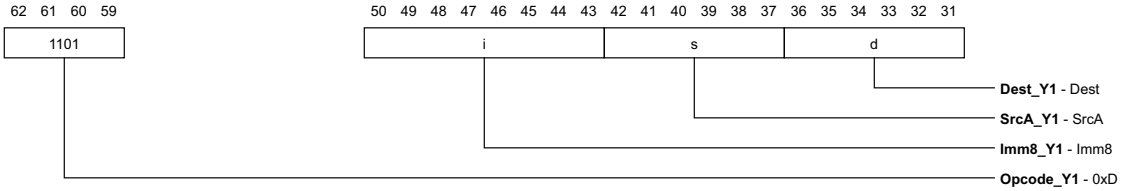

Figure 4-117: slti_u in Y1 Bit Descriptions

sne: Set Not Equal Word

Syntax

```
sne Dest, SrcA, SrcB
```

Example

```
sne r5, r6, r7
```

Description

Sets each 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] =
    ((UnsignedMachineWord) rf[SrcA] !=
     (UnsignedMachineWord) rf[SrcB]) ? 1 : 0;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

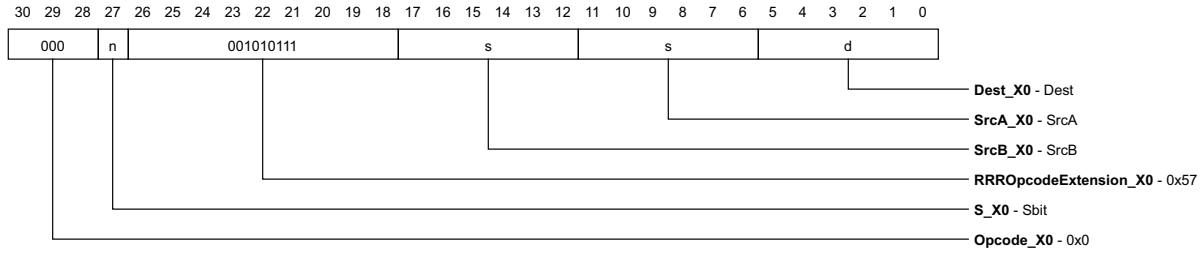

Figure 4-118: sne in XO Bit Descriptions

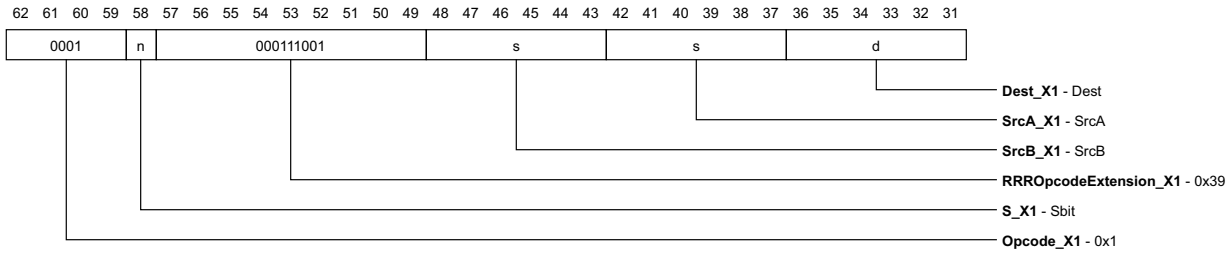

Figure 4-119: sne in X1 Bit Descriptions

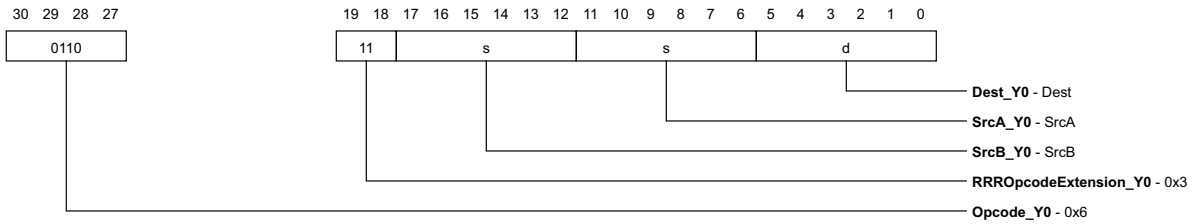

Figure 4-120: sne in YO Bit Descriptions

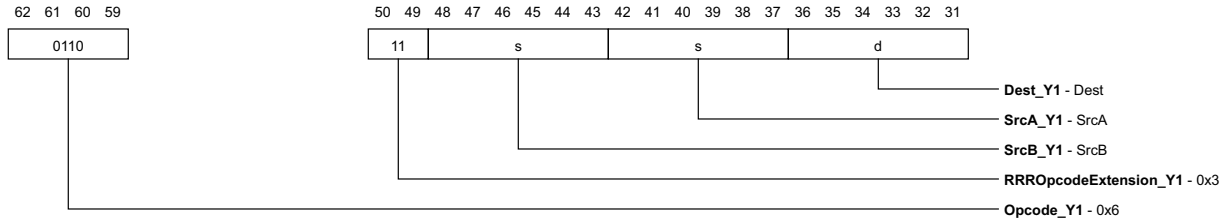

Figure 4-121: sne in Y1 Bit Descriptions

4.1.7 Control Instructions

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

- bbns: Branch Bit Not Set Word
- bbnst: Branch Bit Not Set Taken Word
- bbs: Branch Bit Set Word
- bbst: Branch Bit Set Taken Word
- bgez: Branch Greater Than or Equal to Zero Word
- bgezt: Branch Greater Than or Equal to Zero Predict Taken Word
- bgz: Branch Greater Than Zero Word
- bgzt: Branch Greater Than Zero Predict Taken Word
- blez: Branch Less Than or Equal to Zero Word
- blezt: Branch Less Than or Equal to Zero Taken Word
- blz: Branch Less Than Zero Word
- blzt: Branch Less Than Zero Taken Word
- bnz: Branch Not Zero Word
- bnzt: Branch Not Zero Predict Taken Word
- bz: Branch Zero Word
- bzt: Branch Zero Predict Taken Word
- jalb: Jump and Link Backward
- jalf: Jump and Link Forward
- jalr: Jump and Link Register
- jalrp: Jump and Link Register Predict
- jb: Jump Backward
- jf: Jump Forward
- jr: Jump Register
- jrp: Jump Register Predict
- lnk: Link

bbns: Branch Bit Not Set Word

Syntax

```
bbns SrcA, BrOff
```

Example

```
bbns r5, target
```

Description

Branches to the target if the source operand's bit 0 is not set (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

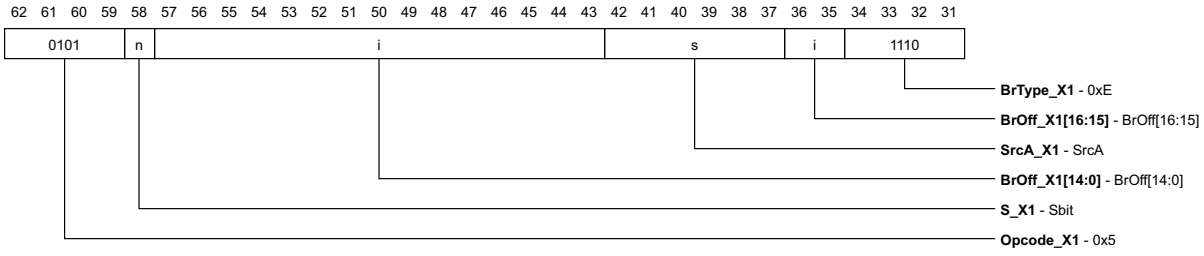

Figure 4-122: bbns in X1 Bit Descriptions

bbnst: Branch Bit Not Set Taken Word

Syntax

```
bbnst SrcA, BrOff
```

Example

```
bbnst r5, target
```

Description

Branches to the target if the source operand's bit 0 is not set (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

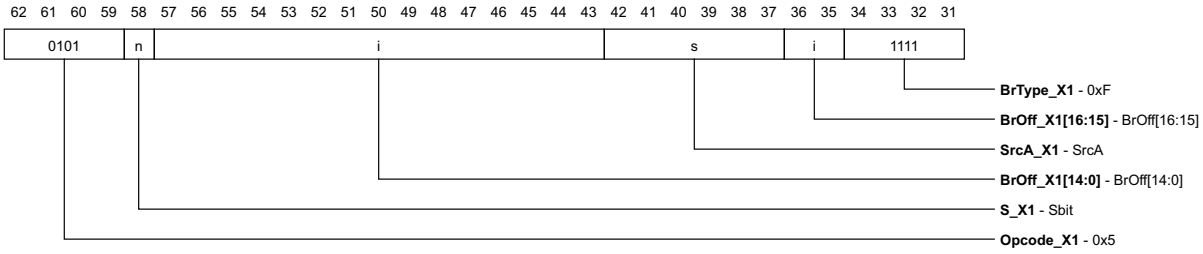

Figure 4-123: bbnst in X1 Bit Descriptions

bbs: Branch Bit Set Word

Syntax

```
bbs SrcA, BrOff
```

Example

```
bbs r5, target
```

Description

Branches to the target if the source operand's bit 0 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

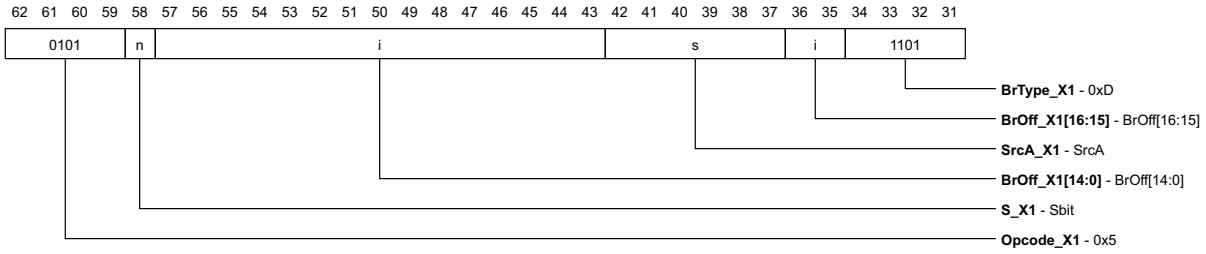

Figure 4-124: bbs in X1 Bit Descriptions

bbst: Branch Bit Set Taken Word

Syntax

```
bbst SrcA, BrOff
```

Example

```
bbst r5, target
```

Description

Branches to the target if the source operand's bit 0 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

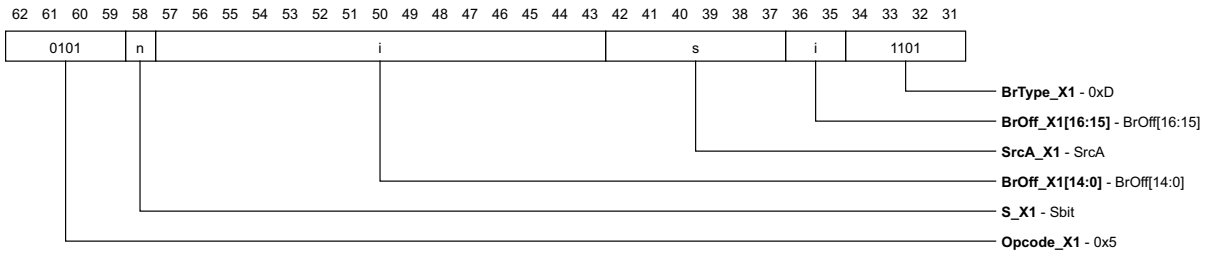

Figure 4-125: bbst in X1 Bit Descriptions

bgez: Branch Greater Than or Equal to Zero Word

Syntax

```
bgez SrcA, BrOff
```

Example

```
bgez r5, target
```

Description

Branches to the target if the source operand is greater than or equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch greater than or equal to 0 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

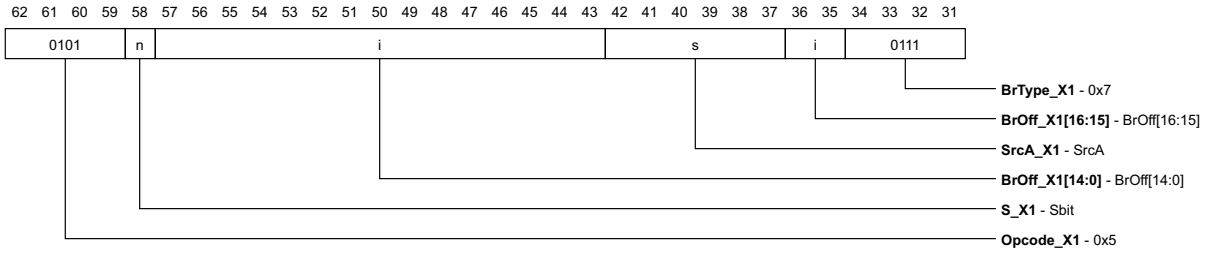

Figure 4-126: bgez in X1 Bit Descriptions

bgezt: Branch Greater Than or Equal to Zero Predict Taken Word

Syntax

```
bgezt SrcA, BrOff
```

Example

```
bgezt r5, target
```

Description

Branches to the target if the source operand is greater than or equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch greater than or equal to 0 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

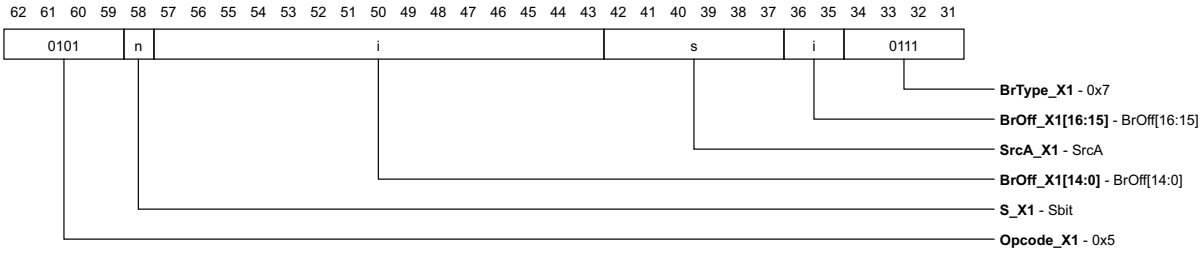

Figure 4-127: bgezt in X1 Bit Descriptions

bgz: Branch Greater Than Zero Word

Syntax

```
bgz SrcA, BrOff
```

Example

```
bgz r5, target
```

Description

Branches to the target if the source operand is greater than 0. Otherwise, the program counter advances to the next instruction in program order. Branch greater than 0 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

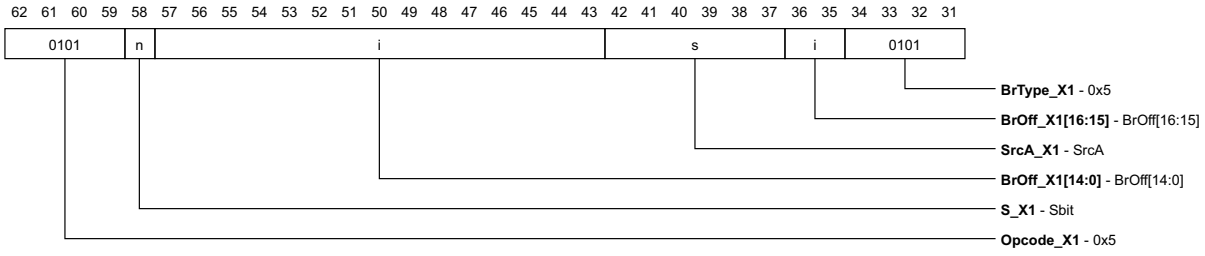

Figure 4-128: bgz in X1 Bit Descriptions

bgzt: Branch Greater Than Zero Predict Taken Word

Syntax

```
bgzt SrcA, BrOff
```

Example

```
bgzt r5, target
```

Description

Branches to the target if the source operand is greater than 0. Otherwise, the program counter advances to the next instruction in program order. Branch greater than 0 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

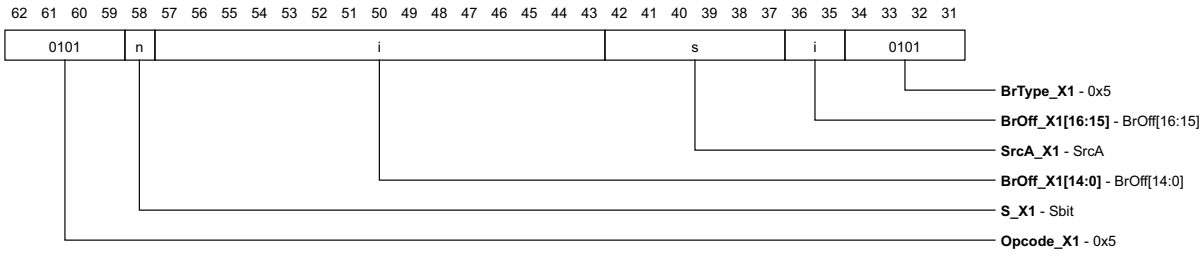

Figure 4-129: bgzt in X1 Bit Descriptions

blez: Branch Less Than or Equal to Zero Word

Syntax

```
blez SrcA, BrOff
```

Example

```
blez r5, target
```

Description

Branches to the target if the source operand is less than or equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch less than or equal to 0 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

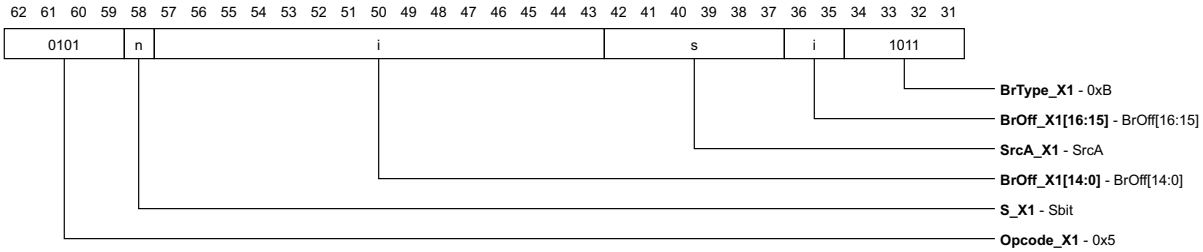

Figure 4-130: blez in X1 Bit Descriptions

blezt: Branch Less Than or Equal to Zero Taken Word

Syntax

```
blezt SrcA, BrOff
```

Example

```
blezt r5, target
```

Description

Branches to the target if the source operand is less than or equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch less than or equal to 0 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

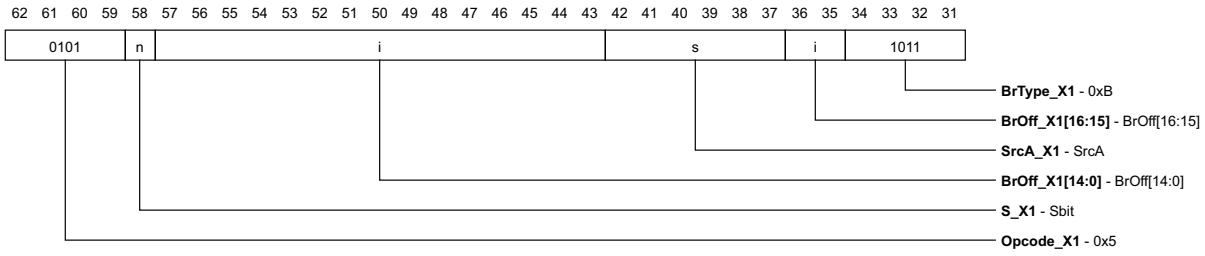

Figure 4-131: blezt in X1 Bit Descriptions

blz: Branch Less Than Zero Word

Syntax

```
blz SrcA, BrOff
```

Example

```
blz r5, target
```

Description

Branches to the target if the source operand is less than 0. Otherwise, the program counter advances to the next instruction in program order. Branch less than 0 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

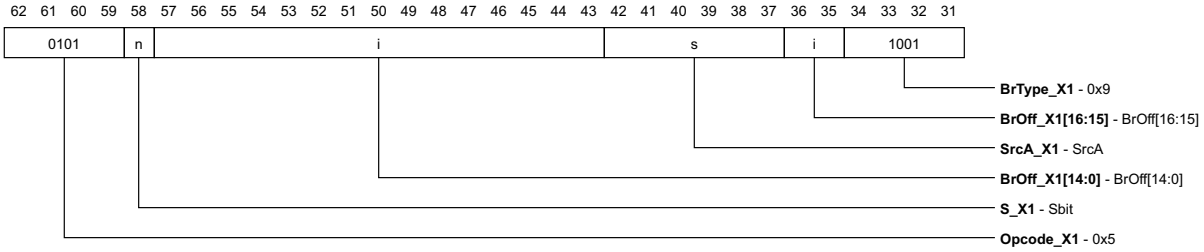

Figure 4-132: blz in X1 Bit Descriptions

blzt: Branch Less Than Zero Taken Word

Syntax

```
blzt SrcA, BrOff
```

Example

```
blzt r5, target
```

Description

Branches to the target if the source operand is less than 0. Otherwise, the program counter advances to the next instruction in program order. Branch less than 0 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

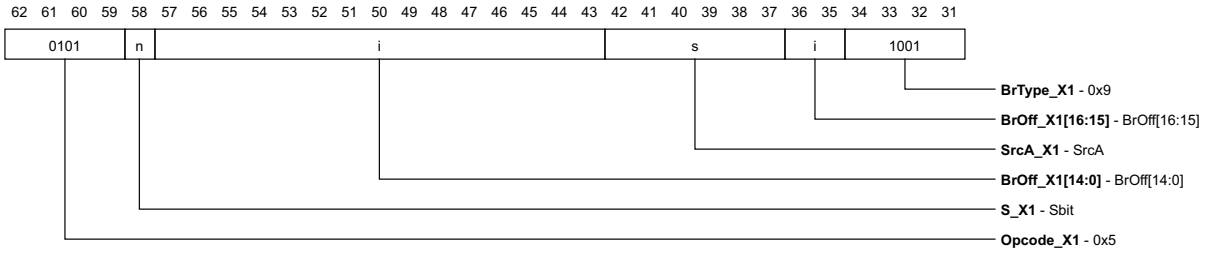

Figure 4-133: blzt in X1 Bit Descriptions

bnz: Branch Not Zero Word

Syntax

```
bnz SrcA, BrOff
```

Example

```
bnz r5, target
```

Description

Branches to the target if the source operand is not equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch not 0 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

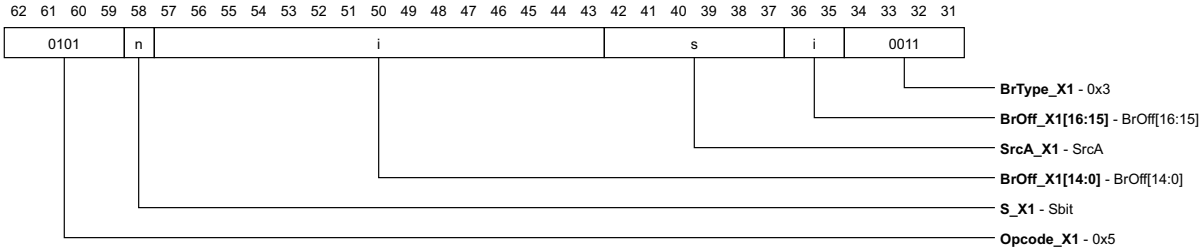

Figure 4-134: bnz in X1 Bit Descriptions

bnzt: Branch Not Zero Predict Taken Word

Syntax

```
bnzt SrcA, BrOff
```

Example

```
bnzt r5, target
```

Description

Branches to the target if the source operand is not equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch not 0 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

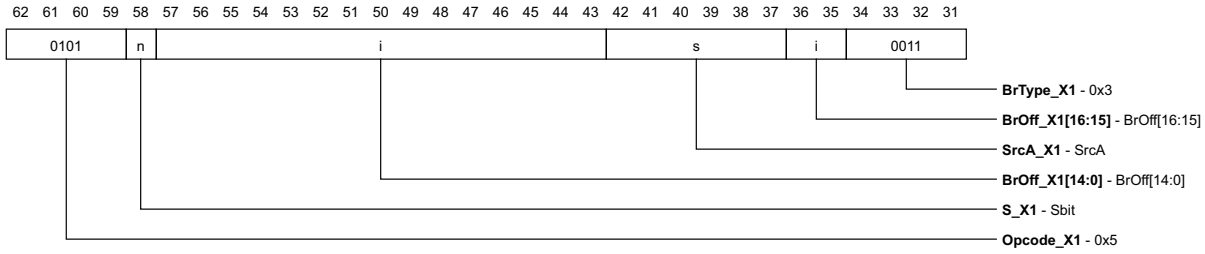

Figure 4-135: bnzt in X1 Bit Descriptions

bz: Branch Zero Word

Syntax

```
bz SrcA, BrOff
```

Example

```
bz r5, target
```

Description

Branches to the target if the source operand is equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch 0 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

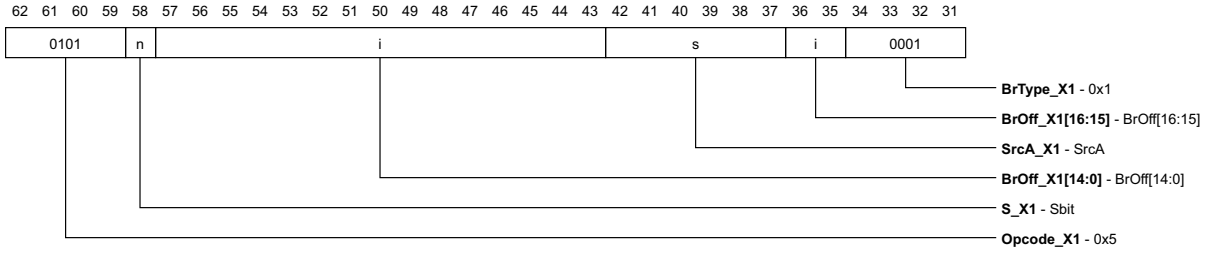

Figure 4-136: bz in X1 Bit Descriptions

bzt: Branch Zero Predict Taken Word

Syntax

```
bzt SrcA, BrOff
```

Example

```
bzt r5, target
```

Description

Branches to the target if the source operand is equal to 0. Otherwise, the program counter advances to the next instruction in program order. Branch 0 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

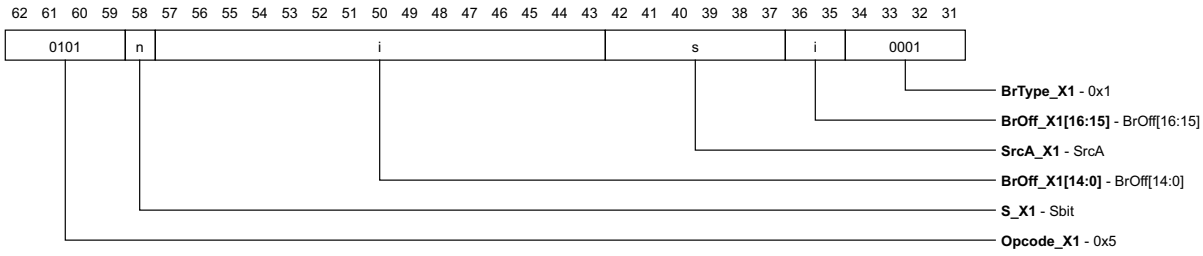

Figure 4-137: bzt in X1 Bit Descriptions

jalb: Jump and Link Backward

Syntax

```
jalb JOff
```

Example

```
jalb target
```

Description

Unconditionally jumps to a backward 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() + BACKWARD_OFFSET +
          (JOff << (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
jumped();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

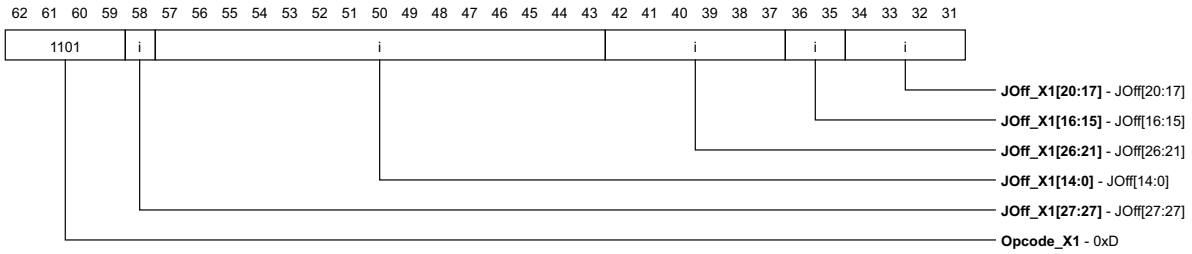

Figure 4-138: jalb in X1 Bit Descriptions

jalf: Jump and Link Forward

Syntax

```
jalf JOff
```

Example

```
jalf target
```

Description

Unconditionally jumps to a forward 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() +
          (JOff << (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
jumped();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

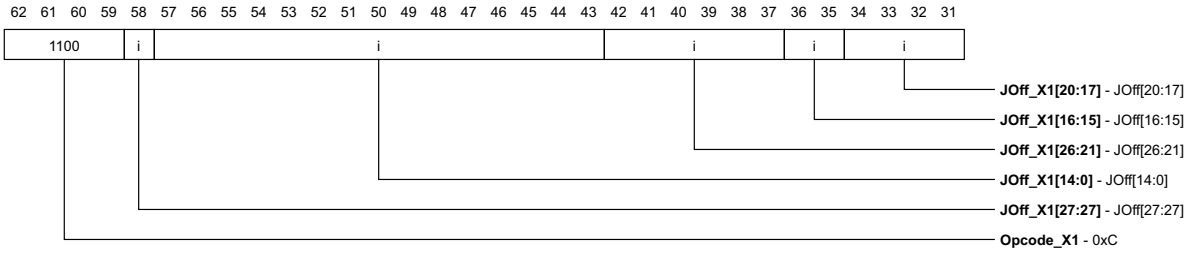

Figure 4-139: jalf in X1 Bit Descriptions

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			

Encoding

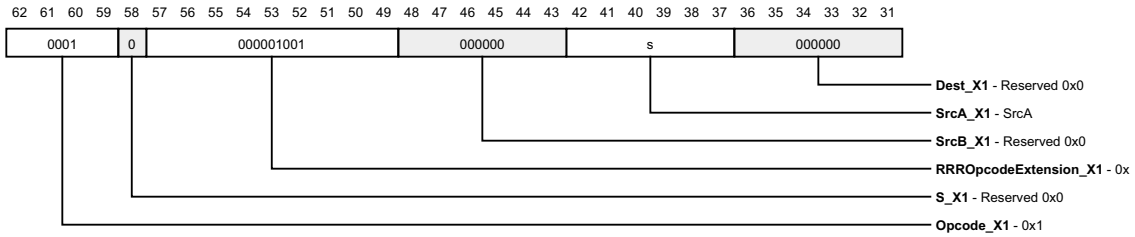

Figure 4-140: jalr in X1 Bit Descriptions

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			

Encoding

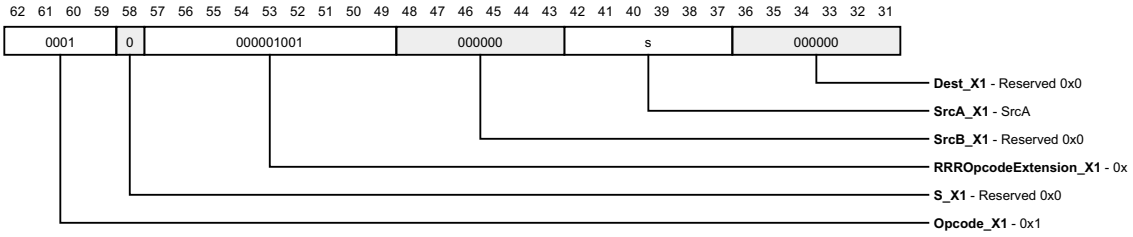

Figure 4-141: jalrp in X1 Bit Descriptions

jb: Jump Backward

Syntax

```
jb JOff
```

Example

```
jb target
```

Description

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

Functional Description

```
setNextPC(getCurrentPC() + BACKWARD_OFFSET +
    (JOff << (INSTRUCTION_SIZE_LOG_2 - BYTE_SIZE_LOG_2)));
jumped();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

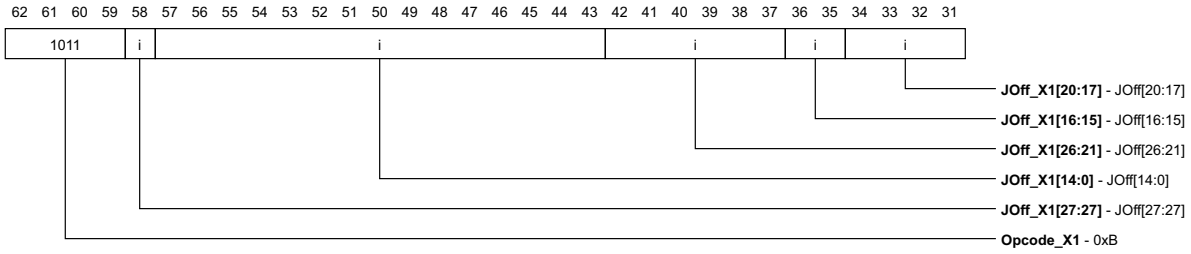

Figure 4-142: jb in X1 Bit Descriptions

jf: Jump Forward

Syntax

```
jf JOff
```

Example

```
jf target
```

Description

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

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

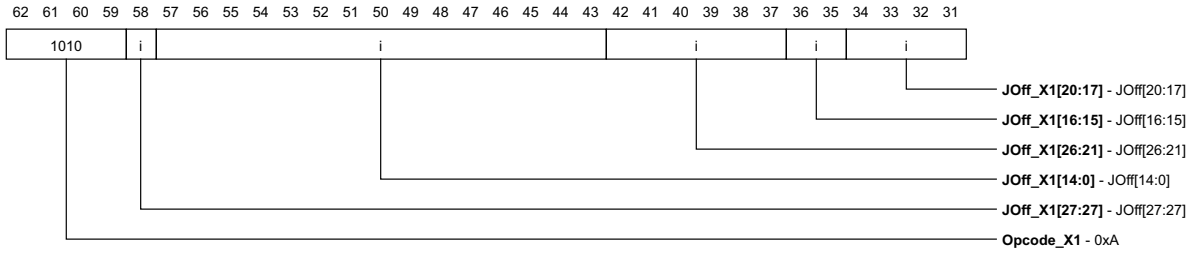

Figure 4-143: jf in X1 Bit Descriptions

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			

Encoding

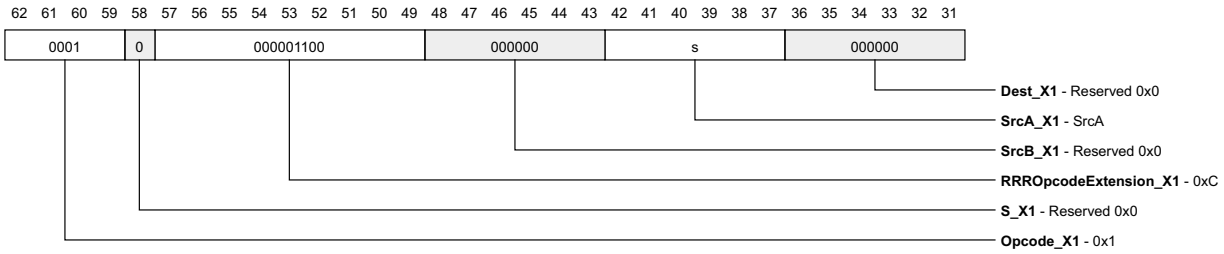

Figure 4-144: jr in X1 Bit Descriptions

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			

Encoding

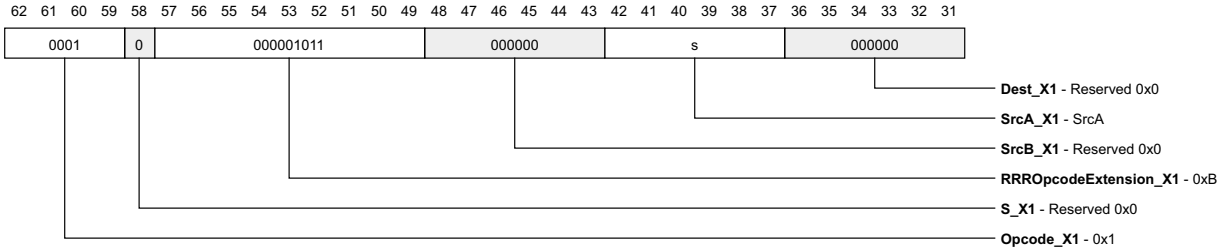

Figure 4-145: jrp in X1 Bit Descriptions

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			

Encoding

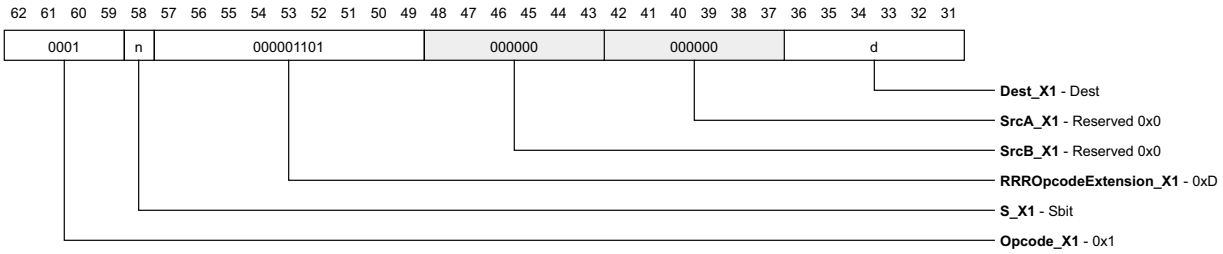

Figure 4-146: lnk in X1 Bit Descriptions

4.1.8 Logical Instructions

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

- and: And Word
- andi: And Immediate Word
- mm: Masked Merge Word
- mnz: Mask Not Zero Word
- mvnz: Move Not Zero Word
- mvz: Move Zero Word
- mz: Mask Zero Word
- nor: Nor Word
- or: Or Word
- ori: Or Immediate Word
- rl: Rotate Left Word
- rli: Rotate Left Immediate Word
- shl: Logical Shift Left Word
- shli: Logical Shift Left Immediate Word
- shr: Logical Shift Right Word
- shri: Logical Shift Right Immediate Word
- sra: Arithmetic Shift Right Word
- srai: Arithmetic Shift Right Immediate Word
- tblidxb0: Table Index Byte 0
- tblidxb1: Table Index Byte 1
- tblidxb2: Table Index Byte 2
- tblidxb3: Table Index Byte 3
- xor: Exclusive Or Word
- xori: Exclusive Or Immediate Word

and: And Word

Syntax

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

Example

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

Description

Compute the boolean AND of two words.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

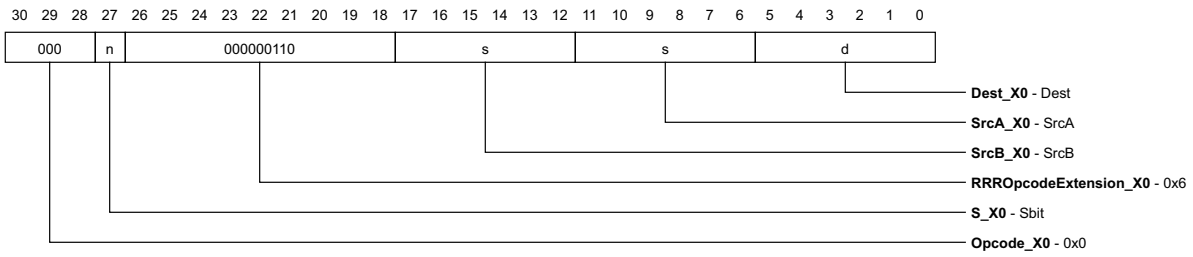

Figure 4-147: and in X0 Bit Descriptions

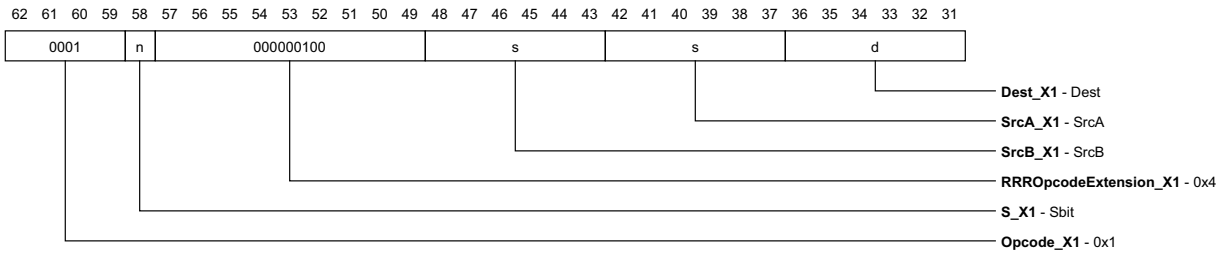

Figure 4-148: and in X1 Bit Descriptions

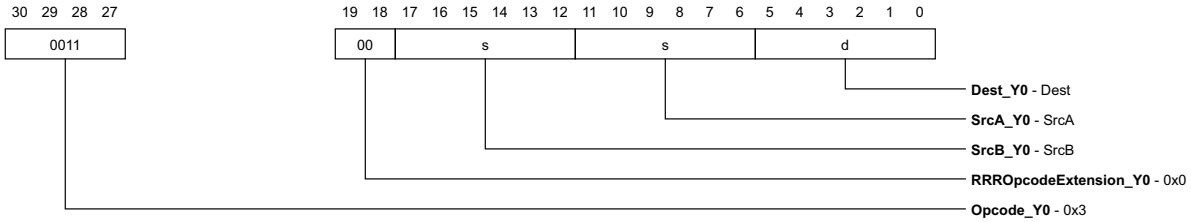

Figure 4-149: and in Y0 Bit Descriptions

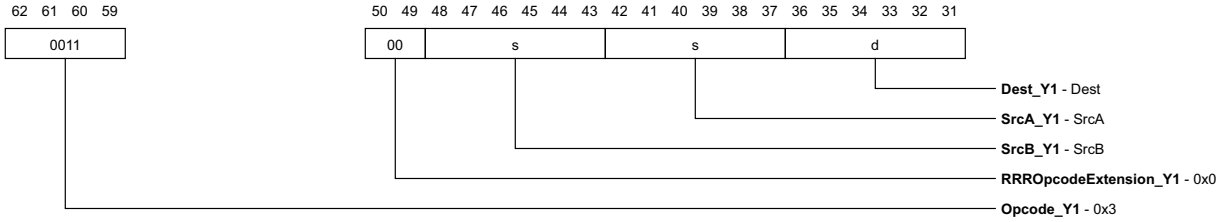

Figure 4-150: and in Y1 Bit Descriptions

andi: And Immediate Word

Syntax

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

Example

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

Description

Compute the boolean AND of a word 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

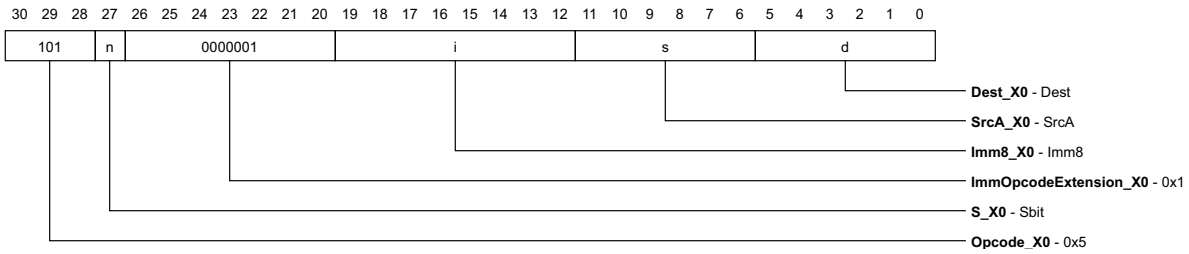

Figure 4-151: andi in X0 Bit Descriptions

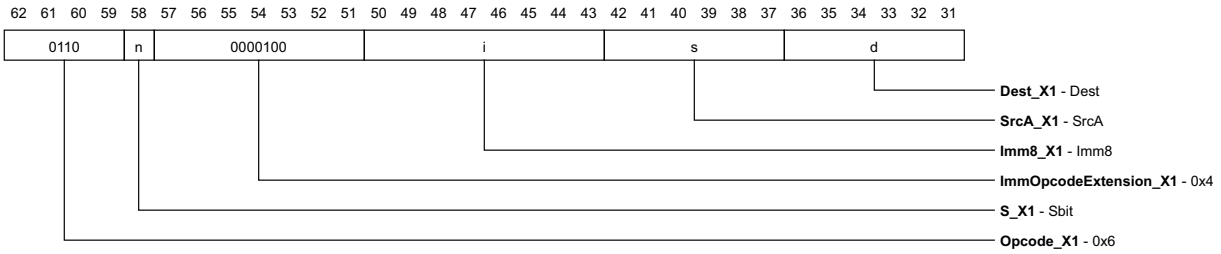

Figure 4-152: andi in X1 Bit Descriptions

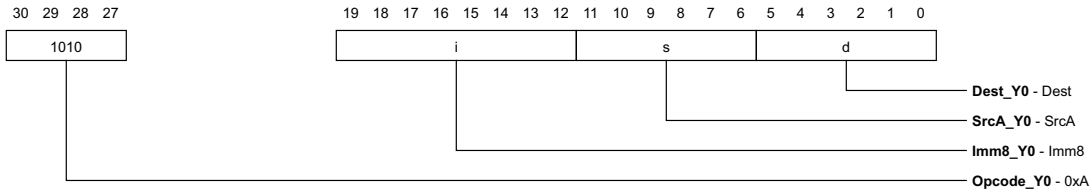

Figure 4-153: andi in Y0 Bit Descriptions

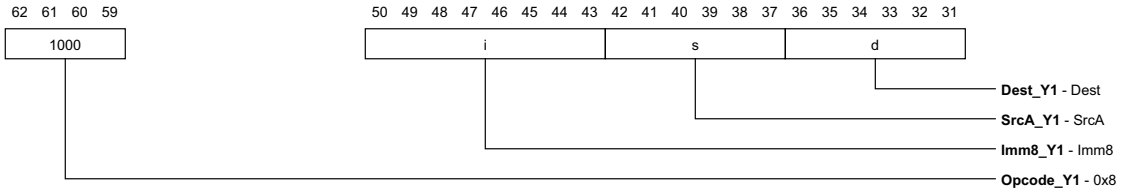

Figure 4-154: andi in Y1 Bit Descriptions

mm: Masked Merge Word

Syntax

```
mm Dest, SrcA, SrcB, MMStart, MMEnd
```

Example

```
mm r5, r6, r7, 5, 7
```

Description

Merge two source operands based on a running mask. The mask is specified by the `MMstart` and `MMend` fields, which contain the mask's starting and ending bit positions. 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 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 0 bit position up to the end bit position. The mask selects bits out of the first source operand and the inverse of the mask selects bits out of the second source operand.

Functional Description

```
UnsignedMachineWord mask = 0;
int start;
int end;
start = MMStart;
end = MMEnd;
mask =
    (start <=
     end) ? ((WORD_MASK << start) ^ ((WORD_MASK << end) << 1))
          : ((WORD_MASK << start) | (WORD_MASK >> ((WORD_SIZE - 1) - end)));
rf[Dest] = (rf[SrcA] & mask) | (rf[SrcB] & (WORD_MASK ^ mask));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

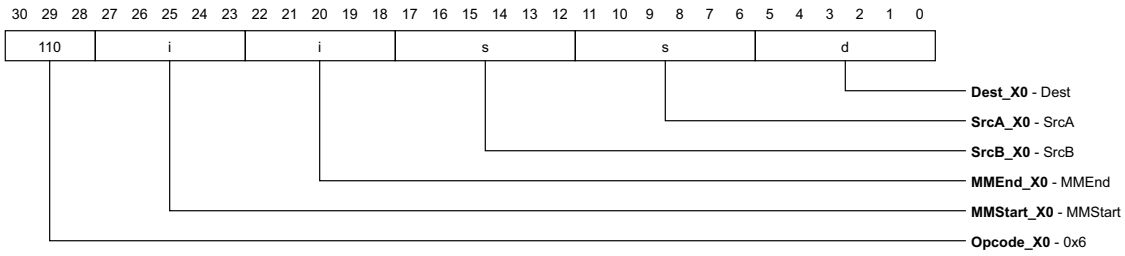

Figure 4-155: mm in x0 Bit Descriptions

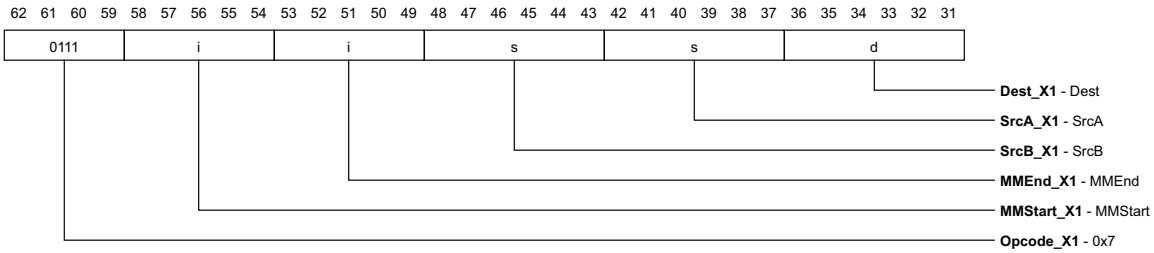

Figure 4-156: mm in X1 Bit Descriptions

mnz: Mask Not Zero Word

Syntax

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

Example

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

Description

If the first operand is not 0, 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

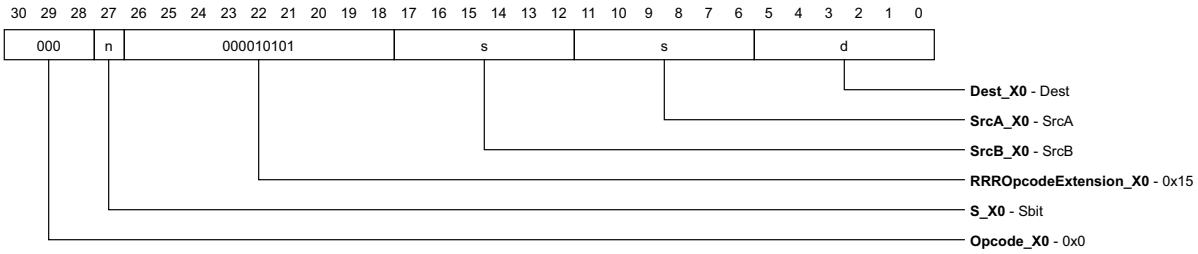

Figure 4-157: mnz in X0 Bit Descriptions

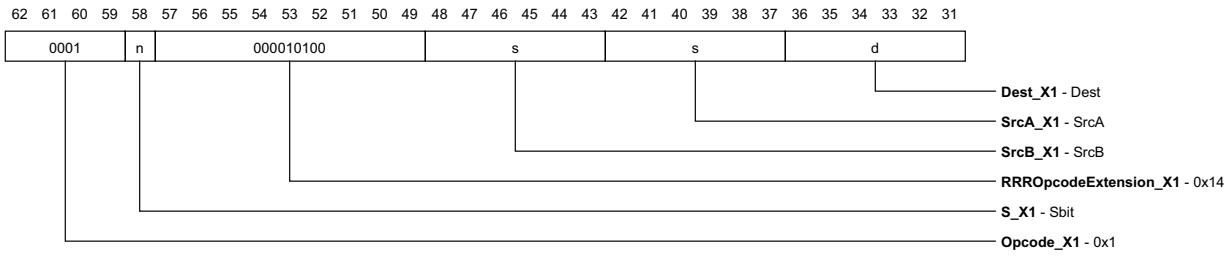

Figure 4-158: mnz in X1 Bit Descriptions

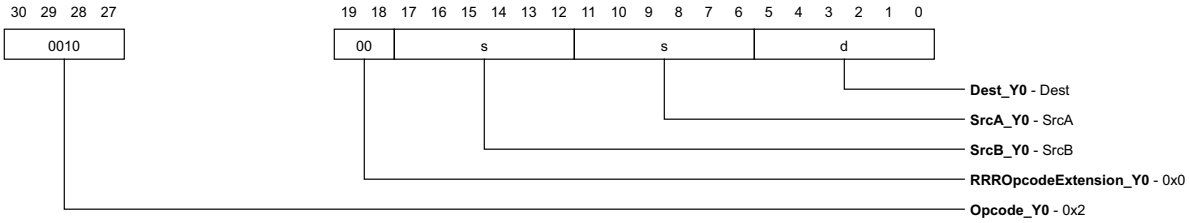

Figure 4-159: mnz in Y0 Bit Descriptions

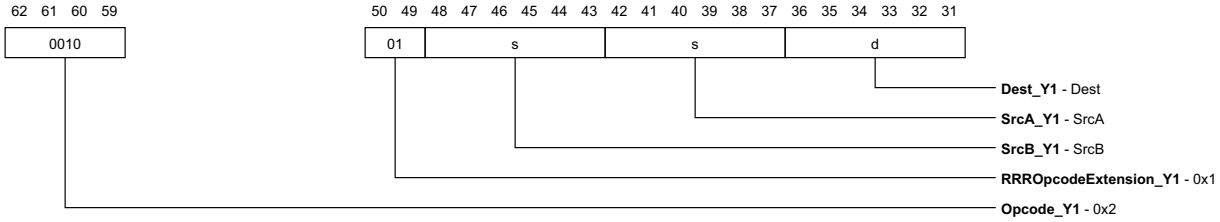

Figure 4-160: mnz in Y1 Bit Descriptions

mvnz: Move Not Zero Word

Syntax

```
mvnz Dest, SrcA, SrcB
```

Example

```
mvnz r5, r6, r7
```

Description

If the first source operand is not 0, move the second operand to the destination. Else, 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

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

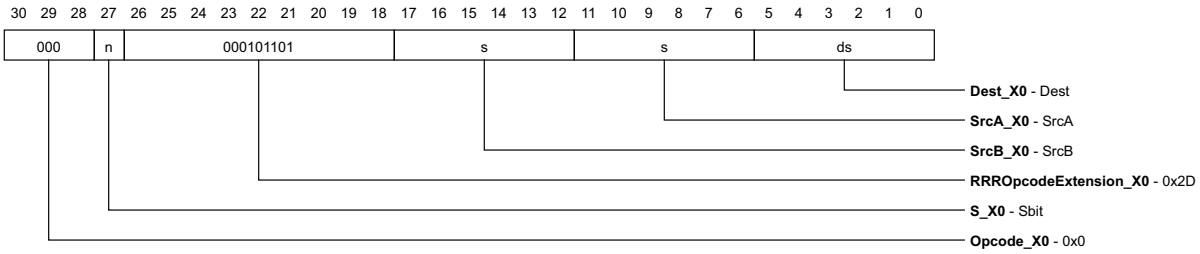

Figure 4-161: mvnz in X0 Bit Descriptions

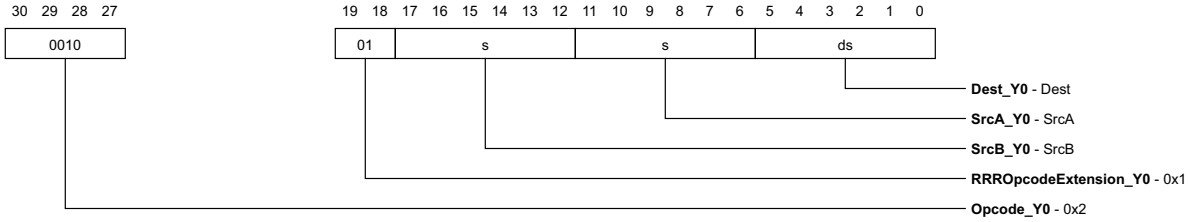

Figure 4-162: mvnz in Y0 Bit Descriptions

mvz: Move Zero Word

Syntax

```
mvz Dest, SrcA, SrcB
```

Example

```
mvz r5, r6, r7
```

Description

If the first source operand is 0, move the second operand to the destination. Else, 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

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

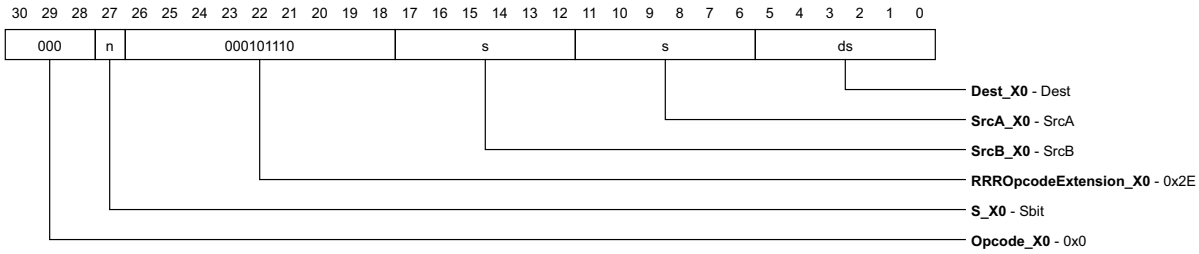

Figure 4-163: mvz in X0 Bit Descriptions

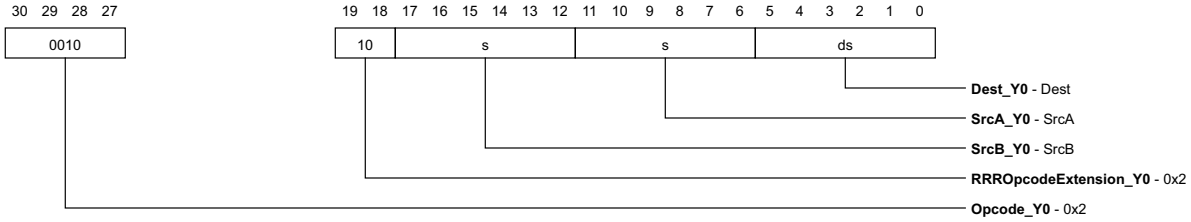

Figure 4-164: mvz in Y0 Bit Descriptions

mz: Mask Zero Word

Syntax

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

Example

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

Description

If the first operand is 0, 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

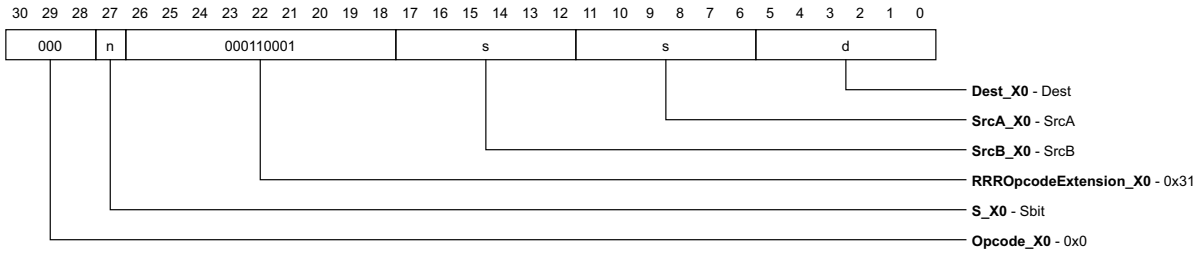

Figure 4-165: mz in X0 Bit Descriptions

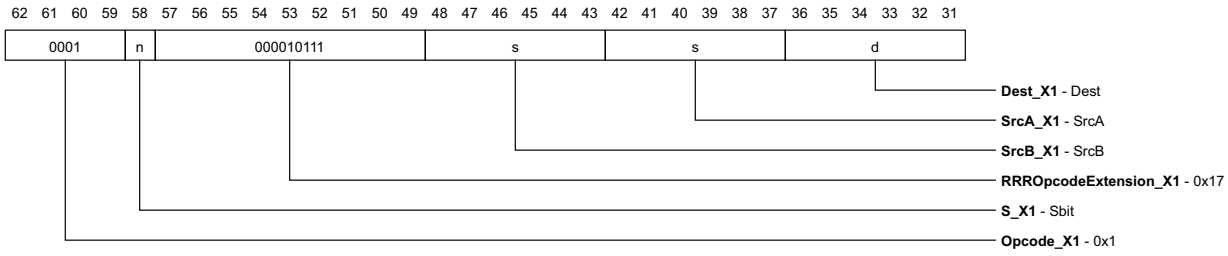

Figure 4-166: mz in X1 Bit Descriptions

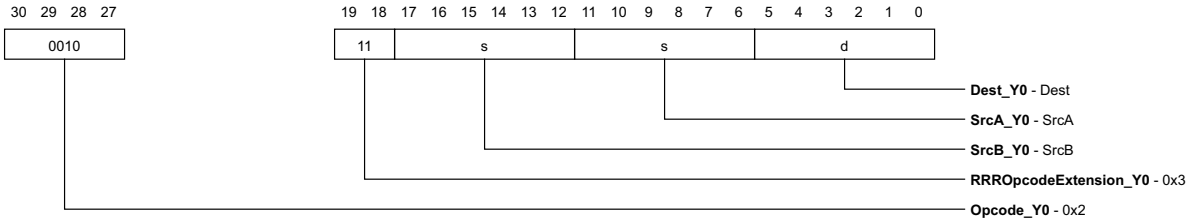

Figure 4-167: mz in Y0 Bit Descriptions

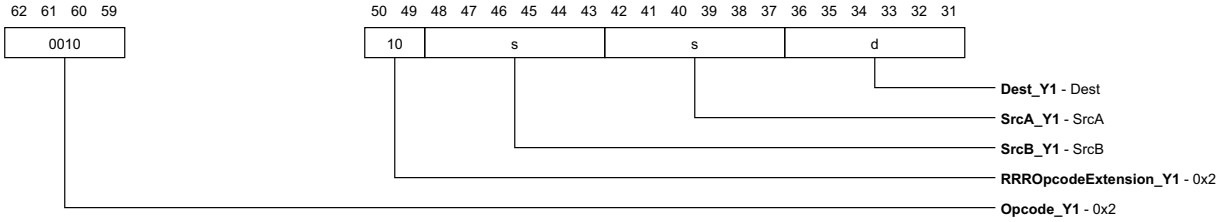

Figure 4-168: mz in Y1 Bit Descriptions

nor: Nor Word

Syntax

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

Example

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

Description

Computer the boolean NOR of two words.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

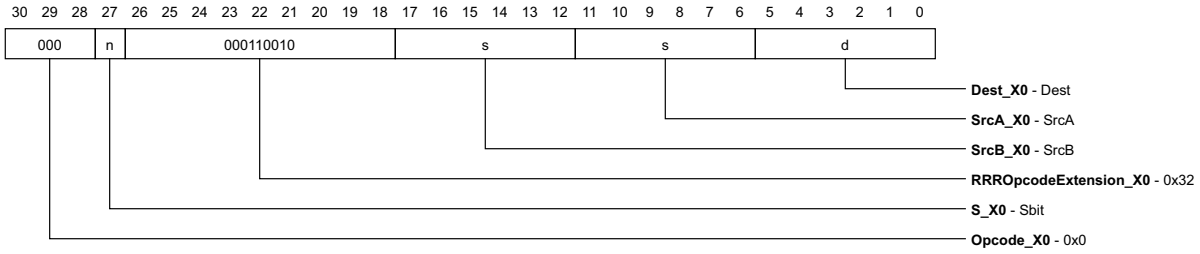

Figure 4-169: nor in X0 Bit Descriptions

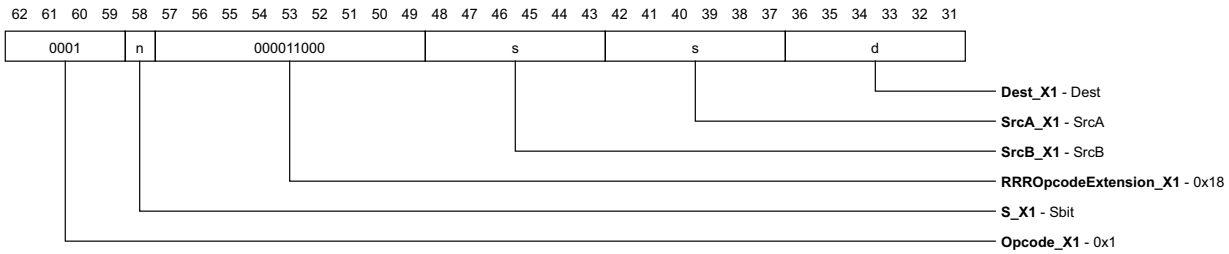

Figure 4-170: nor in X1 Bit Descriptions

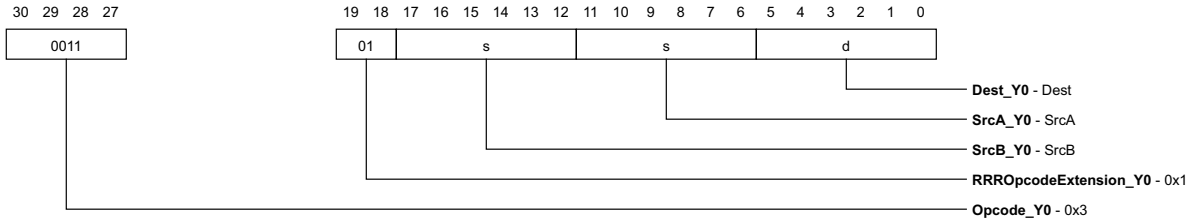

Figure 4-171: nor in Y0 Bit Descriptions

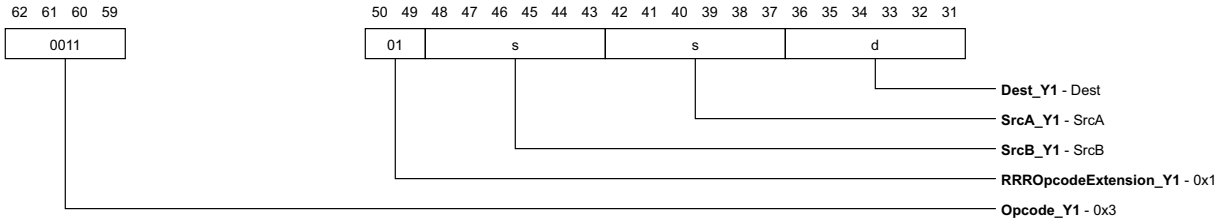

Figure 4-172: nor in Y1 Bit Descriptions

or: Or Word

Syntax

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

Example

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

Description

Compute the boolean OR of two words.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

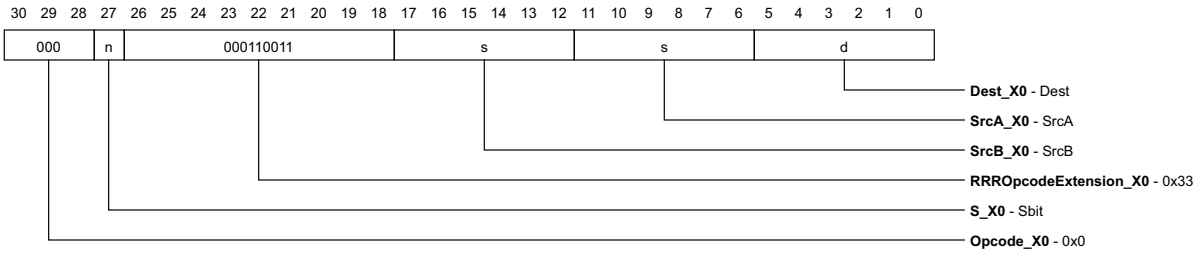

Figure 4-173: or in X0 Bit Descriptions

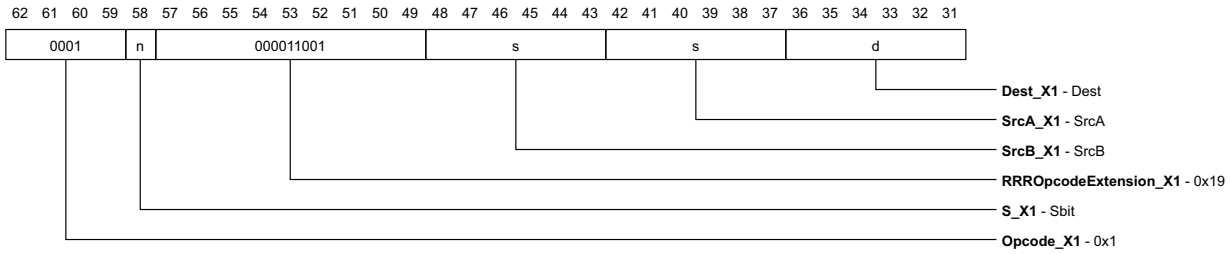

Figure 4-174: or in X1 Bit Descriptions

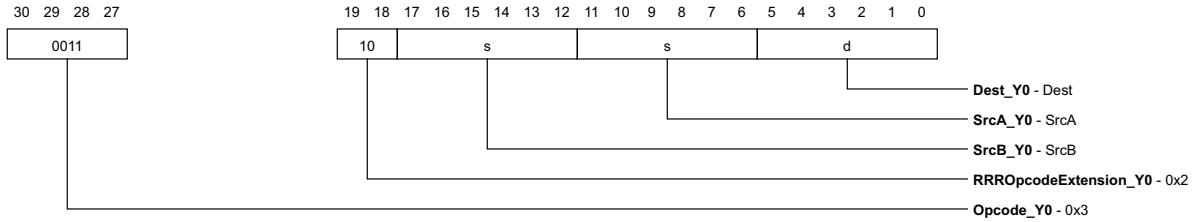

Figure 4-175: or in Y0 Bit Descriptions

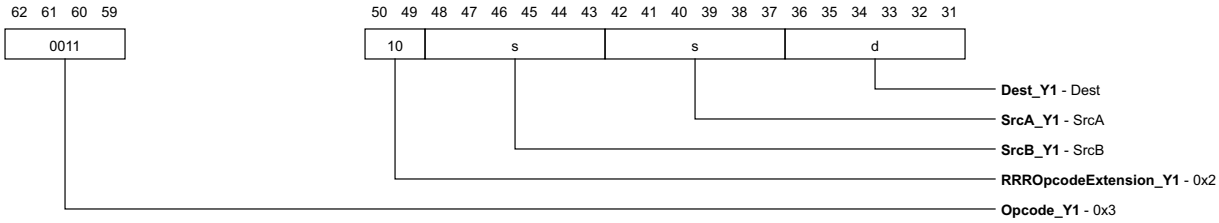

Figure 4-176: or in Y1 Bit Descriptions

ori: Or Immediate Word

Syntax

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

Example

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

Description

Compute the boolean OR of a word 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

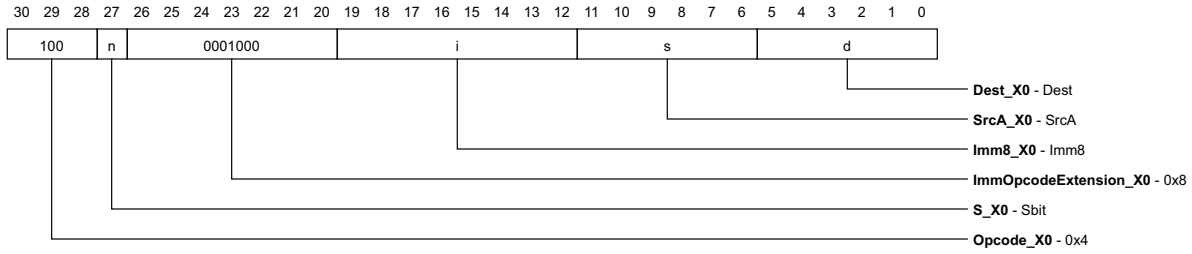

Figure 4-177: ori in X0 Bit Descriptions

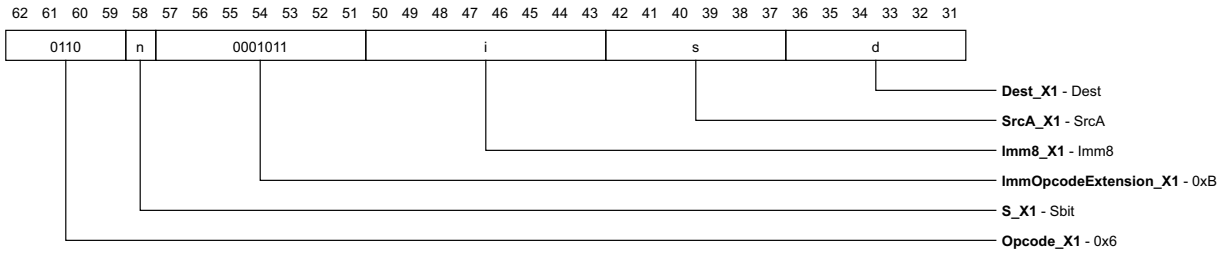

Figure 4-178: ori in X1 Bit Descriptions

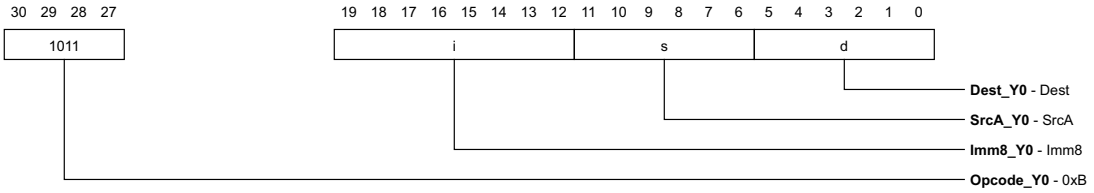

Figure 4-179: ori in Y0 Bit Descriptions

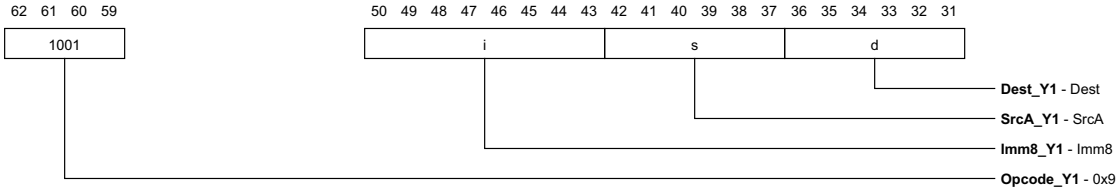

Figure 4-180: ori in Y1 Bit Descriptions

rl: Rotate Left Word

Syntax

```
rl Dest, SrcA, SrcB
```

Example

```
rl r5, r6, r7
```

Description

Rotate the first source operand to the left by the second source operand. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. The main processor ISA does not contain a rotate right.

Functional Description

```
rf[Dest] =
    ((rf[SrcA] << (rf[SrcB] % WORD_SIZE)) |
     (((UnsignedMachineWord) rf[SrcA]) >>
      ((WORD_SIZE - (rf[SrcB] % WORD_SIZE)) % WORD_SIZE)));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

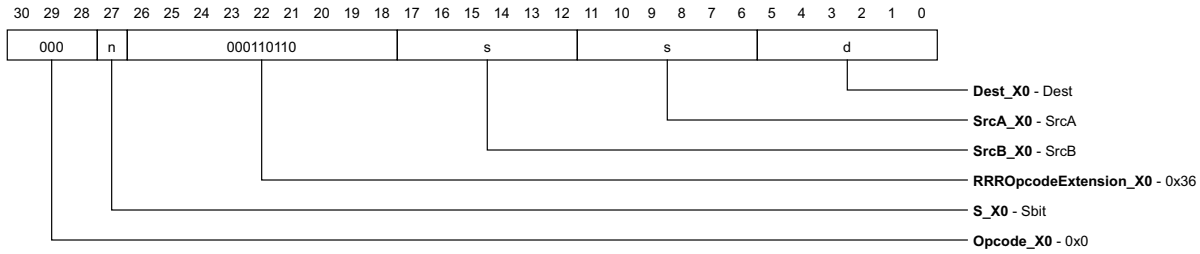

Figure 4-181: rl in X0 Bit Descriptions

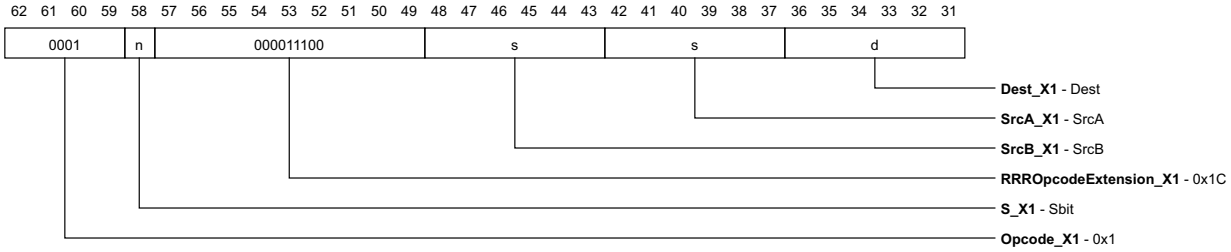

Figure 4-182: rl in X1 Bit Descriptions

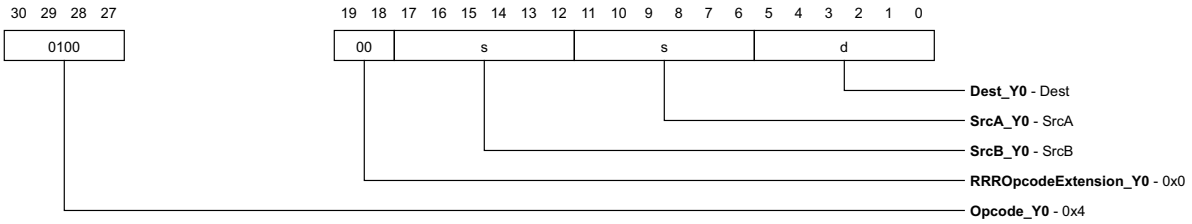

Figure 4-183: rl in Y0 Bit Descriptions

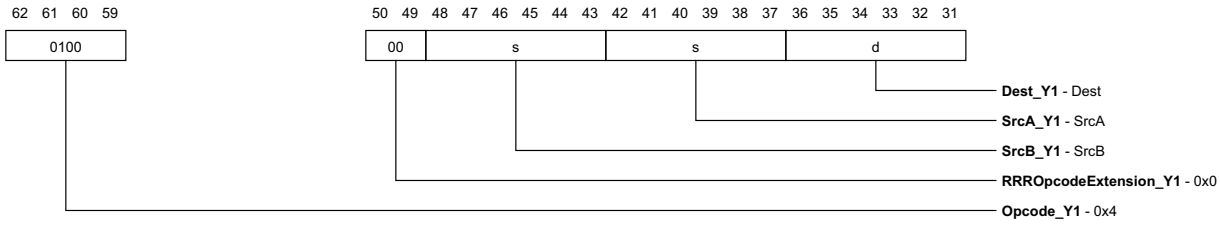

Figure 4-184: rl in Y1 Bit Descriptions

rli: Rotate Left Immediate Word

Syntax

```
rli Dest, SrcA, ShAmt
```

Example

```
rli r5, r6, 5
```

Description

Rotate the first source operand to the left by an immediate. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. The main processor ISA does not contain a rotate right.

Functional Description

```
rf[Dest] =
    ((rf[SrcA] << (((UnsignedMachineWord) ShAmt) % WORD_SIZE)) |
     (((UnsignedMachineWord) rf[SrcA]) >>
      ((WORD_SIZE -
        ((UnsignedMachineWord) ShAmt) % WORD_SIZE) % WORD_SIZE)));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

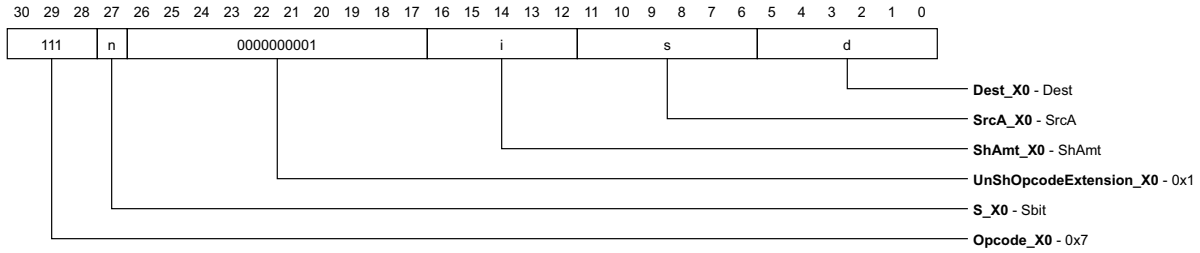

Figure 4-185: rli in X0 Bit Descriptions

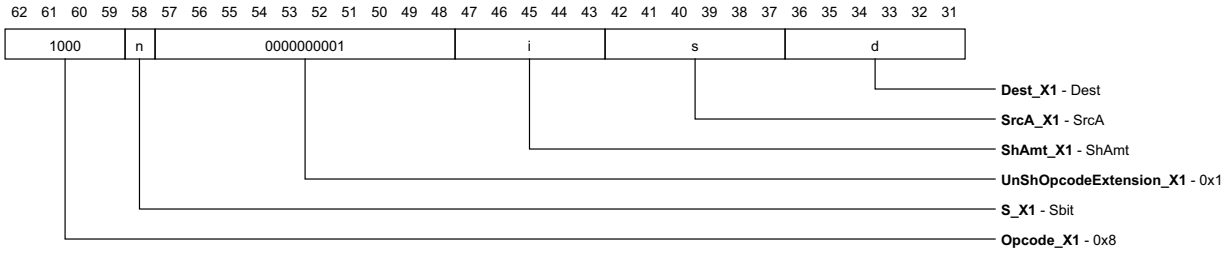

Figure 4-186: rli in X1 Bit Descriptions

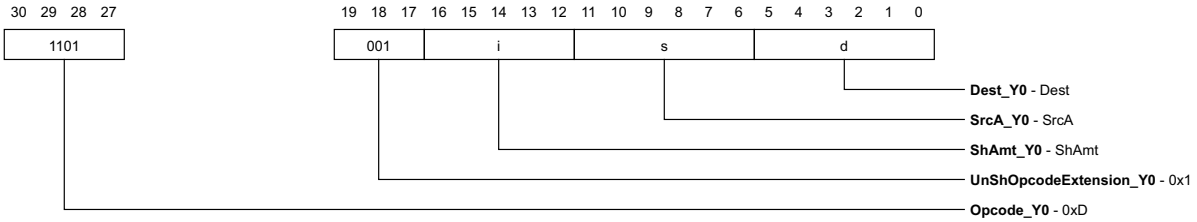

Figure 4-187: rli in Y0 Bit Descriptions

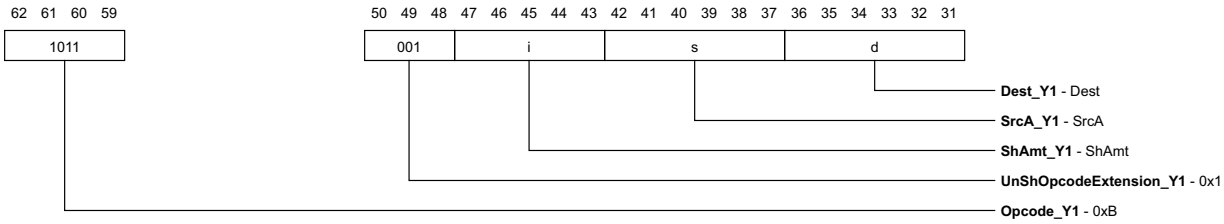

Figure 4-188: rli in Y1 Bit Descriptions

shl: Logical Shift Left Word

Syntax

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

Example

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

Description

Logically shift the first source operand to the left by the second source operand. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. Left shifts shift zeros into the low ordered bits in a word and are suitable to be used as unsigned multiplication by powers of 2.

Functional Description

```
rf[Dest] = rf[SrcA] << (rf[SrcB] % WORD_SIZE);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

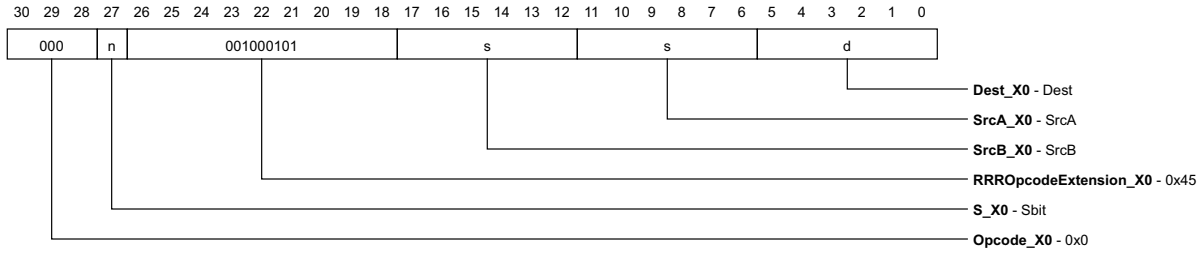

Figure 4-189: shl in X0 Bit Descriptions

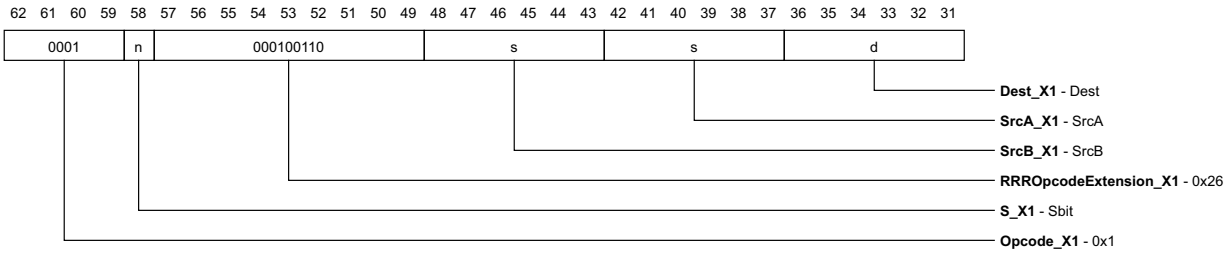

Figure 4-190: shl in X1 Bit Descriptions

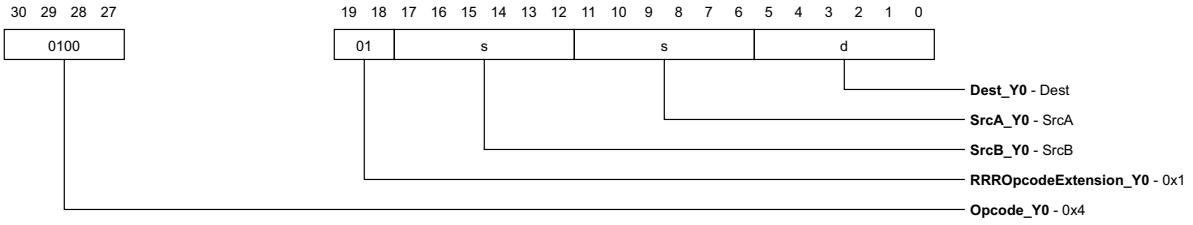

Figure 4-191: shl in Y0 Bit Descriptions

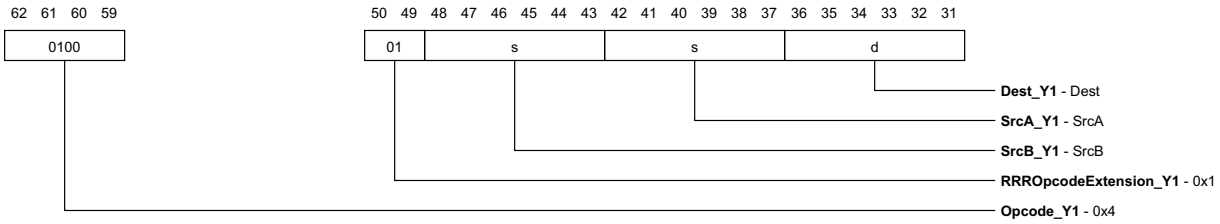

Figure 4-192: shl in Y1 Bit Descriptions

shli: Logical Shift Left Immediate Word

Syntax

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

Example

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

Description

Logically shift the first source operand to the left by an immediate. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. Left shifts shift zeros into the low ordered bits in a word and are suitable to be used as unsigned multiplication by powers of 2.

Functional Description

```
rf[Dest] = rf[SrcA] << (((UnsignedMachineWord) ShAmt) % WORD_SIZE);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

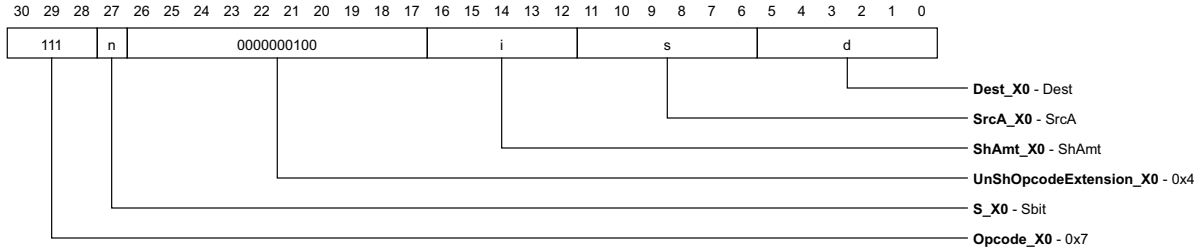

Figure 4-193: shli in X0 Bit Descriptions

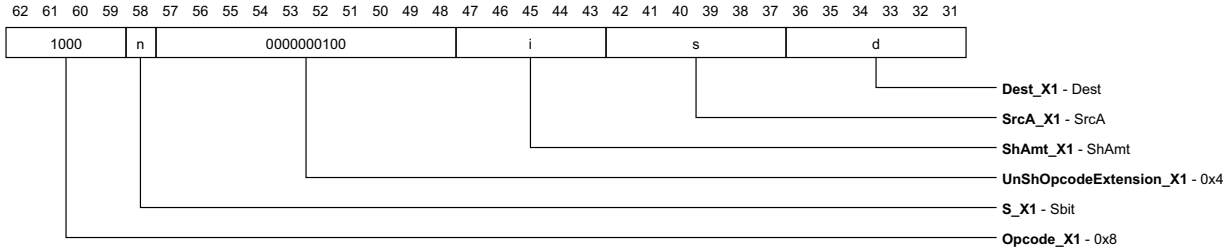

Figure 4-194: shli in X1 Bit Descriptions

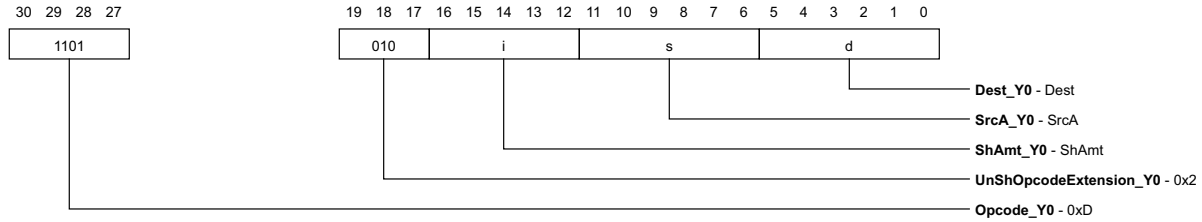

Figure 4-195: shli in Y0 Bit Descriptions

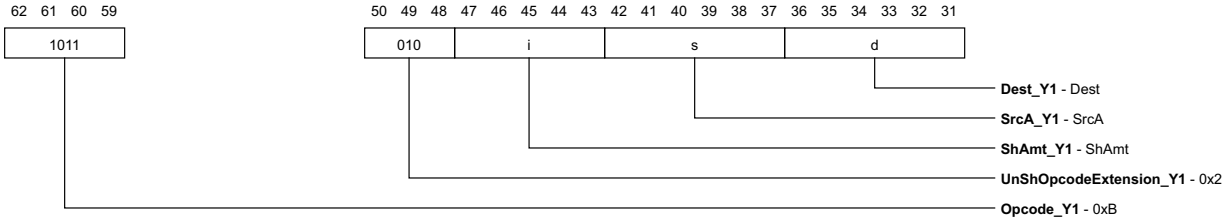

Figure 4-196: shli in Y1 Bit Descriptions

shr: Logical Shift Right Word

Syntax

```
shr Dest, SrcA, SrcB
```

Example

```
shr r5, r6, r7
```

Description

Logically shift the first source operand to the right by the second source operand. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. Logical right shifts shift zeros into the high ordered bits in a word and are suitable to be used as unsigned integer division by powers of 2.

Functional Description

```
rf[Dest] = (UnsignedMachineWord) rf[SrcA] >> (rf[SrcB] % WORD_SIZE);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

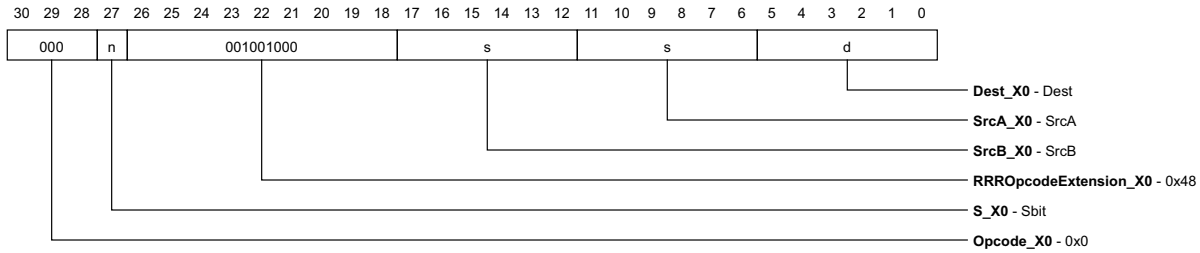

Figure 4-197: shr in X0 Bit Descriptions

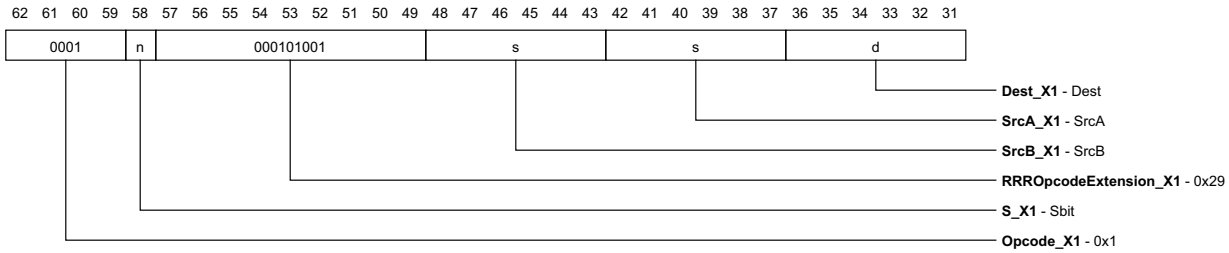

Figure 4-198: shr in X1 Bit Descriptions

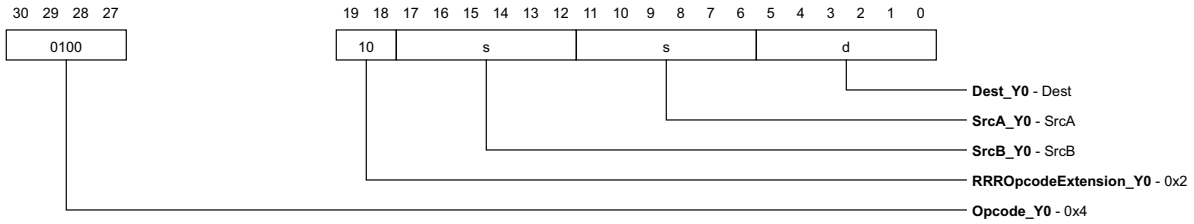

Figure 4-199: shr in Y0 Bit Descriptions

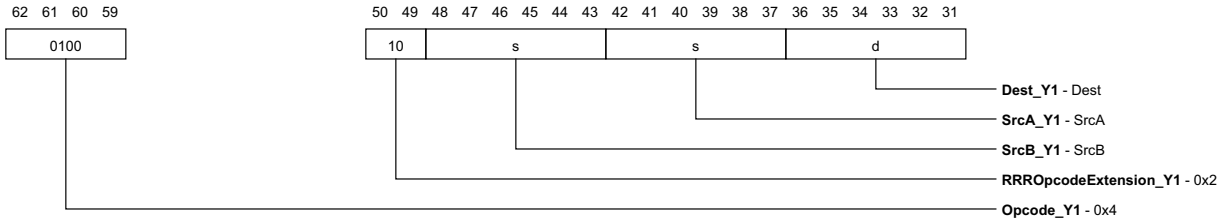

Figure 4-200: shr in Y1 Bit Descriptions

shri: Logical Shift Right Immediate Word

Syntax

```
shri Dest, SrcA, ShAmt
```

Example

```
shri r5, r6, 5
```

Description

Logically shift the first source operand to the right by an immediate. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. Logical right shifts shift zeros into the high ordered bits in a word and are suitable to be used as unsigned integer division by powers of 2.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

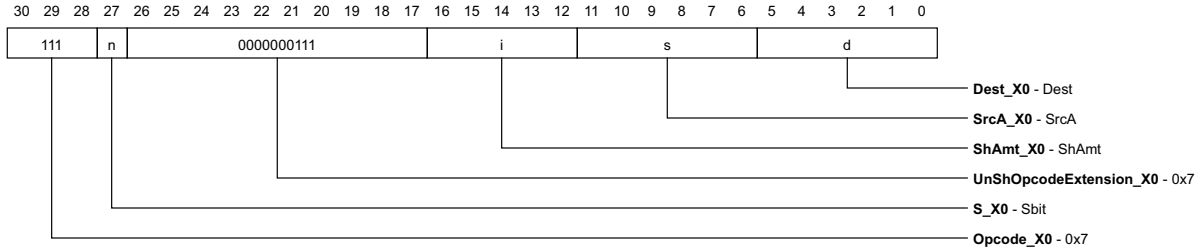

Figure 4-201: shri in X0 Bit Descriptions

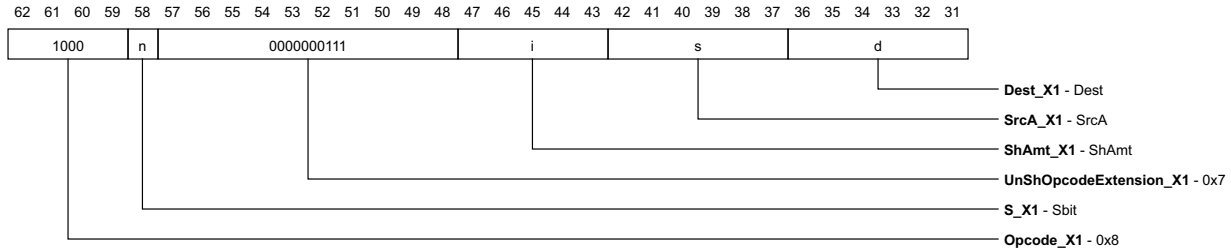

Figure 4-202: shri in X1 Bit Descriptions

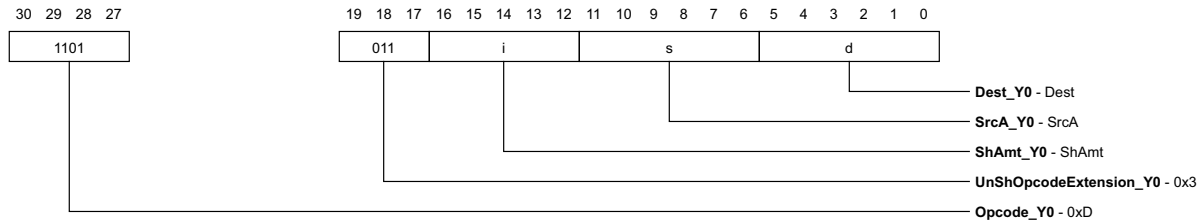

Figure 4-203: shri in Y0 Bit Descriptions

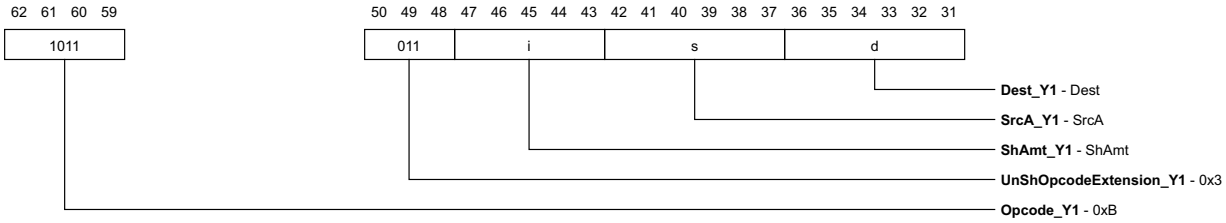

Figure 4-204: shri in Y1 Bit Descriptions

sra: Arithmetic Shift Right Word

Syntax

```
sra Dest, SrcA, SrcB
```

Example

```
sra r5, r6, r7
```

Description

Arithmetically shift the first source operand to the right by the second source operand. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. Arithmetic right shift shifts the high ordered bit into the high ordered bits in a word.

Functional Description

```
rf[Dest] = ((SignedMachineWord) rf[SrcA]) >> (rf[SrcB] % WORD_SIZE);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

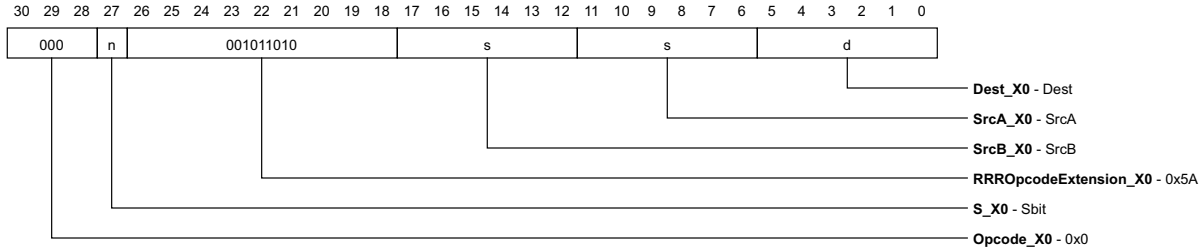

Figure 4-205: sra in X0 Bit Descriptions

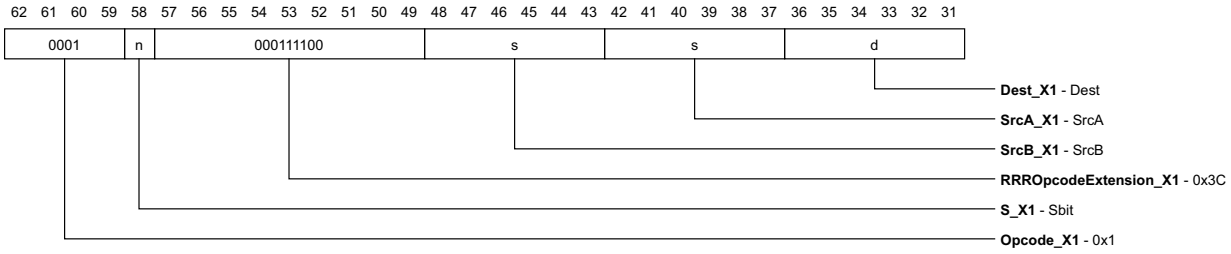

Figure 4-206: sra in X1 Bit Descriptions

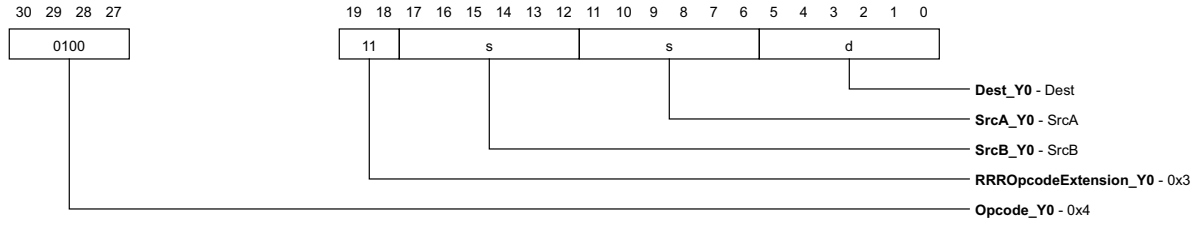

Figure 4-207: sra in Y0 Bit Descriptions

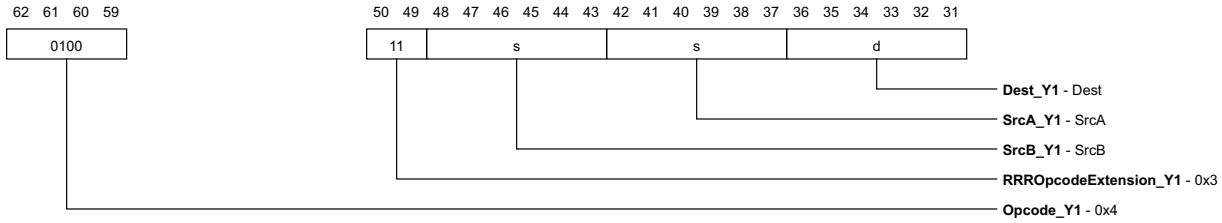

Figure 4-208: sra in Y1 Bit Descriptions

srai: Arithmetic Shift Right Immediate Word

Syntax

```
srai Dest, SrcA, ShAmt
```

Example

```
srai r5, r6, 5
```

Description

Arithmetically shift the first source operand to the right by an immediate. If the shift amount is larger than the number of bits in a word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a word. Arithmetic right shifts shift the high ordered bit into the high ordered bits in a word.

Functional Description

```
rf[Dest] =
    ((SignedMachineWord) rf[SrcA]) >> ((((UnsignedMachineWord) ShAmt))
                                       % WORD_SIZE);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

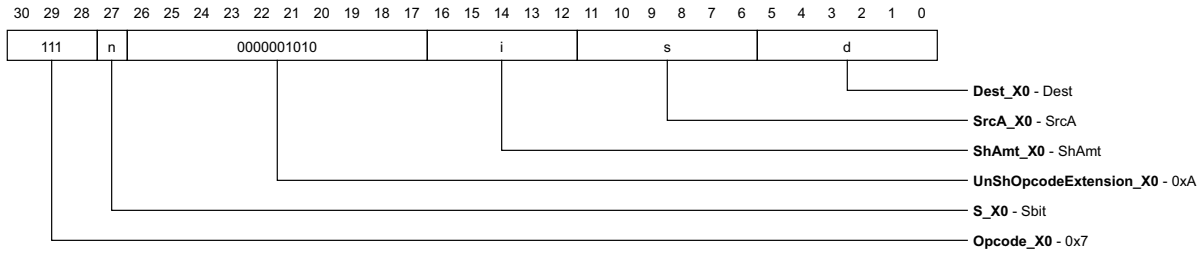

Figure 4-209: srai in X0 Bit Descriptions

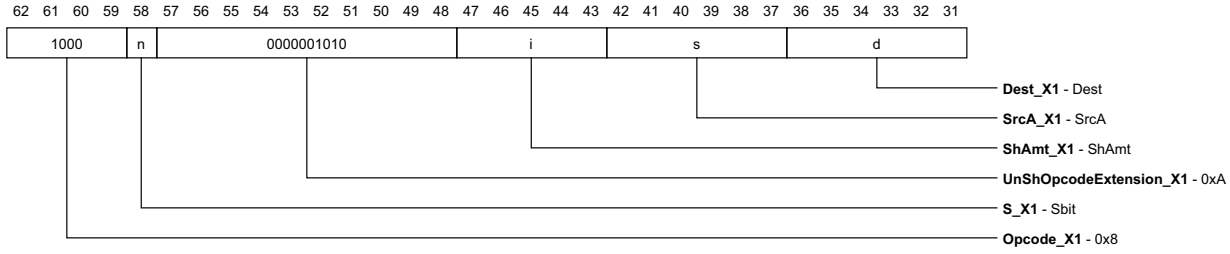

Figure 4-210: srai in X1 Bit Descriptions

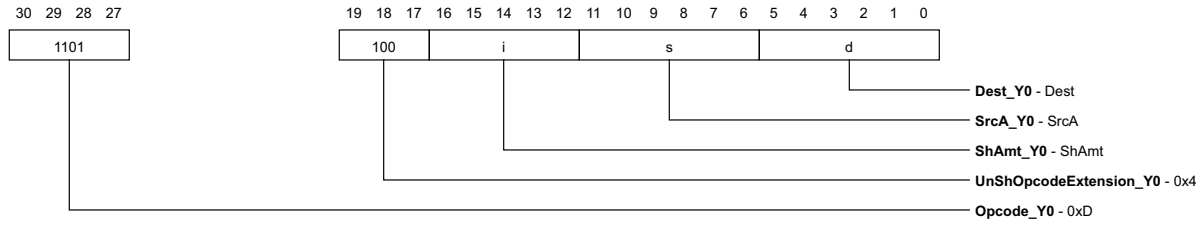

Figure 4-211: srai in Y0 Bit Descriptions

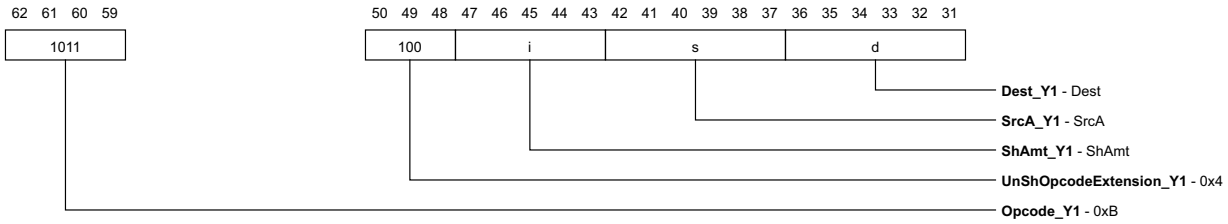

Figure 4-212: srai in Y1 Bit Descriptions

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

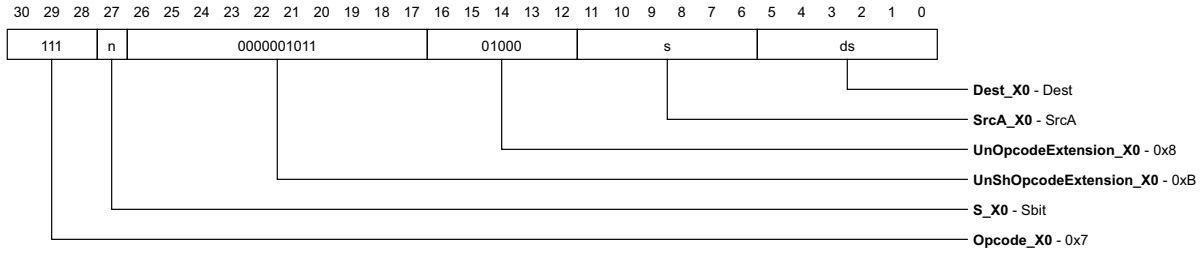

Figure 4-213: tblidxb0 in X0 Bit Descriptions

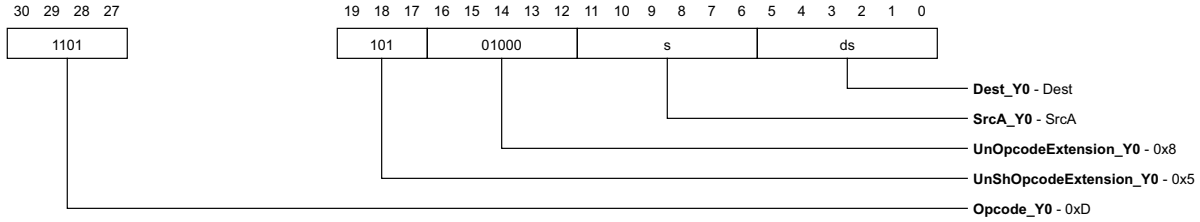

Figure 4-214: tblidxb0 in Y0 Bit Descriptions

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

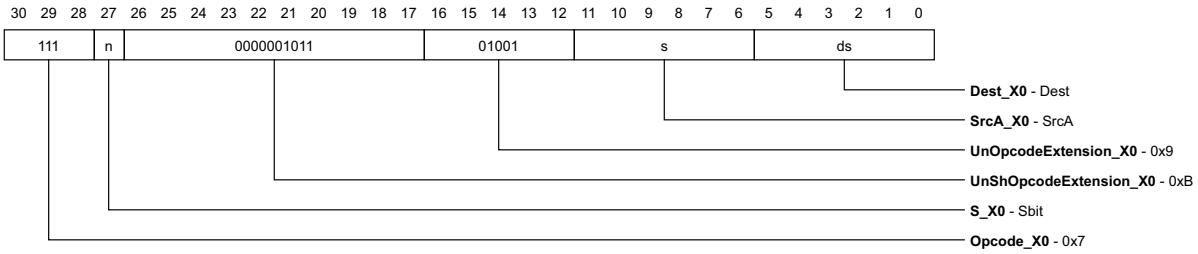

Figure 4-215: tblidxb1 in X0 Bit Descriptions

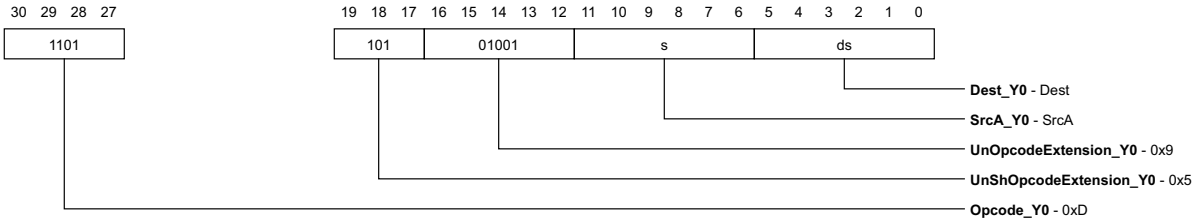

Figure 4-216: tblidxb1 in Y0 Bit Descriptions

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

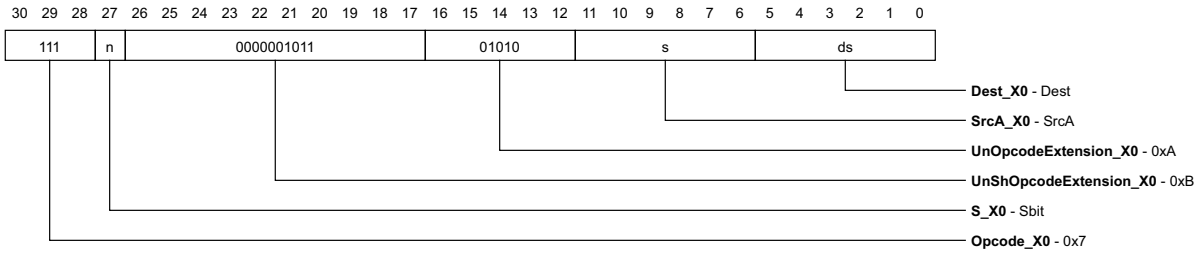

Figure 4-217: tblidxb2 in X0 Bit Descriptions

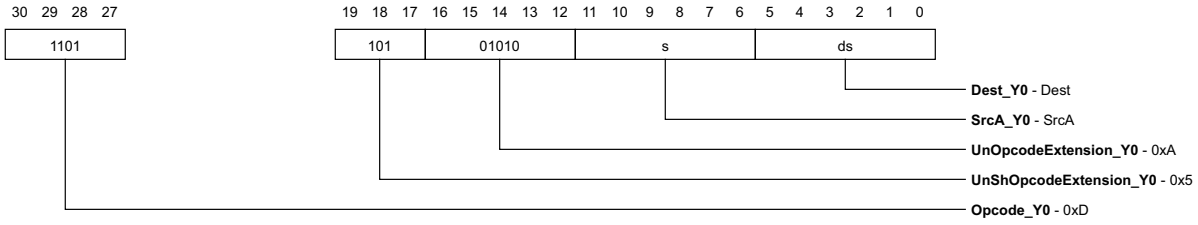

Figure 4-218: tblidxb2 in Y0 Bit Descriptions

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

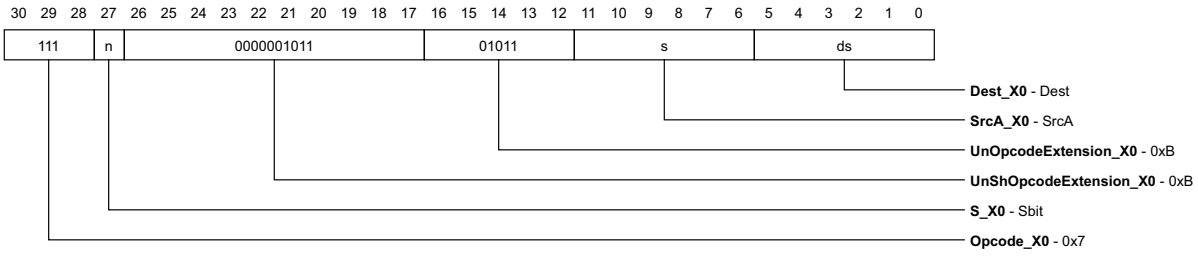

Figure 4-219: tblidxb3 in X0 Bit Descriptions

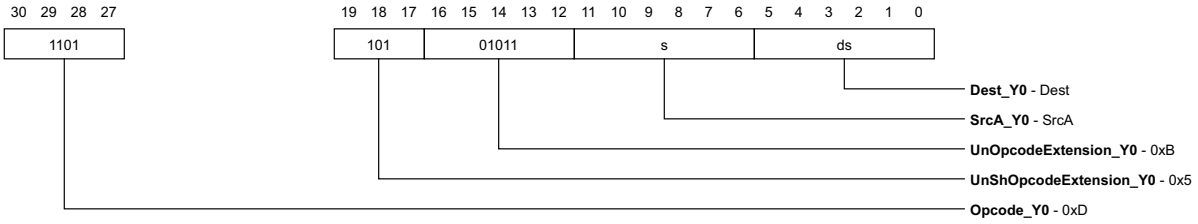

Figure 4-220: tblidxb3 in Y0 Bit Descriptions

xor: Exclusive Or Word

Syntax

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

Example

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

Description

Compute the boolean XOR of two words.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X	X	X	

Encoding

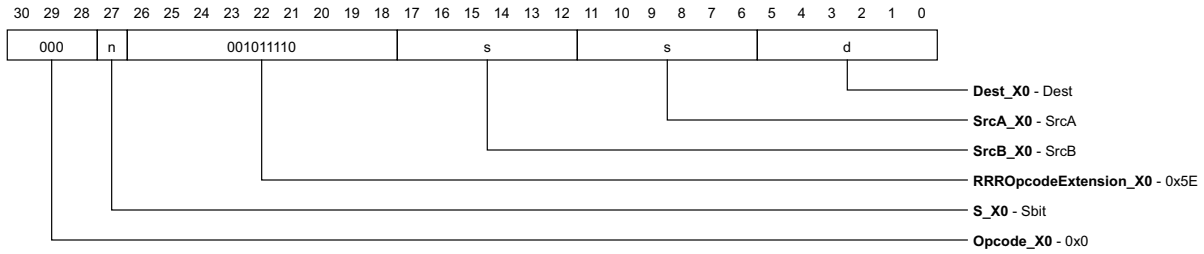

Figure 4-221: xor in X0 Bit Descriptions

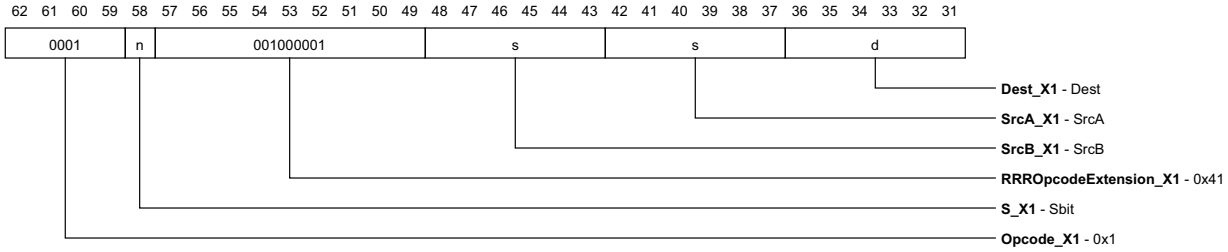

Figure 4-222: xor in X1 Bit Descriptions

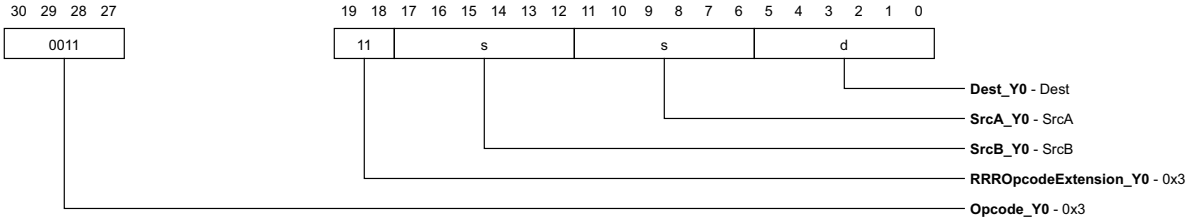

Figure 4-223: xor in Y0 Bit Descriptions

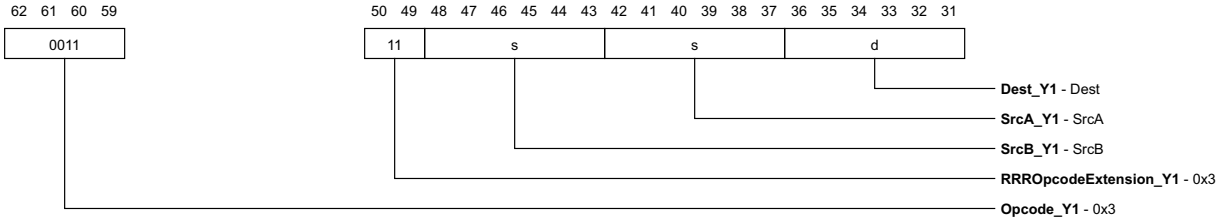

Figure 4-224: xor in Y1 Bit Descriptions

xori: Exclusive Or Immediate Word

Syntax

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

Example

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

Description

Compute the boolean XOR of a word and a sign extended immediate.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

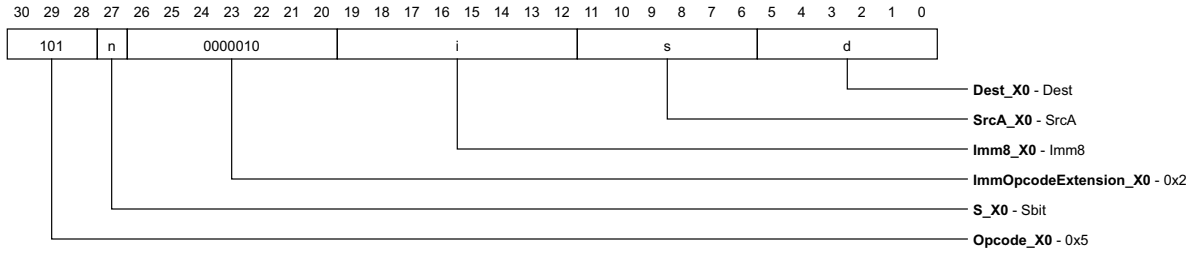

Figure 4-225: xori in X0 Bit Descriptions

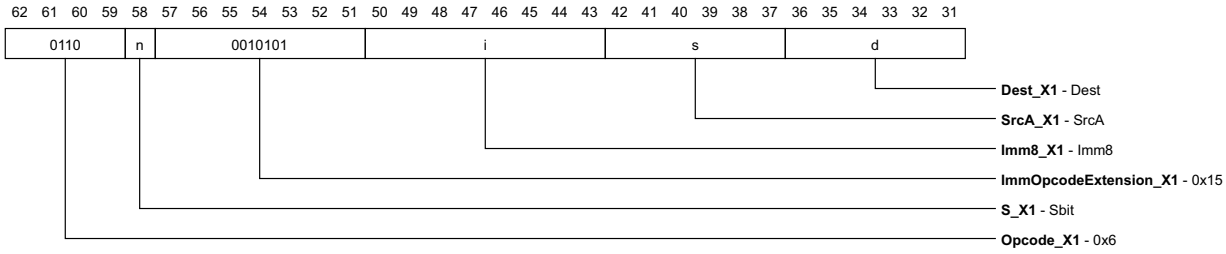

Figure 4-226: xori in X1 Bit Descriptions

4.1.9 Memory Instructions

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

- lb: Load Byte
- lb_u: Load Byte Unsigned
- lbadd: Load Byte and Add
- lbadd_u: Load Byte Unsigned and Add
- lh: Load Half Word
- lh_u: Load Half Word Unsigned
- lhadd: Load Half Word and Add
- lhadd_u: Load Half Word Unsigned and Add
- lw: Load Word
- lw_na: Load Word No Alignment Trap
- lwadd: Load Word and Add
- lwadd_na: Load Word No Alignment Trap and Add
- sb: Store Byte
- sbadd: Store Byte and Add
- sh: Store Half Word
- shadd: Store Half Word and Add
- sw: Store Word
- swadd: Store Word and Add
- tns: Test and Set Word

lb: Load Byte

Syntax

```
lb Dest, Src
```

Example

```
lb 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

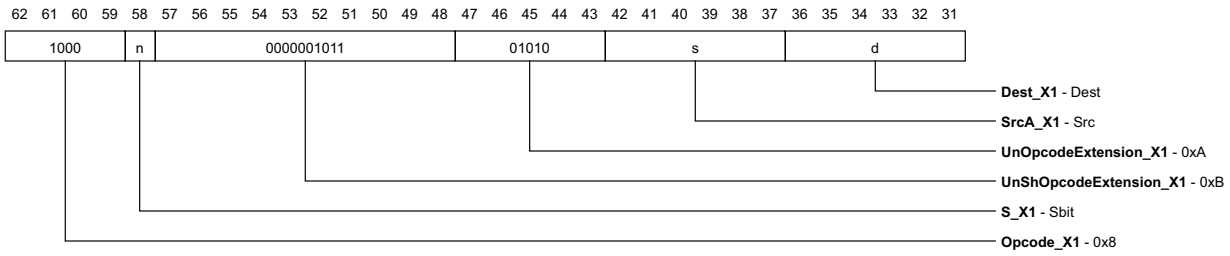

Figure 4-227: lb in X1 Bit Descriptions

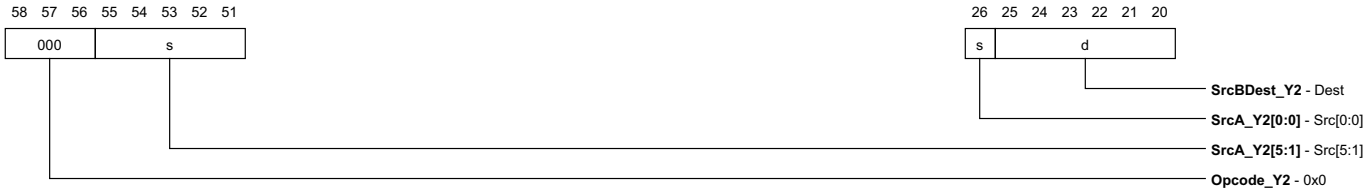

Figure 4-228: lb in Y2 Bit Descriptions

lb_u: Load Byte Unsigned

Syntax

```
lb_u Dest, Src
```

Example

```
lb_u 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 0 extended to a complete word.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

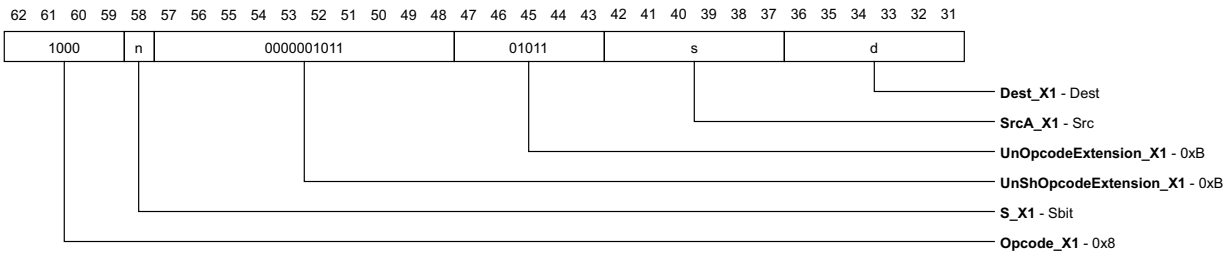

Figure 4-229: lb_u in X1 Bit Descriptions

Figure 4-230: lb_u in Y2 Bit Descriptions

lbadd: Load Byte and Add

Syntax

```
lbadd Dest, SrcA, Imm8
```

Example

```
lbadd 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.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

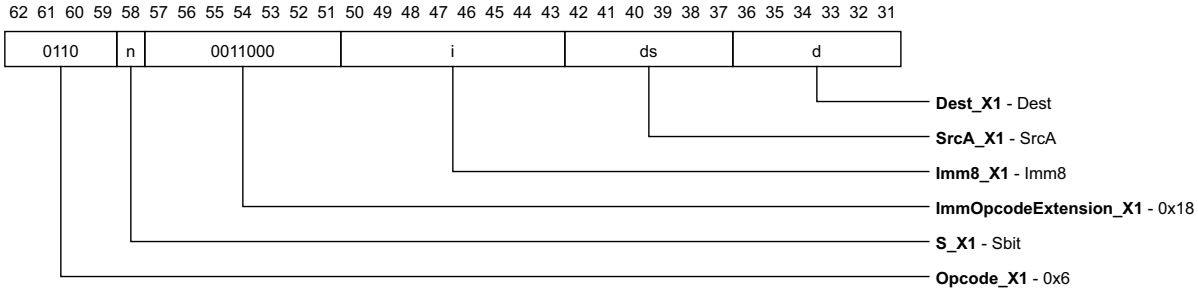

Figure 4-231: lbadd in X1 Bit Descriptions

lbadd_u: Load Byte Unsigned and Add

Syntax

```
lbadd_u Dest, SrcA, Imm8
```

Example

```
lbadd_u 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 0-extended to a complete word. Add the signed immediate argument to the address register.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

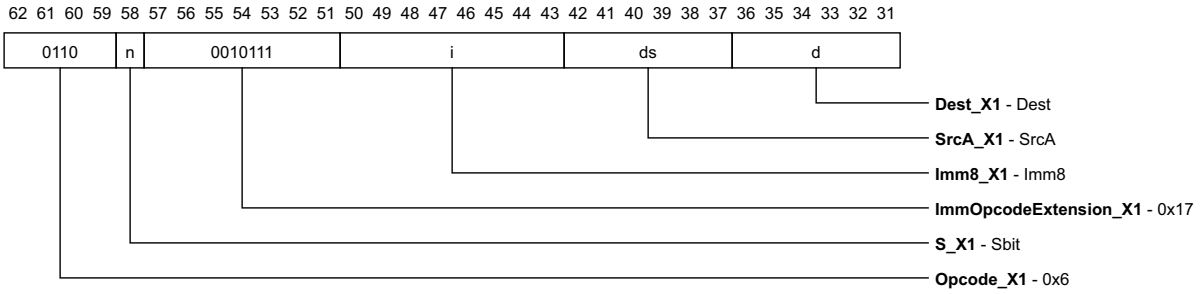

Figure 4-232: lbadd_u in X1 Bit Descriptions

lh: Load Half Word

Syntax

```
lh Dest, Src
```

Example

```
lh r5, r6
```

Description

Load a half word 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 half word aligned loads. 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

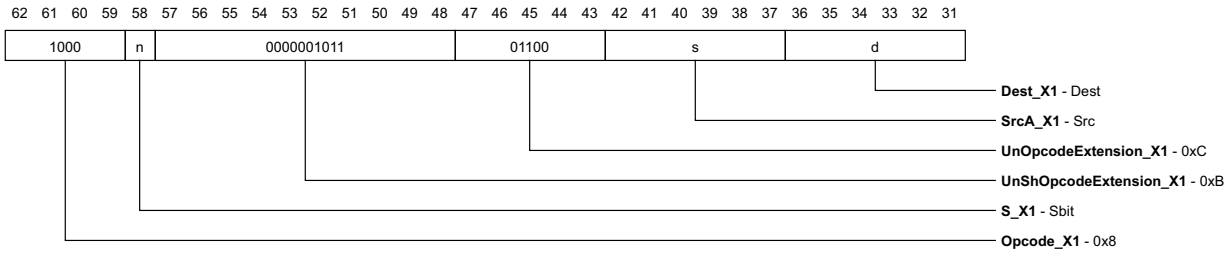

Figure 4-233: lh in X1 Bit Descriptions

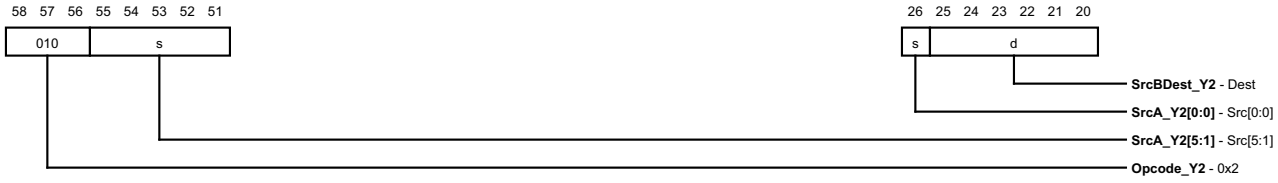

Figure 4-234: lh in Y2 Bit Descriptions

lh_u: Load Half Word Unsigned

Syntax

```
lh_u Dest, Src
```

Example

```
lh_u r5, r6
```

Description

Load a half word 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 half word aligned loads. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is 0 extended to a complete word.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			X

Encoding

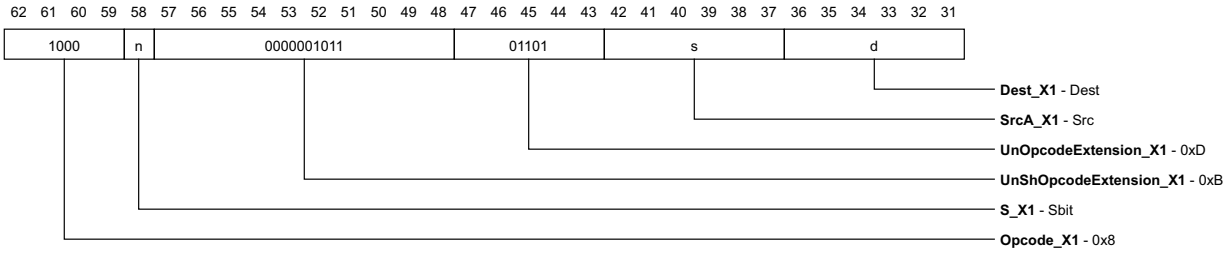

Figure 4-235: lh_u in X1 Bit Descriptions

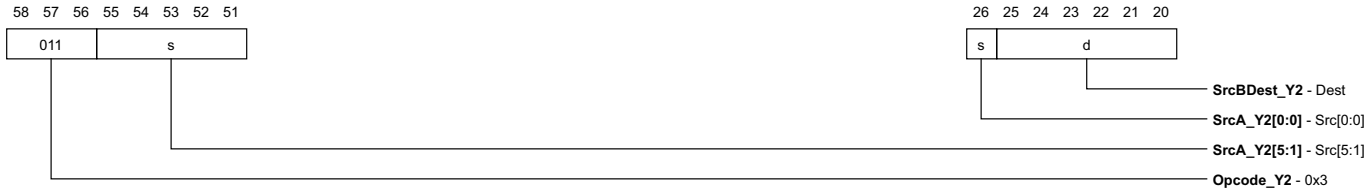

Figure 4-236: lh_u in Y2 Bit Descriptions

lhadd: Load Half Word and Add

Syntax

```
lhadd Dest, SrcA, Imm8
```

Example

```
lhadd r5, r6, 5
```

Description

Load a half word 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 half word aligned loads. 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.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

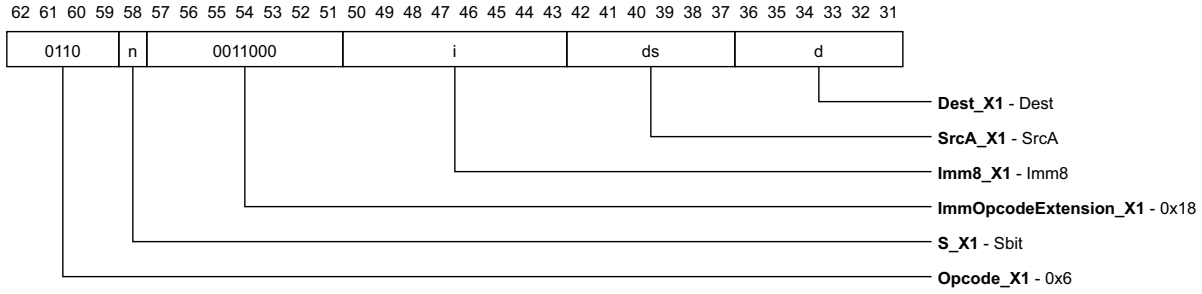

Figure 4-237: lhadd in X1 Bit Descriptions

lhadd_u: Load Half Word Unsigned and Add

Syntax

```
lhadd_u Dest, SrcA, Imm8
```

Example

```
lhadd_u r5, r6, 5
```

Description

Load a half word 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 half word aligned loads. Unaligned memory access causes an Unaligned Data Reference interrupt. The value read from memory is 0 extended to a complete word. Add the signed immediate argument to the address register.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

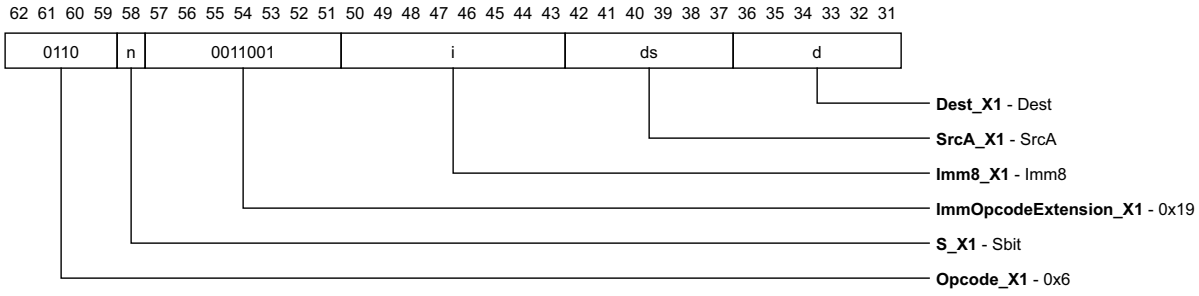

Figure 4-238: lhadd_u in X1 Bit Descriptions

lw: Load Word

Syntax

```
lw Dest, Src
```

Example

```
lw r5, r6
```

Description

Load a word 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 word aligned loads. 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

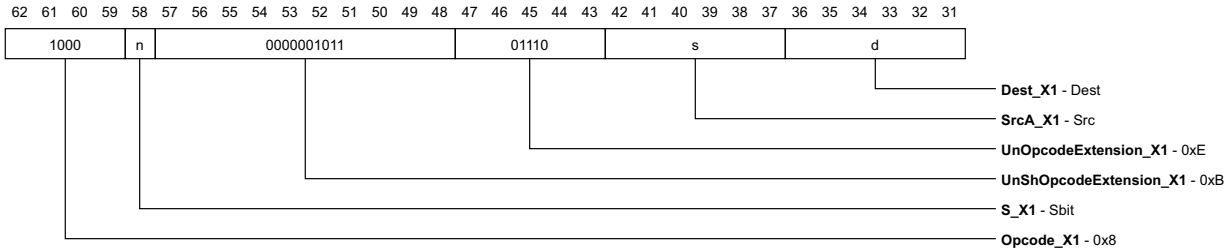

Figure 4-239: lw in X1 Bit Descriptions

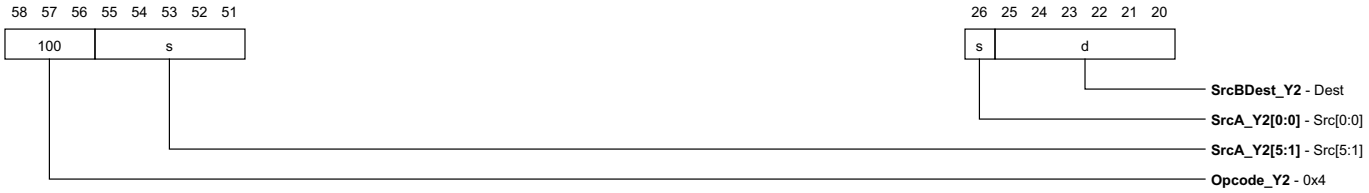

Figure 4-240: lw in Y2 Bit Descriptions

lw_na: Load Word No Alignment Trap

Syntax

```
lw_na Dest, Src
```

Example

```
lw_na r5, r6
```

Description

Load a word from memory into the destination register. The address of the value to be loaded is read from the source operand and the bottom two bits are set to 0. No Unaligned Data Reference interrupts are caused by this instruction.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

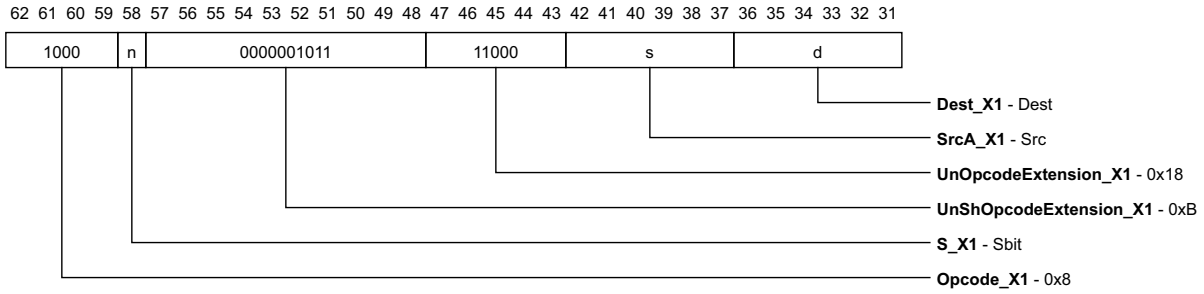

Figure 4-241: lw_na in X1 Bit Descriptions

lwadd: Load Word and Add

Syntax

```
lwadd Dest, SrcA, Imm8
```

Example

```
lwadd r5, r6, 5
```

Description

Load a word 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 word aligned loads. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

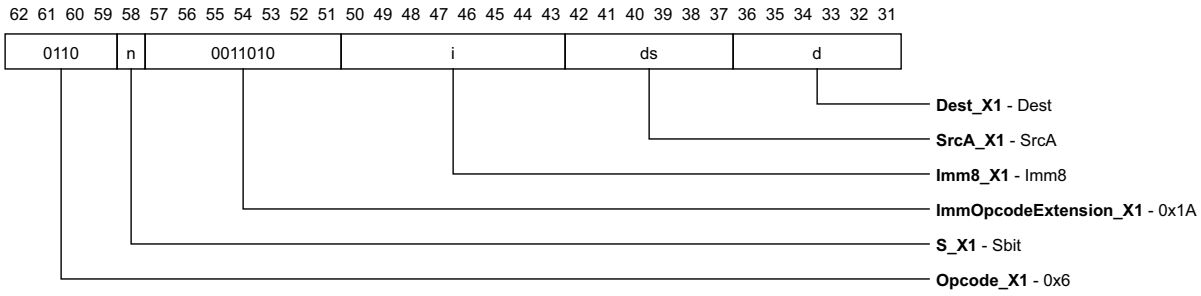

Figure 4-242: lbadd in X1 Bit Descriptions

lwadd_na: Load Word No Alignment Trap and Add

Syntax

```
lwadd_na Dest, SrcA, Imm8
```

Example

```
lwadd_na r5, r6, 5
```

Description

Load a word from memory into the destination register. The address of the value to be loaded is read from the source operand and the bottom two bits are set to 0. No Unaligned Data Reference interrupts are caused by this instruction. Add the signed immediate argument to the address register.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

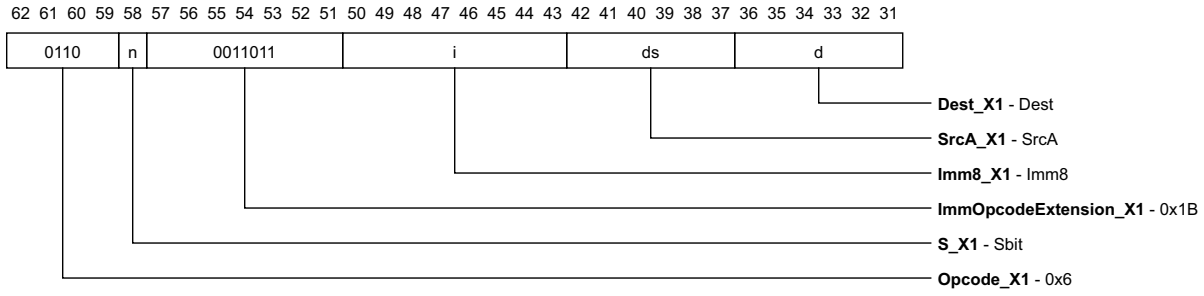

Figure 4-243: lwadd_na in X1 Bit Descriptions

sb: Store Byte

Syntax

```
sb SrcA, SrcB
```

Example

```
sb 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

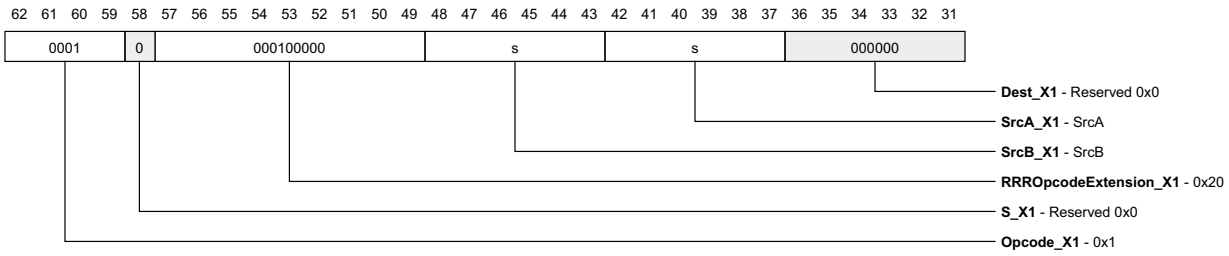

Figure 4-244: sb in X1 Bit Descriptions

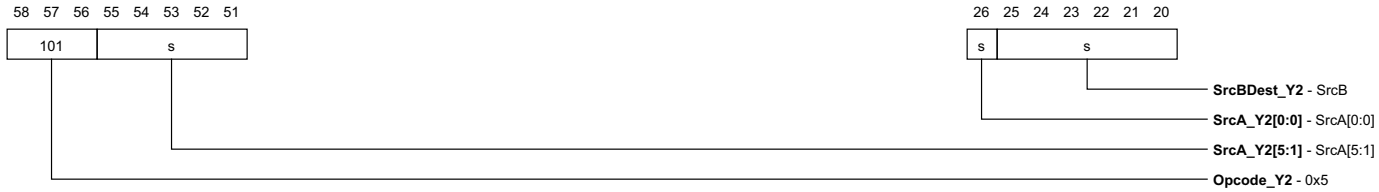

Figure 4-245: sb in Y2 Bit Descriptions

sbadd: Store Byte and Add

Syntax

```
sbadd SrcA, SrcB, Imm8
```

Example

```
sbadd 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.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

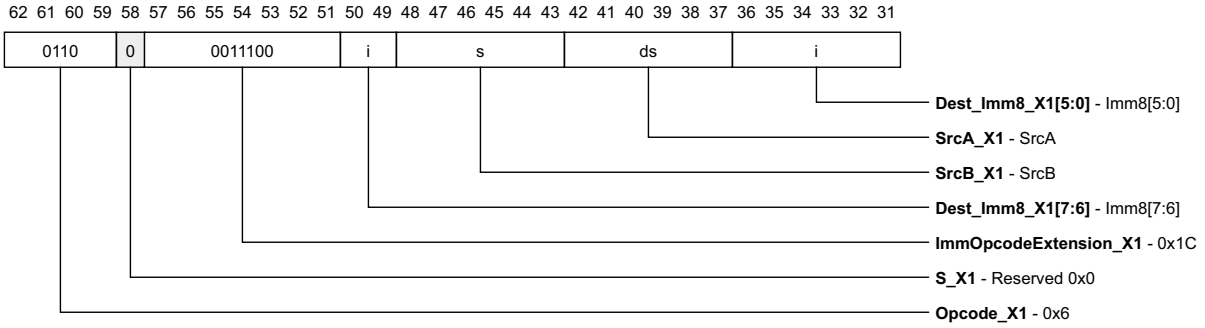

Figure 4-246: sbadd in X1 Bit Descriptions

sh: Store Half Word

Syntax

```
sh SrcA, SrcB
```

Example

```
sh r5, r6
```

Description

Store a half word from the second source register into memory at the address held in the first source register. This store only operates for half word aligned stores. 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

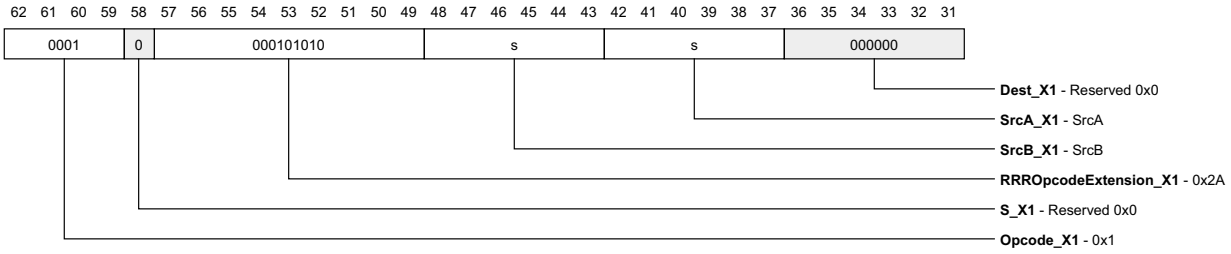

Figure 4-247: sh in X1 Bit Descriptions

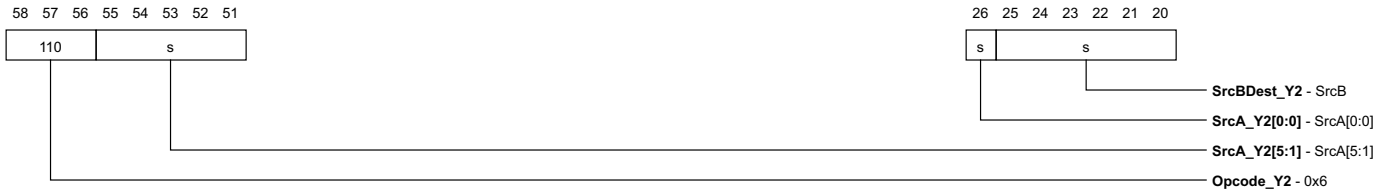

Figure 4-248: sh in Y2 Bit Descriptions

shadd: Store Half Word and Add

Syntax

```
shadd SrcA, SrcB, Imm8
```

Example

```
shadd r5, r6, 5
```

Description

Store a half word from the second source register into memory at the address held in the first source register. This store only operates for half word aligned stores. Unaligned memory access causes an Unaligned Data Reference interrupt. Add the signed immediate argument to the address register.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

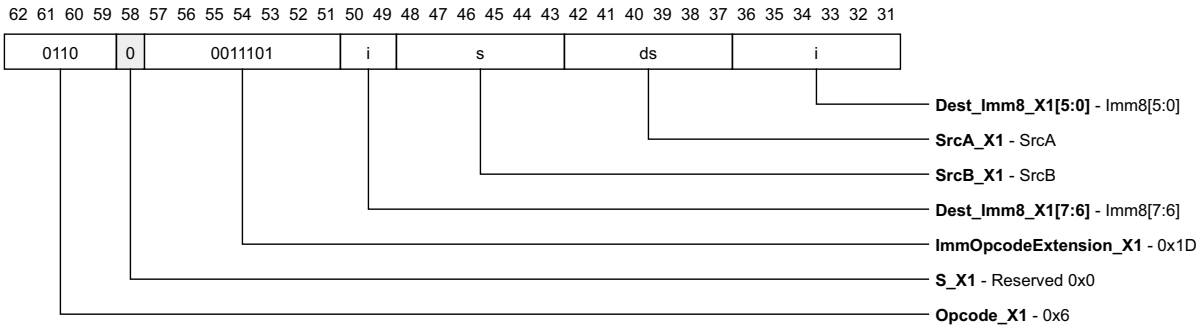

Figure 4-249: shadd in X1 Bit Descriptions

sw: Store Word

Syntax

```
sw SrcA, SrcB
```

Example

```
sw r5, r6
```

Description

Store a word from the second source register into memory at the address held in the first source register. This store only operates for word aligned stores. 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

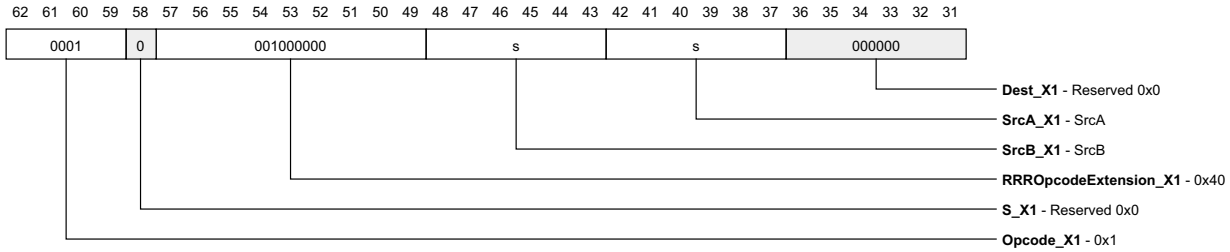

Figure 4-250: sw in X1 Bit Descriptions

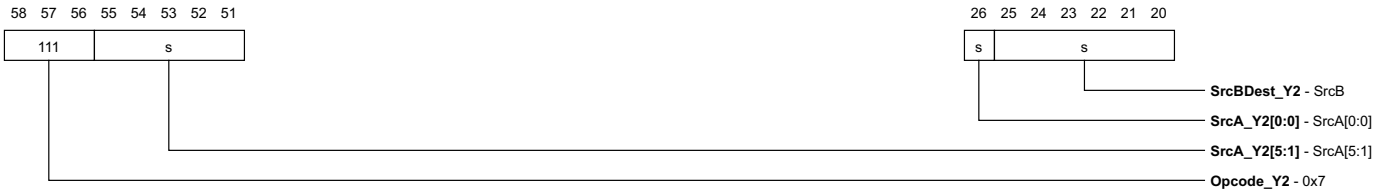

Figure 4-251: sw in Y2 Bit Descriptions

swadd: Store Word and Add

Syntax

```
swadd SrcA, SrcB, Imm8
```

Example

```
swadd r5, r6, 5
```

Description

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

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

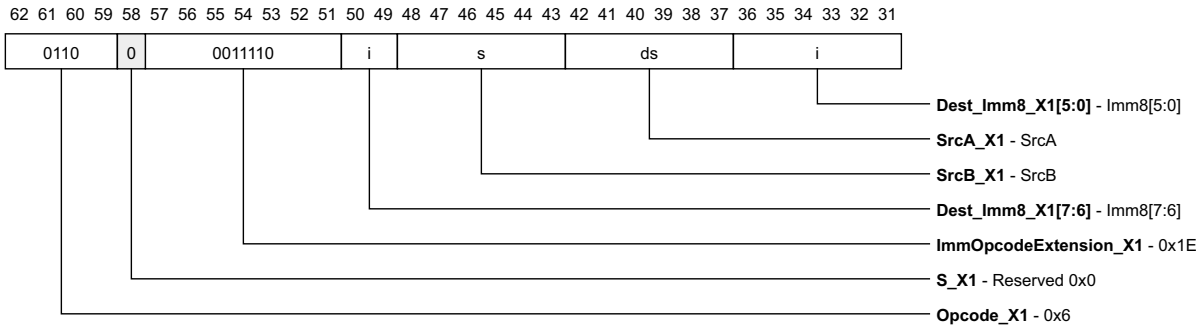

Figure 4-252: swadd in X1 Bit Descriptions

tns: Test and Set Word

Syntax

```
tns Dest, Src
```

Example

```
tns r5, r6
```

Description

Load a word from memory into the destination register and atomically write the value one (1) into the addressed memory location. The address of the value to be loaded then written to is read from the source operand. This instruction only operates for word aligned addresses. Unaligned memory access causes an Unaligned Data Reference interrupt.

Functional Description

```
rf[Dest] = memoryReadWord(rf[Src]);
memoryWriteWord(rf[Src] & WORD_ADDR_MASK, 0x00000001);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

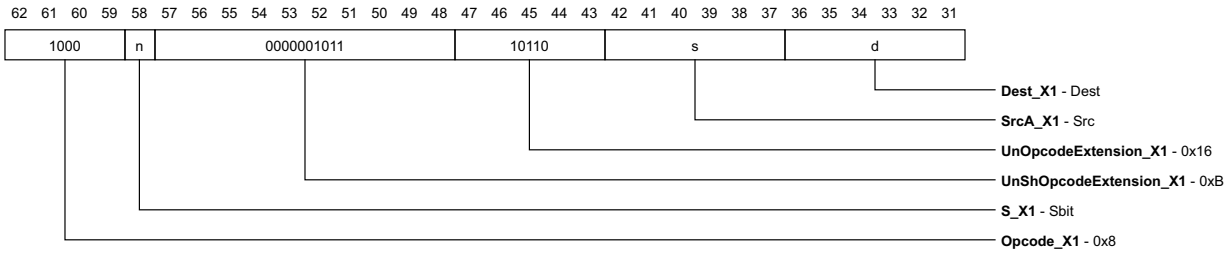

Figure 4-253: tns in X1 Bit Descriptions

4.1.10 Memory Maintenance Instructions

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

- dtlbpr: Data TLB Probe
- finv: Flush and Invalidate Cache Line
- flush: Flush Cache Line
- inv: Invalidate Cache Line
- mf: Memory Fence
- wh64: Write Hint 64 Bytes

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 in to SPR `DTLB_MATCH_0`. 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

Encoding

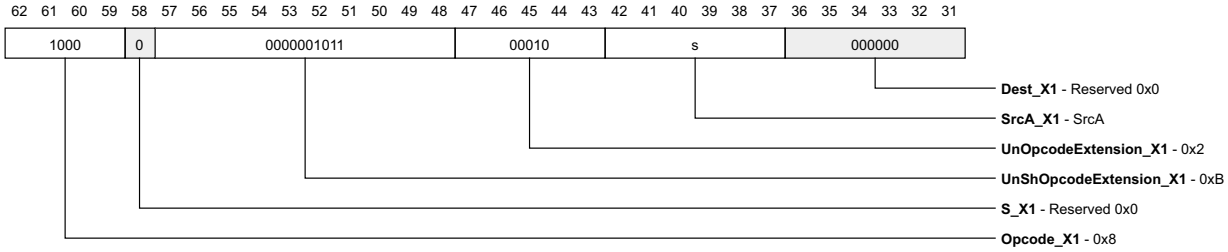

Figure 4-254: dtlbpr in X1 Bit Descriptions

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 on the cache contents. 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

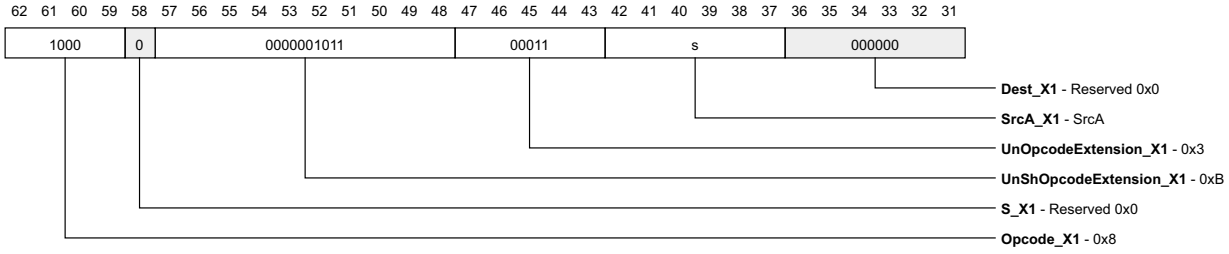

Figure 4-255: finv in X1 Bit Descriptions

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

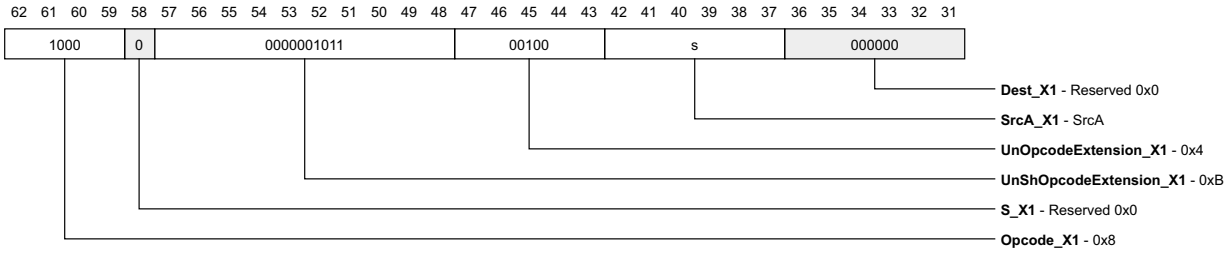

Figure 4-256: flush in X1 Bit Descriptions

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 on the cache contents. 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

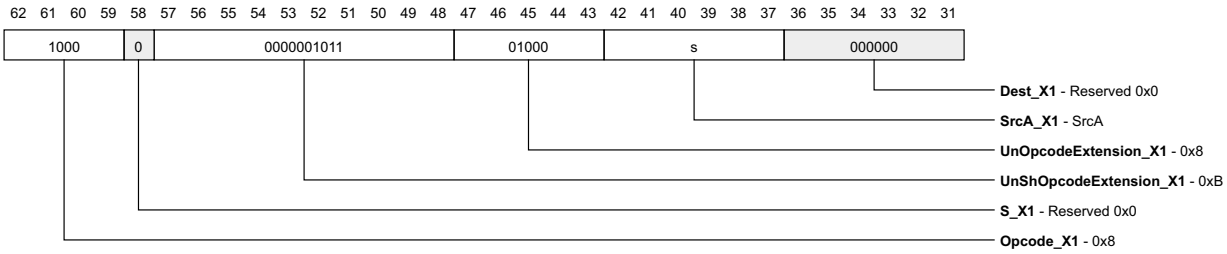

Figure 4-257: inv in X1 Bit Descriptions

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 order that is established depends on the page attributes of the pages that the memory operations are targeting. For more information refer to Memory and Cache Architecture.

Functional Description

```
memoryFence();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

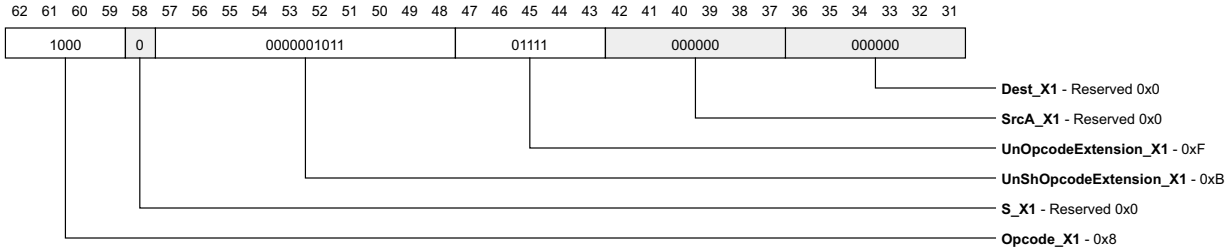

Figure 4-258: mf in X1 Bit Descriptions

wh64: Write Hint 64 Bytes

Syntax

```
wh64 Src
```

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.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

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

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

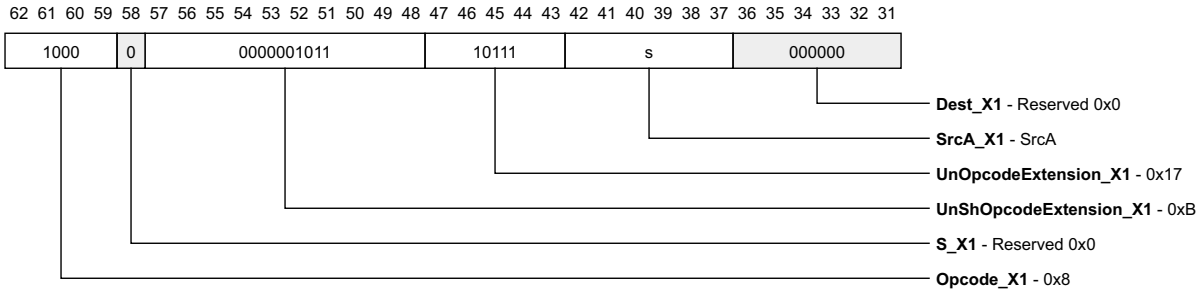

Figure 4-259: wh64 in X1 Bit Descriptions

4.1.11 Multiply Instructions

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

- mulhh_ss: Multiply High Signed High Signed Half Word
- mulhh_su: Multiply High Signed High Unsigned Half Word
- mulhh_uu: Multiply High Unsigned High Unsigned Half Word
- mulhha_ss: Multiply Accumulate High Signed High Signed Half Word
- mulhha_su: Multiply Accumulate High Signed High Unsigned Half Word
- mulhha_uu: Multiply Accumulate High Unsigned High Unsigned Half Word
- mulhhsa_uu: Multiply Shift Accumulate High Unsigned High Unsigned Half Word
- mulhl_ss: Multiply High Signed Low Signed Half Word
- mulhl_su: Multiply High Signed Low Unsigned Half Word
- mulhl_us: Multiply High Unsigned Low Signed Half Word
- mulhl_uu: Multiply High Unsigned Low Unsigned Half Word
- mulhla_ss: Multiply Accumulate High Signed Low Signed Half Word
- mulhla_su: Multiply Accumulate High Signed Low Unsigned Half Word
- mulhla_us: Multiply Accumulate High Unsigned Low Signed Half Word
- mulhla_uu: Multiply Accumulate High Unsigned Low Unsigned Half Word
- mulhlsa_uu: Multiply Shift Accumulate High Unsigned Low Unsigned Half Word
- mulll_ss: Multiply Low Signed Low Signed Half Word
- mulll_su: Multiply Low Signed Low Unsigned Half Word
- mulll_uu: Multiply Low Unsigned Low Unsigned Half Word
- mullla_ss: Multiply Accumulate Low Signed Low Signed Half Word
- mullla_su: Multiply Accumulate Low Signed Low Unsigned Half Word
- mullla_uu: Multiply Accumulate Low Unsigned Low Unsigned Half Word
- mulllsa_uu: Multiply Shift Accumulate Low Unsigned Low Unsigned Half Word

mulhh_ss: Multiply High Signed High Signed Half Word

Syntax

```
mulhh_ss Dest, SrcA, SrcB
```

Example

```
mulhh_ss r5, r6, r7
```

Description

Multiply the high half word of the first operand by the high half word of the second operand. The result returned is a full word in length. The input operands are interpreted as signed half words.

Functional Description

```
rf[Dest] =
    ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
    ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

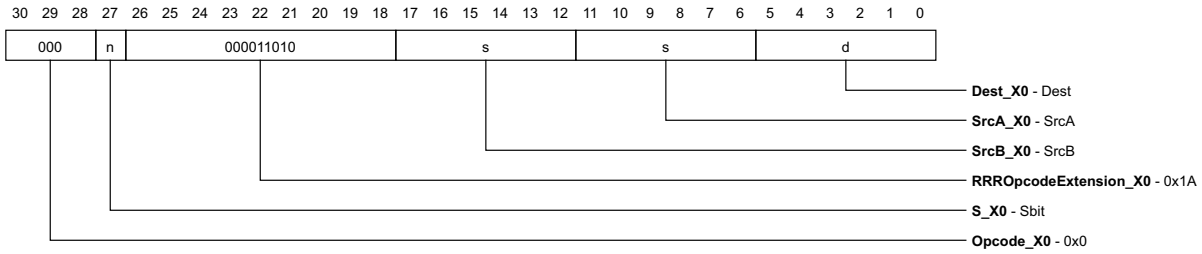

Figure 4-260: mulhh_ss in X0 Bit Descriptions

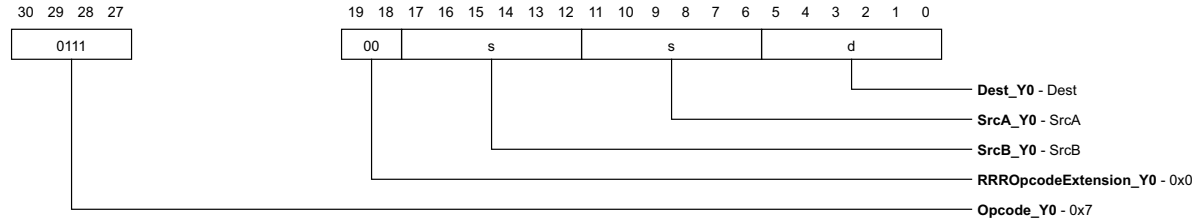

Figure 4-261: mulhh_ss in Y0 Bit Descriptions

mulhh_su: Multiply High Signed High Unsigned Half Word

Syntax

```
mulhh_su Dest, SrcA, SrcB
```

Example

```
mulhh_su r5, r6, r7
```

Description

Multiply the high half word of the first operand by the high half word of the second operand. The result returned is a full word in length. The first input operand is interpreted as a signed half word and the second input operand is interpreted as an unsigned half word.

Functional Description

```
rf[Dest] =
    ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
    ((UnsignedMachineWord) getHighHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

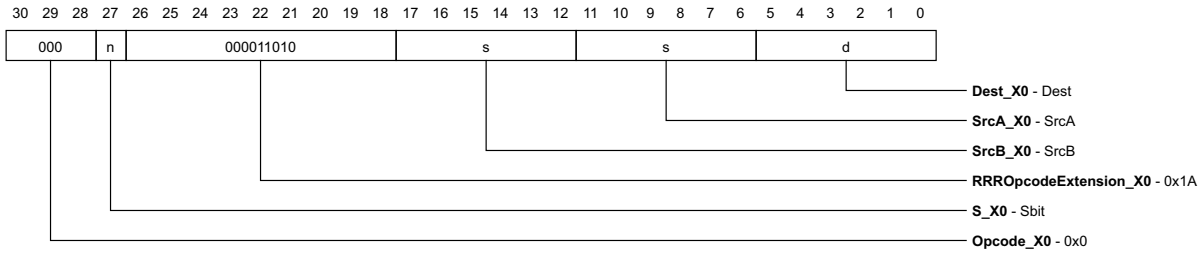

Figure 4-262: mulhh_ss in X0 Bit Descriptions

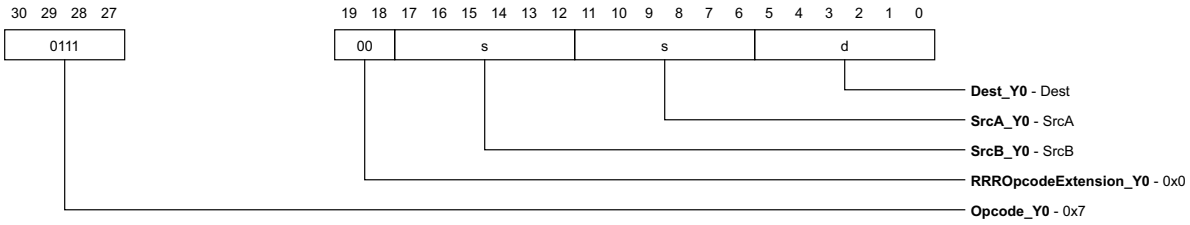

Figure 4-263: mulhh_ss in Y0 Bit Descriptions

mulhh_uu: Multiply High Unsigned High Unsigned Half Word

Syntax

```
mulhh_uu Dest, SrcA, SrcB
```

Example

```
mulhh_uu r5, r6, r7
```

Description

Multiply the high half word of the first operand by the high half word of the second operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
    ((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
    ((UnsignedMachineWord) getHighHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

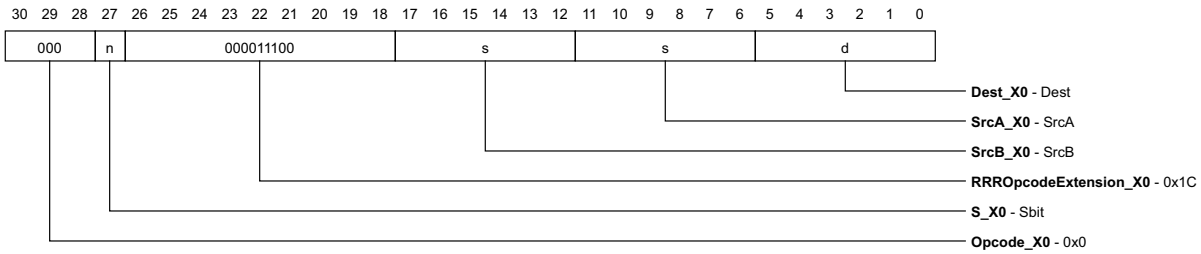

Figure 4-264: mulhh_uu in X0 Bit Descriptions

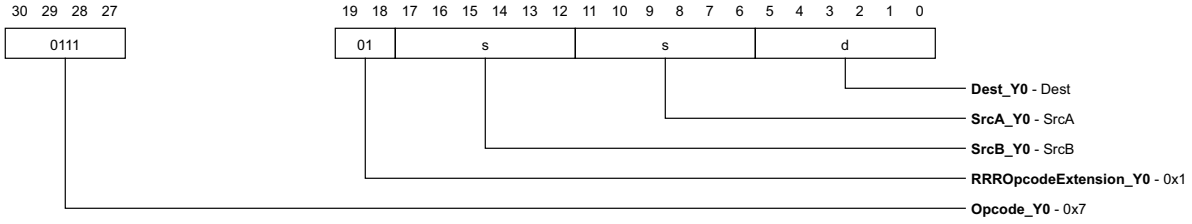

Figure 4-265: mulhh_uu in Y0 Bit Descriptions

mulhha_ss: Multiply Accumulate High Signed High Signed Half Word

Syntax

```
mulhha_ss Dest, SrcA, SrcB
```

Example

```
mulhha_ss r5, r6, r7
```

Description

Multiply the high half word of the first operand by the high half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as signed half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
   ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

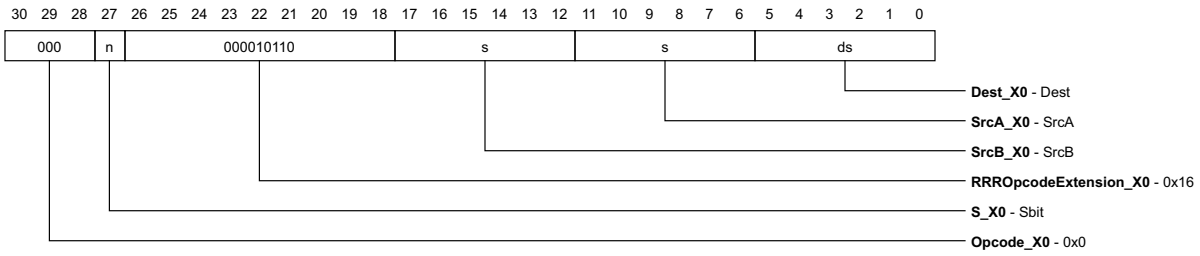

Figure 4-266: mulhha_ss in X0 Bit Descriptions

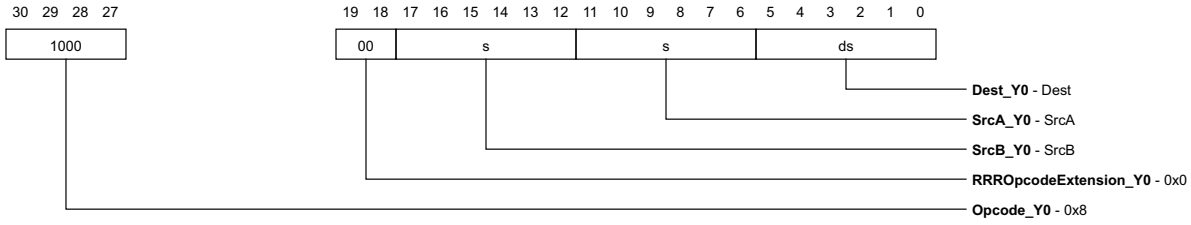

Figure 4-267: mulhha_ss in Y0 Bit Descriptions

mulhha_su: Multiply Accumulate High Signed High Unsigned Half Word

Syntax

```
mulhha_su Dest, SrcA, SrcB
```

Example

```
mulhha_su r5, r6, r7
```

Description

Multiply the high half word of the first operand by the high half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The first input operand is interpreted as a signed half word and the second input operand is interpreted as an unsigned half word.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
   ((UnsignedMachineWord) getHighHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

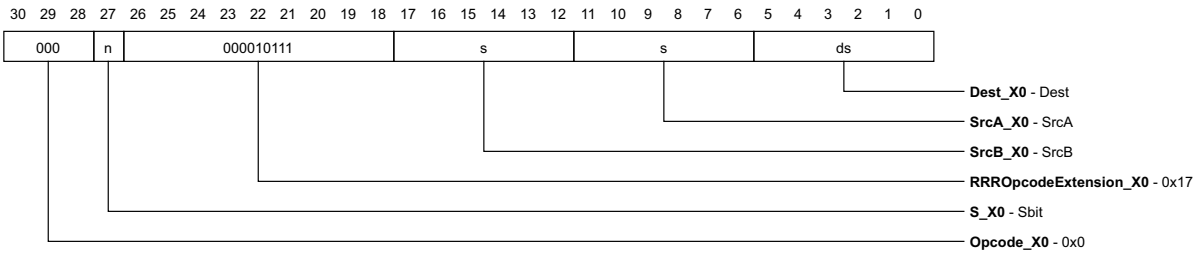

Figure 4-268: mulhha_su in X0 Bit Descriptions

mulhha_uu: Multiply Accumulate High Unsigned High Unsigned Half Word

Syntax

```
mulhha_uu Dest, SrcA, SrcB
```

Example

```
mulhha_uu r5, r6, r7
```

Description

Multiply the high half word of the first operand by the high half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
   ((UnsignedMachineWord) getHighHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

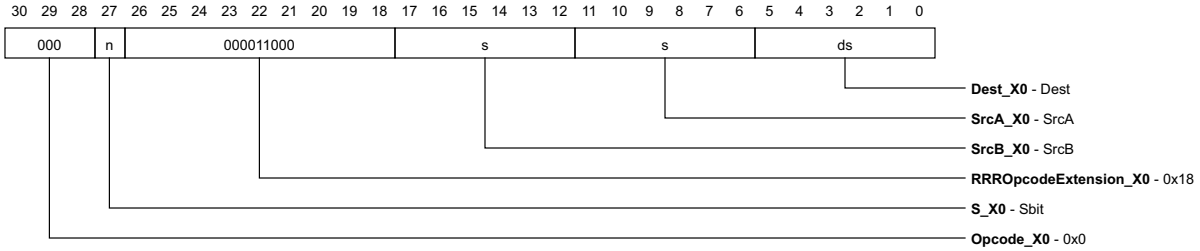

Figure 4-269: mulhha_uu in X0 Bit Descriptions

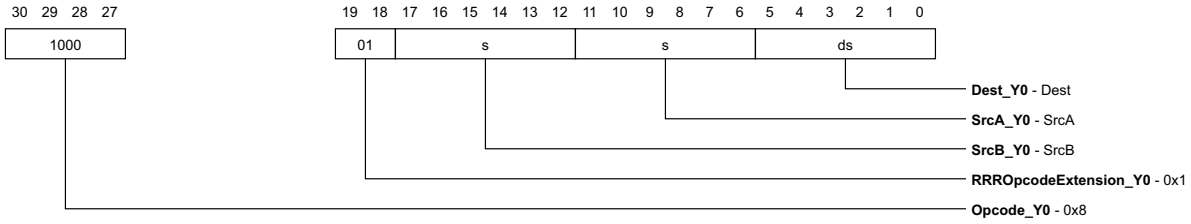

Figure 4-270: mulhha_uu in Y0 Bit Descriptions

mulhhsa_uu: Multiply Shift Accumulate High Unsigned High Unsigned Half Word

Syntax

```
mulhhsa_uu Dest, SrcA, SrcB
```

Example

```
mulhhsa_uu r5, r6, r7
```

Description

Multiply the high half word of the first operand by the high half word of the second operand, shift the multiply left by 16, and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
     ((UnsignedMachineWord) getHighHalfWord(rf[SrcB]))) << 16);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

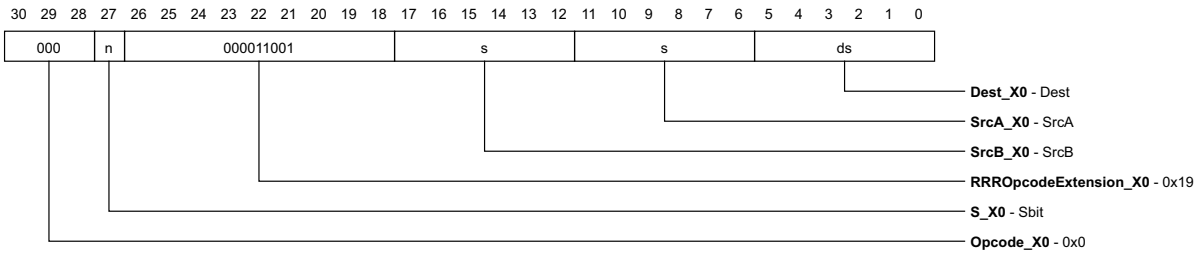

Figure 4-271: mulhhsa_uu in X0 Bit Descriptions

mulhl_ss: Multiply High Signed Low Signed Half Word

Syntax

```
mulhl_ss Dest, SrcA, SrcB
```

Example

```
mulhl_ss r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand. The result returned is a full word in length. The input operands are interpreted as signed half words.

Functional Description

```
rf[Dest] =
    ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
    ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

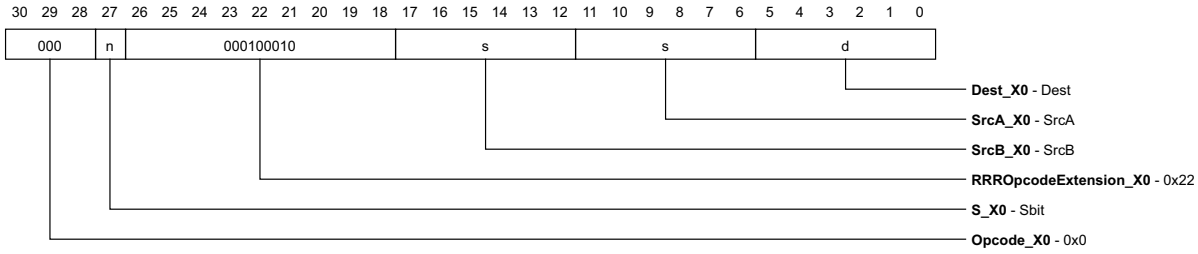

Figure 4-272: mulhl_ss in X0 Bit Descriptions

mulhl_su: Multiply High Signed Low Unsigned Half Word

Syntax

```
mulhl_su Dest, SrcA, SrcB
```

Example

```
mulhl_su r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand. The result returned is a full word in length. The first input operand is interpreted as a signed half word and the second input operand is interpreted as an unsigned half word.

Functional Description

```
rf[Dest] =
    ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
    ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

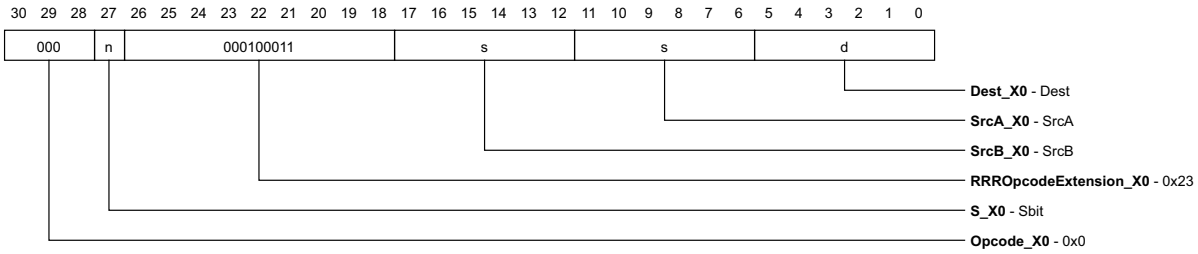

Figure 4-273: mulhl_su in X0 Bit Descriptions

mulhl_us: Multiply High Unsigned Low Signed Half Word

Syntax

```
mulhl_us Dest, SrcA, SrcB
```

Example

```
mulhl_us r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand. The result returned is a full word in length. The first input operand is interpreted as an unsigned half word and the second input operand is interpreted as a signed half word.

Functional Description

```
rf[Dest] =
    ((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
    ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

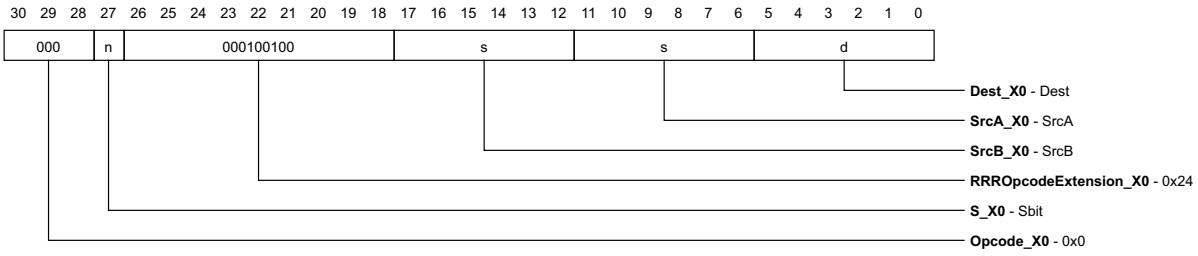

Figure 4-274: mulhl_us in X0 Bit Descriptions

mulhl_uu: Multiply High Unsigned Low Unsigned Half Word

Syntax

```
mulhl_uu Dest, SrcA, SrcB
```

Example

```
mulhl_uu r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
    ((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
    ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

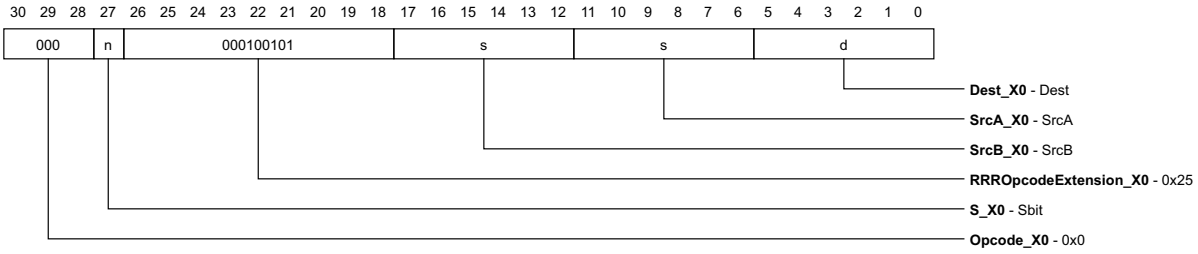

Figure 4-275: mulhl_uu in X0 Bit Descriptions

mulhla_ss: Multiply Accumulate High Signed Low Signed Half Word

Syntax

```
mulhla_ss Dest, SrcA, SrcB
```

Example

```
mulhla_ss r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as signed half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
   ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

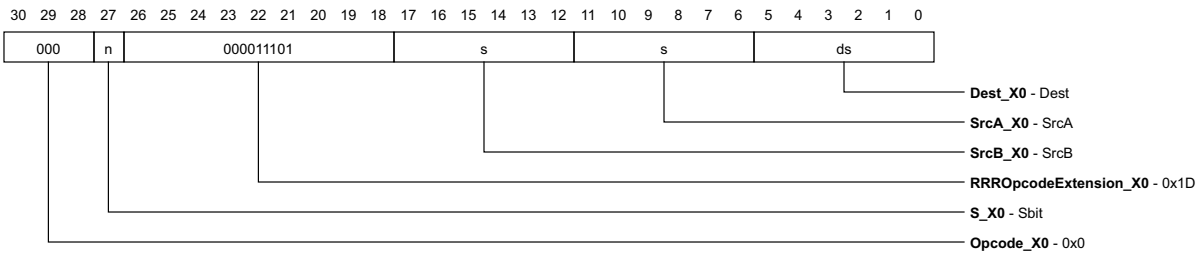

Figure 4-276: mulhla_ss in X0 Bit Descriptions

mulhla_su: Multiply Accumulate High Signed Low Unsigned Half Word

Syntax

```
mulhla_su Dest, SrcA, SrcB
```

Example

```
mulhla_su r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The first input operand is interpreted as a signed half word and the second input operand is interpreted as an unsigned half word.

Functional Description

```
rf[Dest] =
    rf[Dest] +
    ((SignedMachineWord) signExtend16(getHighHalfWord(rf[SrcA]))) *
    ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

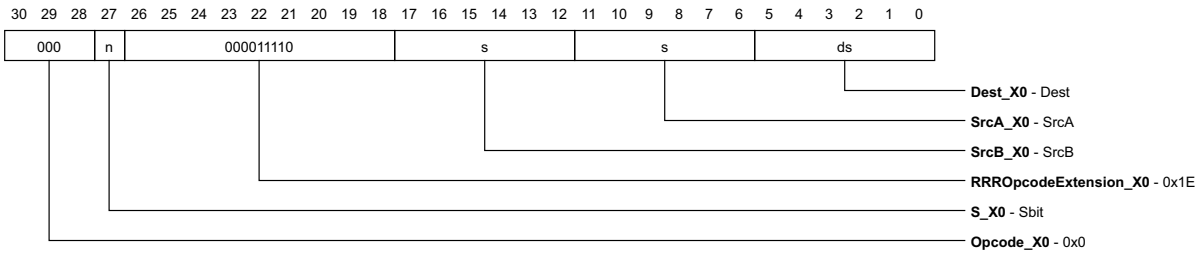

Figure 4-277: mulhla_su in X0 Bit Descriptions

mulhla_us: Multiply Accumulate High Unsigned Low Signed Half Word

Syntax

```
mulhla_us Dest, SrcA, SrcB
```

Example

```
mulhla_us r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The first input operand is interpreted as an unsigned half word and the second input operand is interpreted as a signed half word.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
   ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

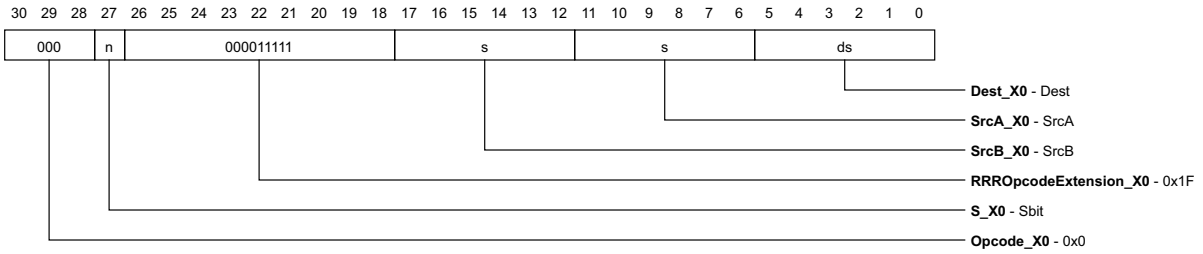

Figure 4-278: mulhla_us in X0 Bit Descriptions

mulhla_uu: Multiply Accumulate High Unsigned Low Unsigned Half Word

Syntax

```
mulhla_uu Dest, SrcA, SrcB
```

Example

```
mulhla_uu r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
   ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

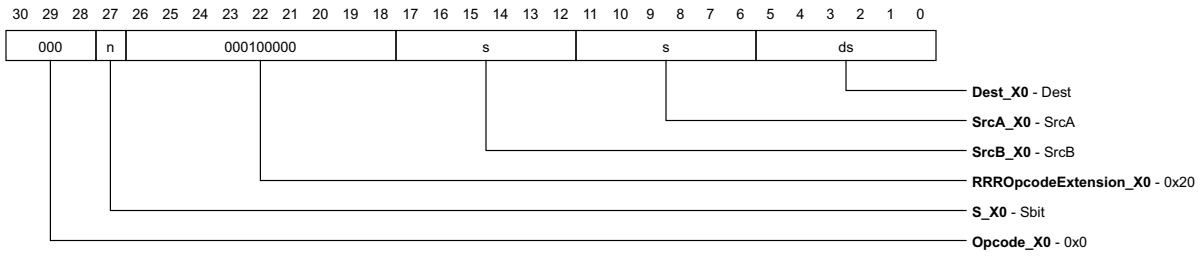

Figure 4-279: mulhla_uu in X0 Bit Descriptions

mulhlsa_uu: Multiply Shift Accumulate High Unsigned Low Unsigned Half Word

Syntax

```
mulhlsa_uu Dest, SrcA, SrcB
```

Example

```
mulhlsa_uu r5, r6, r7
```

Description

Multiply the high half word of the first operand by the low half word of the second operand, shift the multiply left by 16, and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((((UnsignedMachineWord) getHighHalfWord(rf[SrcA])) *
     ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]))) << 16);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

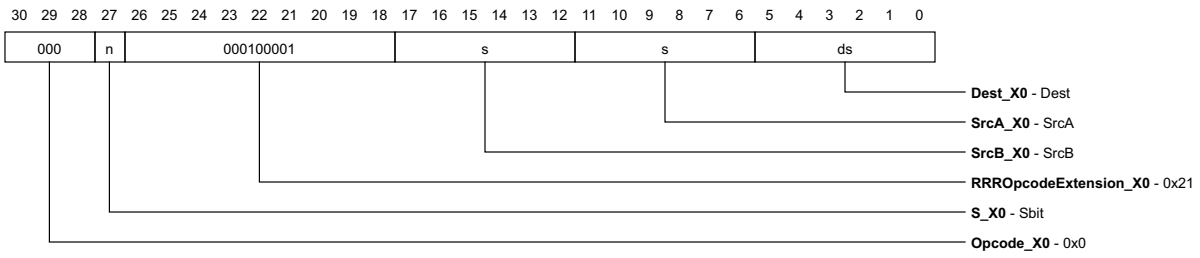

Figure 4-280: mulhlsa_uu in X0 Bit Descriptions

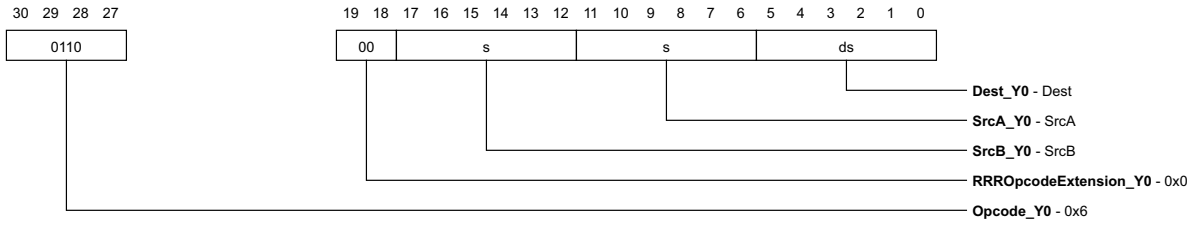

Figure 4-281: mulhlsa_uu in Y0 Bit Descriptions

mulll_ss: Multiply Low Signed Low Signed Half Word

Syntax

```
mulll_ss Dest, SrcA, SrcB
```

Example

```
mulll_ss r5, r6, r7
```

Description

Multiply the low half word of the first operand by the low half word of the second operand. The result returned is a full word in length. The input operands are interpreted as signed half words.

Functional Description

```
rf[Dest] =
    ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcA]))) *
    ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

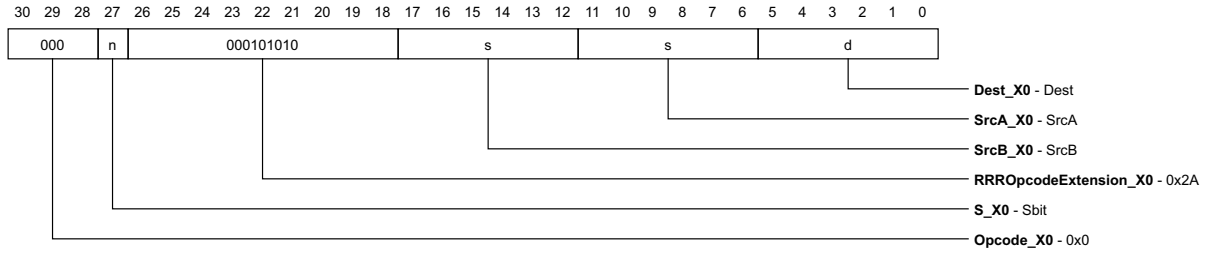

Figure 4-282: mulll_ss in X0 Bit Descriptions

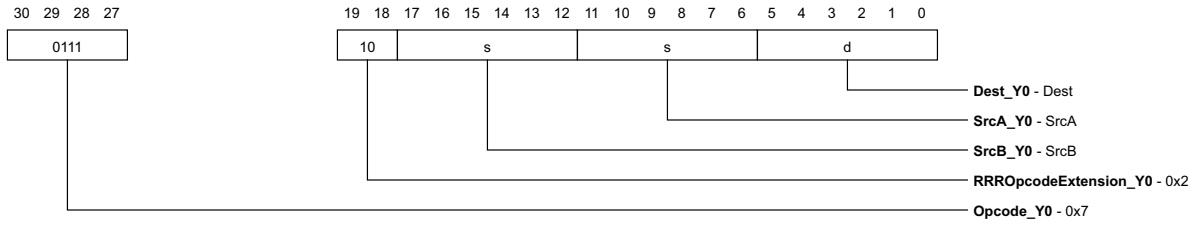

Figure 4-283: mulll_ss in Y0 Bit Descriptions

mulll_su: Multiply Low Signed Low Unsigned Half Word

Syntax

```
mulll_su Dest, SrcA, SrcB
```

Example

```
mulll_su r5, r6, r7
```

Description

Multiply the low half word of the first operand by the low half word of the second operand. The result returned is a full word in length. The first input operand is interpreted as a signed half word and the second input operand is interpreted as an unsigned half word.

Functional Description

```
rf[Dest] =
    ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcA]))) *
    ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

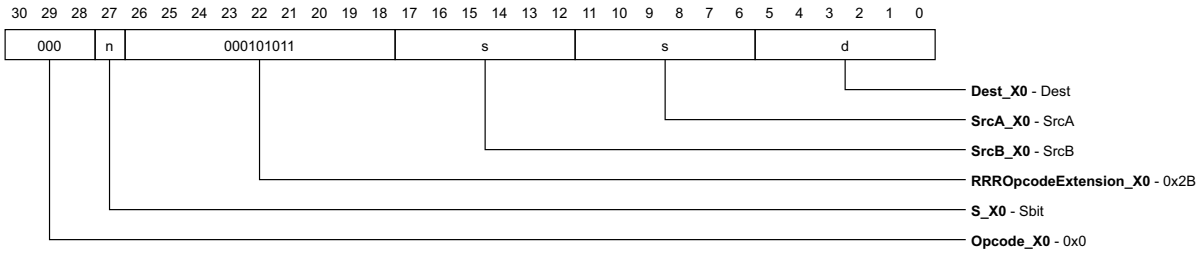

Figure 4-284: mulll_su in X0 Bit Descriptions

mulll_uu: Multiply Low Unsigned Low Unsigned Half Word

Syntax

```
mulll_uu Dest, SrcA, SrcB
```

Example

```
mulll_uu r5, r6, r7
```

Description

Multiply the low half word of the first operand by the low half word of the second operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
    ((UnsignedMachineWord) getLowHalfWord(rf[SrcA])) *
    ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

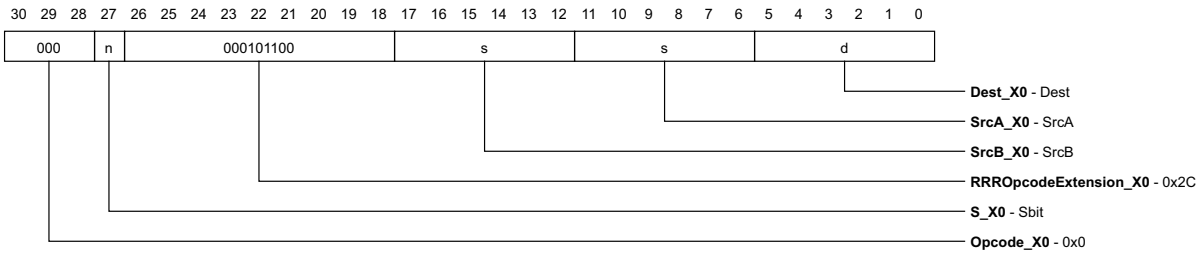

Figure 4-285: mulll_uu in X0 Bit Descriptions

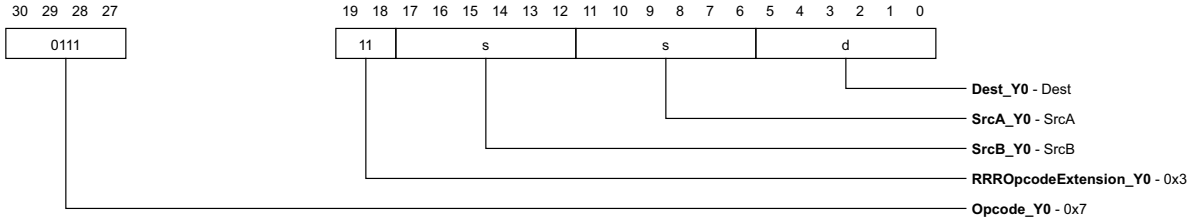

Figure 4-286: mulll_uu in Y0 Bit Descriptions

mullla_ss: Multiply Accumulate Low Signed Low Signed Half Word

Syntax

```
mullla_ss Dest, SrcA, SrcB
```

Example

```
mullla_ss r5, r6, r7
```

Description

Multiply the low half word of the first operand by the low half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as signed half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcA]))) *
   ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcB])));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

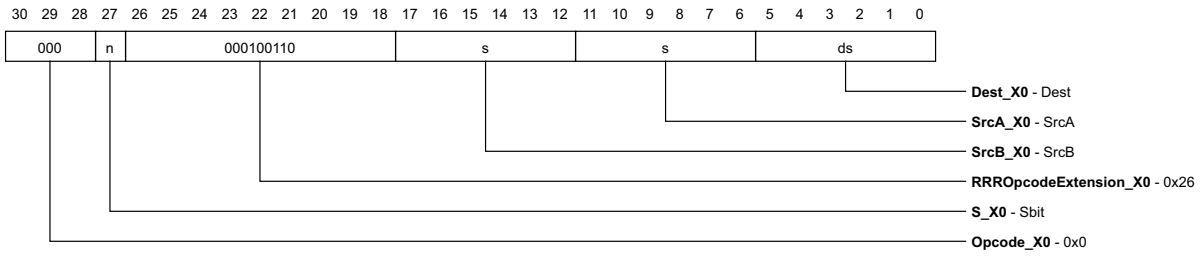

Figure 4-287: mullla_ss in X0 Bit Descriptions

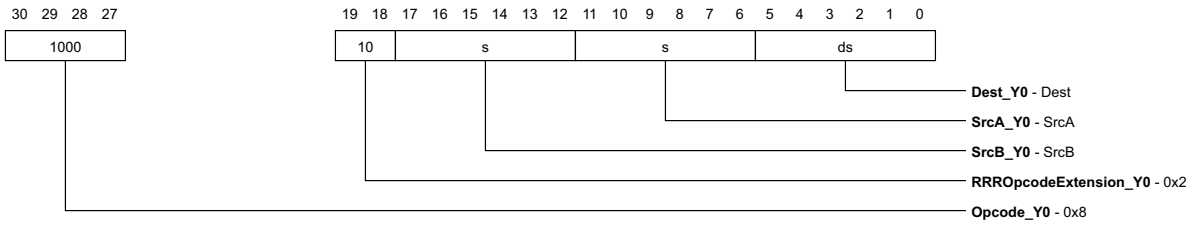

Figure 4-288: mullla_ss in Y0 Bit Descriptions

mullla_su: Multiply Accumulate Low Signed Low Unsigned Half Word

Syntax

```
mullla_su Dest, SrcA, SrcB
```

Example

```
mullla_su r5, r6, r7
```

Description

Multiply the low half word of the first operand by the low half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The first input operand is interpreted as a signed half word and the second input operand is interpreted as an unsigned half word.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((SignedMachineWord) signExtend16(getLowHalfWord(rf[SrcA]))) *
   ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

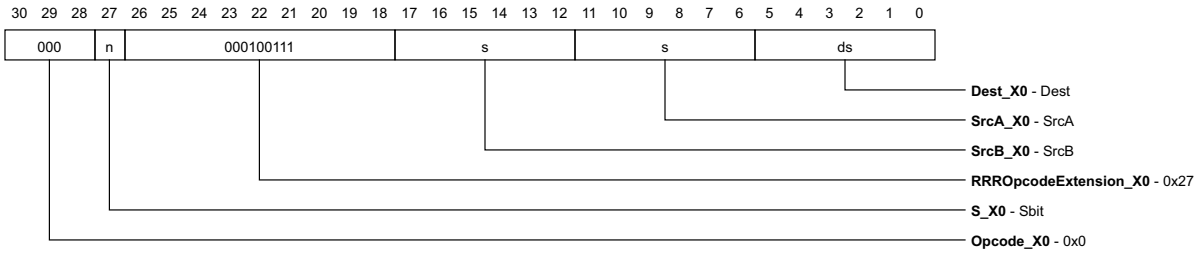

Figure 4-289: mullla_su in X0 Bit Descriptions

mullla_uu: Multiply Accumulate Low Unsigned Low Unsigned Half Word

Syntax

```
mullla_uu Dest, SrcA, SrcB
```

Example

```
mullla_uu r5, r6, r7
```

Description

Multiply the low half word of the first operand by the low half word of the second operand and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((UnsignedMachineWord) getLowHalfWord(rf[SrcA])) *
   ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]));
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X		X		

Encoding

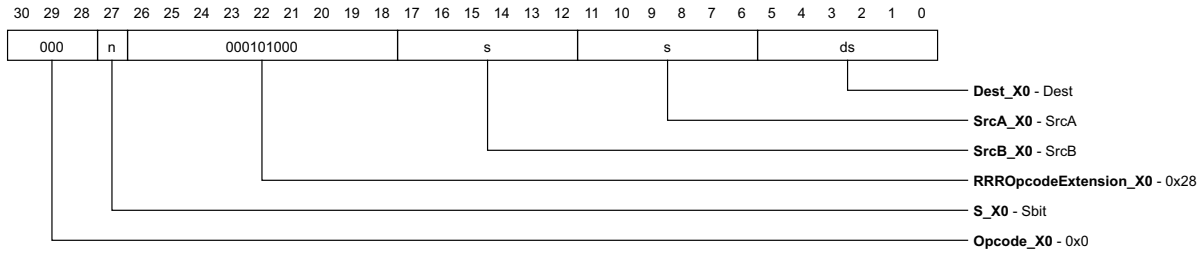

Figure 4-290: mullla_uu in X0 Bit Descriptions

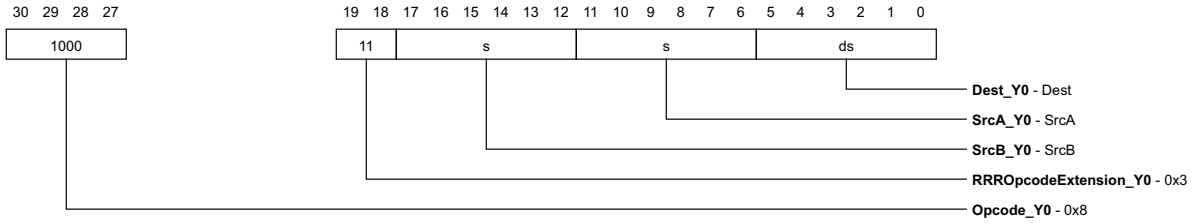

Figure 4-291: mullla_uu in Y0 Bit Descriptions

mulllsa_uu: Multiply Shift Accumulate Low Unsigned Low Unsigned Half Word

Syntax

```
mulllsa_uu Dest, SrcA, SrcB
```

Example

```
mulllsa_uu r5, r6, r7
```

Description

Multiply the low half word of the first operand by the low half word of the second operand, shift the multiply left 16, and accumulate the result into the destination operand. The result returned is a full word in length. The input operands are interpreted as unsigned half words.

Functional Description

```
rf[Dest] =
   rf[Dest] +
   ((((UnsignedMachineWord) getLowHalfWord(rf[SrcA])) *
     ((UnsignedMachineWord) getLowHalfWord(rf[SrcB]))) << 16);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

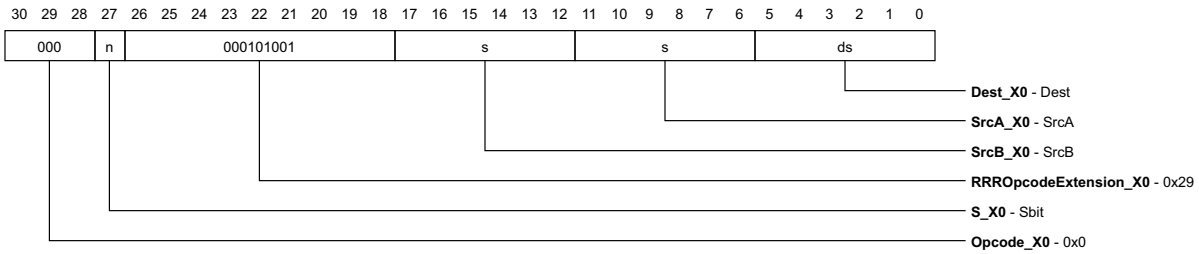

Figure 4-292: mulllsa_uu in X0 Bit Descriptions

4.1.12 NOP Instructions

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

- fnop: Filler No Operation
- nop: Architectural No Operation

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. `fnop` 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

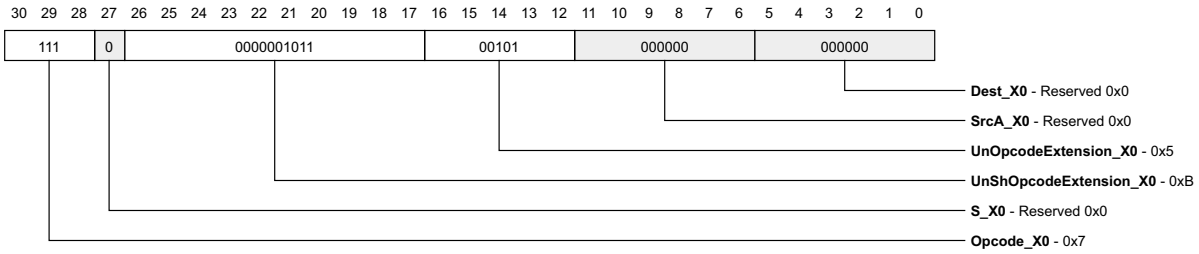

Figure 4-293: fnop in X0 Bit Descriptions

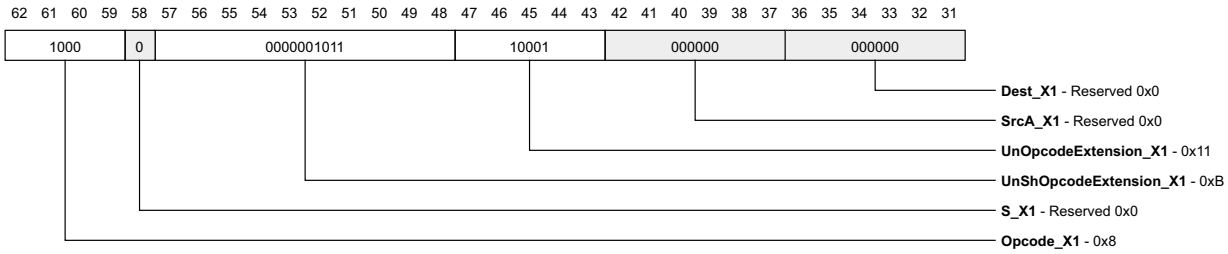

Figure 4-294: fnop in X1 Bit Descriptions

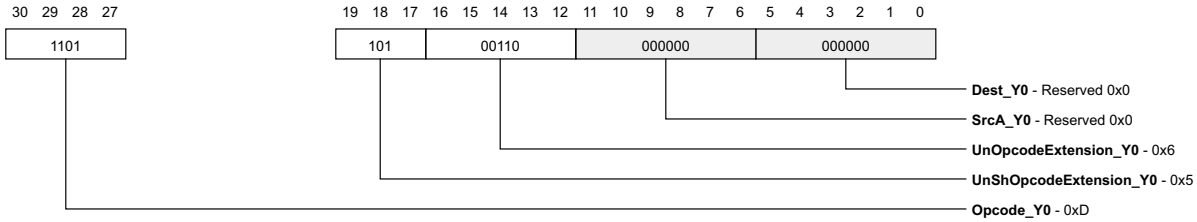

Figure 4-295: fnop in Y0 Bit Descriptions

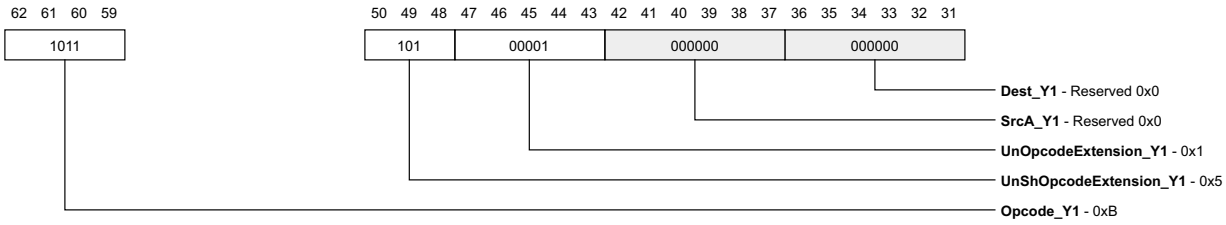

Figure 4-296: fnop in Y1 Bit Descriptions

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

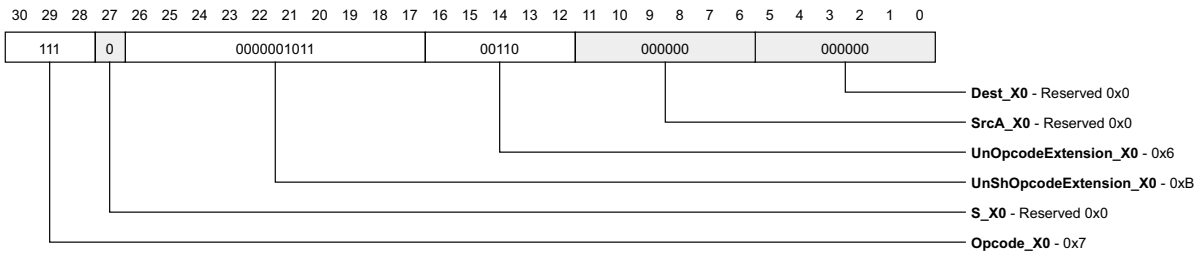

Figure 4-297: nop in X0 Bit Descriptions

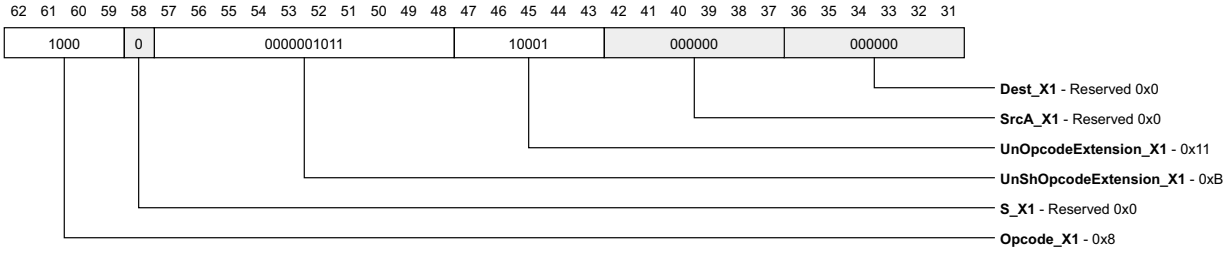

Figure 4-298: nop in X1 Bit Descriptions

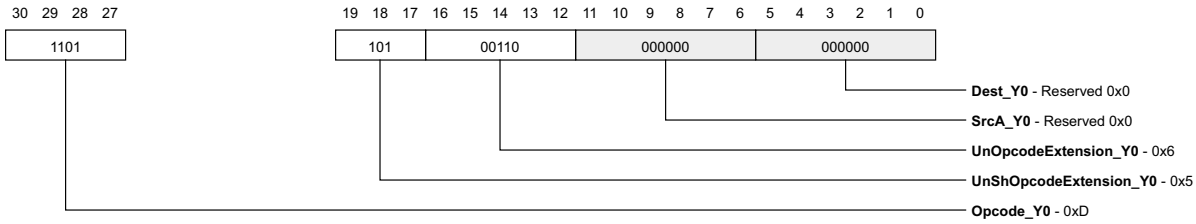

Figure 4-299: nop in Y0 Bit Descriptions

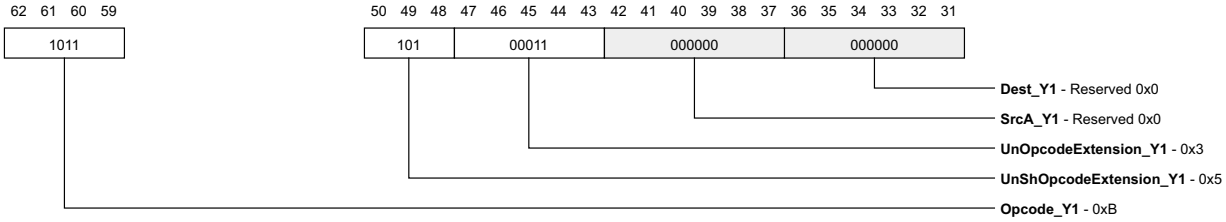

Figure 4-300: nop in Y1 Bit Descriptions

4.1.13 SIMD Instructions

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

- addb: Add Bytes
- addbs_u: Add Bytes Saturating Unsigned
- addh: Add Half Words
- addhs: Add Half Words Saturating
- addib: Add Immediate Bytes
- addih: Add Immediate Half Words
- adiffb_u: Absolute Difference Unsigned Bytes
- adiffh: Absolute Difference Half Words
- avgb_u: Average Byte Unsigned
- avgh: Average Half Words
- inthb: Interleave High Byte
- inthh: Interleave High Half Words
- intlb: Interleave Low Byte
- intlh: Interleave Low Half Words
- maxb_u: Maximum Byte Unsigned
- maxh: Maximum Half Words
- maxib_u: Maximum Immediate Byte Unsigned
- maxih: Maximum Immediate Half Words
- minb_u: Minimum Byte Unsigned
- minh: Minimum Half Words
- minib_u: Minimum Immediate Byte Unsigned
- minih: Minimum Immediate Half Words
- mnzb: Mask Not Zero Byte
- mnzh: Mask Not Zero Half Words
- mzb: Mask Zero Byte
- mzh: Mask Zero Half Words
- packbs_u: Pack Half Words Saturating
- packhb: Pack High Byte
- packhs: Pack Half Words Saturating
- packlb: Pack Low Byte
- sadab_u: Sum of Absolute Difference Accumulate Unsigned Bytes
- sadah: Sum of Absolute Difference Accumulate Half Words
- sadah_u: Sum of Absolute Difference Accumulate Unsigned Half Words
- sadb_u: Sum of Absolute Difference Unsigned Bytes
- sadh: Sum of Absolute Difference Half Words

- sadh_u: Sum of Absolute Difference Unsigned Half Words
- seqb: Set Equal to Byte
- seqh: Set Equal To Half Words
- seqib: Set Equal To Immediate Byte
- seqih: Set Equal To Immediate Half Words
- shlb: Logical Shift Left Bytes
- shlh: Logical Shift Left Half Words
- shlib: Logical Shift Left Immediate Bytes
- shlih: Logical Shift Left Immediate Half Words
- shrb: Logical Shift Right Bytes
- shrh: Logical Shift Right Half Words
- shrib: Logical Shift Right Immediate Bytes
- shrih: Logical Shift Right Immediate Half Words
- sltb: Set Less Than Byte
- sltb_u: Set Less Than Unsigned Byte
- slteb: Set Less Than or Equal Byte
- slteb_u: Set Less Than or Equal Unsigned Byte
- slteh: Set Less Than or Equal Half Words
- slteh_u: Set Less Than or Equal Unsigned Half Words
- slth: Set Less Than Half Words
- slth_u: Set Less Than Unsigned Half Words
- sltib: Set Less Than Immediate Byte
- sltib_u: Set Less Than Unsigned Immediate Byte
- sltih: Set Less Than Immediate Half Words
- sltih_u: Set Less Than Unsigned Immediate Half Words
- sneb: Set Not Equal To Byte
- sneh: Set Not Equal To Half Words
- srab: Arithmetic Shift Right Bytes
- srah: Arithmetic Shift Right Half Words
- sraib: Arithmetic Shift Right Immediate Bytes
- sraih: Arithmetic Shift Right Immediate Half Words
- subb: Subtract Bytes
- subbs_u: Subtract Bytes Saturating Unsigned
- subh: Subtract Half Words
- subhs: Subtract Half Words Saturating

addb: Add Bytes

Syntax

```
addb Dest, SrcA, SrcB
```

Example

```
addb r5, r6, r7
```

Description

Add the four bytes in the first source operand to the four bytes in the second source operand.

Functional Description

```
UnsignedMachineWord 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

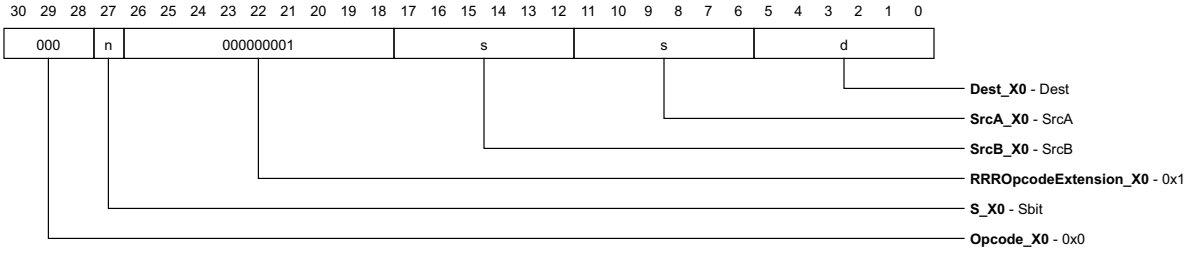

Figure 4-301: addb in X0 Bit Descriptions

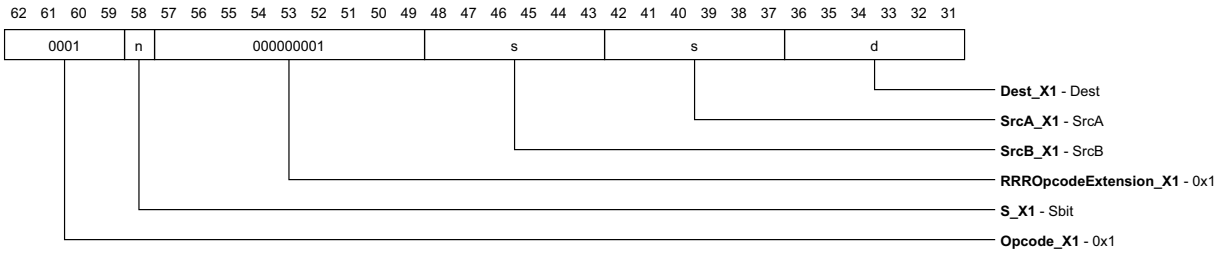

Figure 4-302: addb in X1 Bit Descriptions

addbs_u: Add Bytes Saturating Unsigned

Syntax

```
addbs_u Dest, SrcA, SrcB
```

Example

```
addbs_u r5, r6, r7
```

Description

Add the four bytes in the first source operand to the four bytes in the second source operand and saturate each result to 0 or the maximum positive value.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

```
UnsignedMachineWord 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

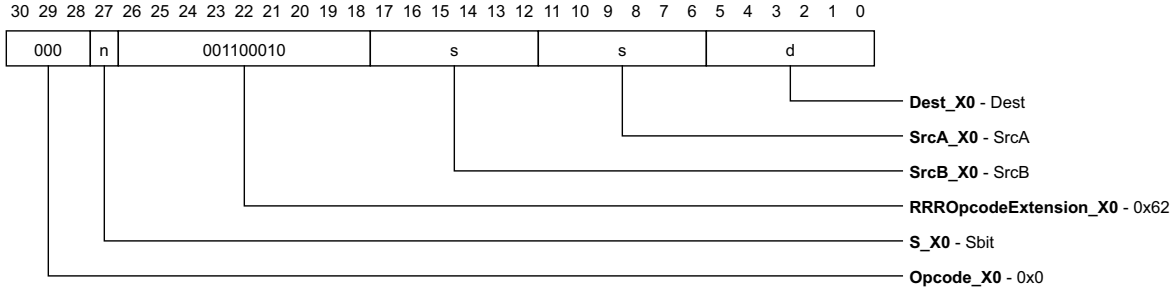

Figure 4-303: addbs_u in X0 Bit Descriptions

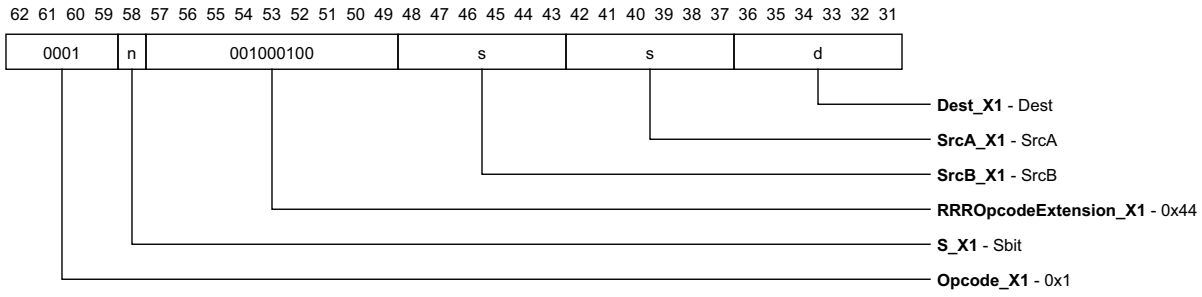

Figure 4-304: addbs_u in X1 Bit Descriptions

addh: Add Half Words

Syntax

```
addh Dest, SrcA, SrcB
```

Example

```
addh r5, r6, r7
```

Description

Add the pair of half words in the first source operand to the pair of half words in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output =
      setHalfWord(output, counter,
                  (getHalfWord(rf[SrcA], counter) +
                   getHalfWord(rf[SrcB], counter)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

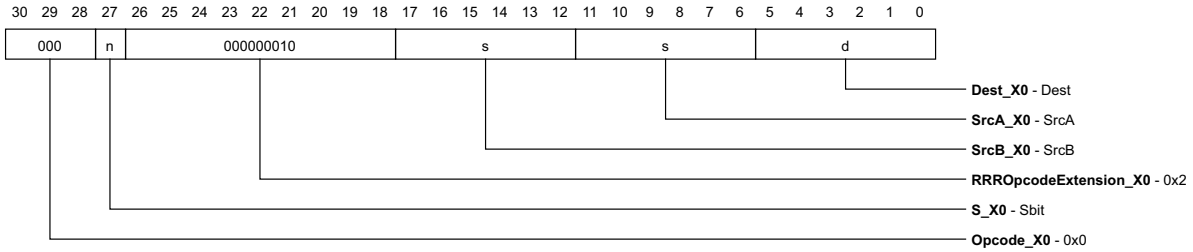

Figure 4-305: addh in X0 Bit Descriptions

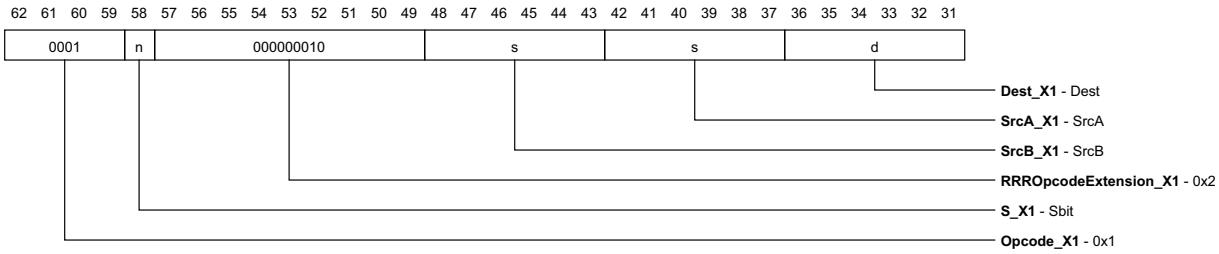

Figure 4-306: addh in X1 Bit Descriptions

addhs: Add Half Words Saturating

Syntax

```
addhs Dest, SrcA, SrcB
```

Example

```
addhs r5, r6, r7
```

Description

Add the pair of half words in the first source operand to the pair of half words in the second source operand and saturate each result to the minimum negative value or maximum positive value.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output =
      setHalfWord(output, counter,
                  signed_saturate16(signExtend16
                                    (getHalfWord(rf[SrcA], counter)) +
                                    signExtend16(getHalfWord
                                                 (rf[SrcB],
                                                  counter))));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

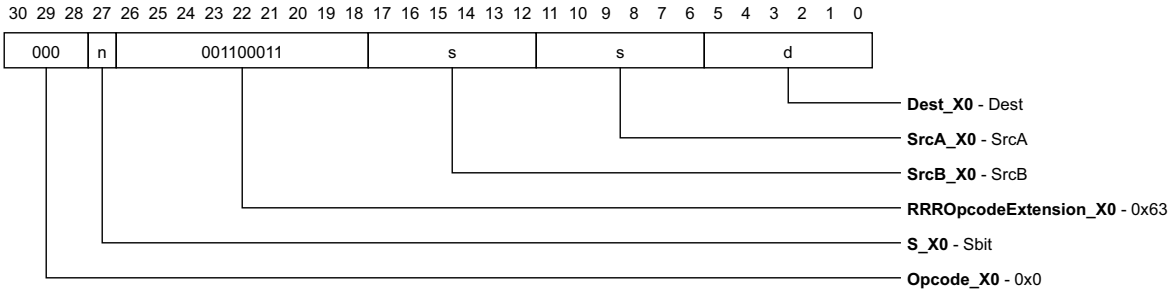

Figure 4-307: addhs in X0 Bit Descriptions

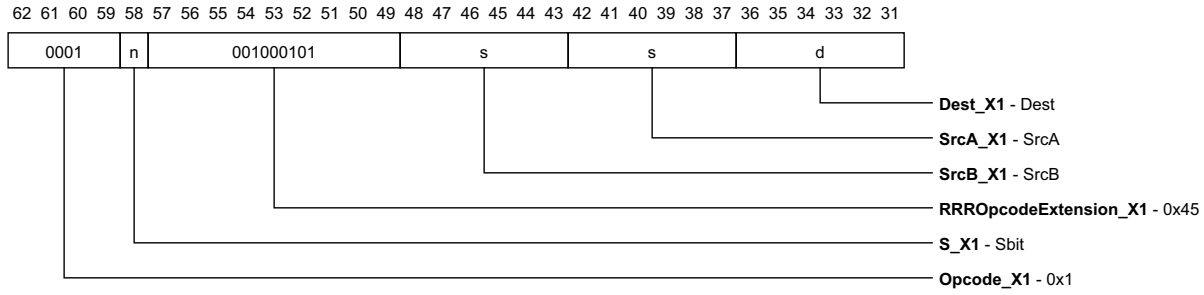

Figure 4-308: addhs in X1 Bit Descriptions

addib: Add Immediate Bytes

Syntax

```
addib Dest, SrcA, Imm8
```

Example

```
addib r5, r6, 5
```

Description

Add an immediate to all four of the bytes in the source operand.

Functional Description

```
UnsignedMachineWord 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

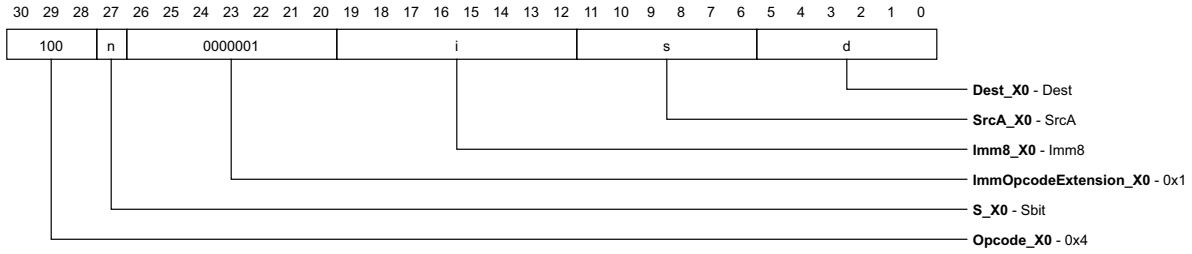

Figure 4-309: addib in X0 Bit Descriptions

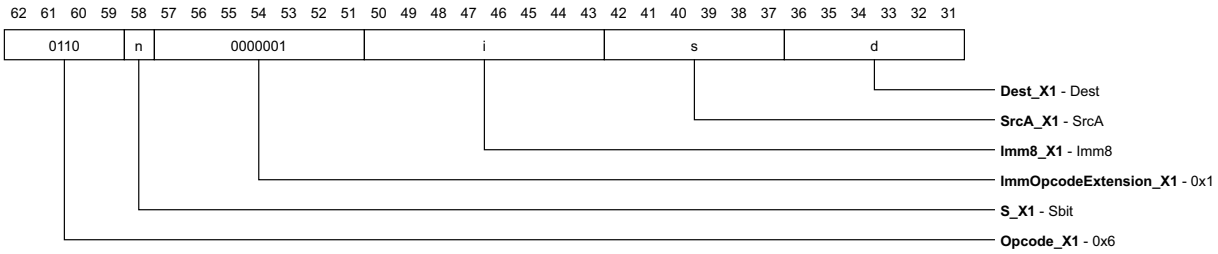

Figure 4-310: addib in X1 Bit Descriptions

addih: Add Immediate Half Words

Syntax

```
addih Dest, SrcA, Imm8
```

Example

```
addih r5, r6, 5
```

Description

Add a sign extended immediate to both of the half words in the source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output =
      setHalfWord(output, counter,
                  (getHalfWord(rf[SrcA], counter) +
                   signExtend8(Imm8)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

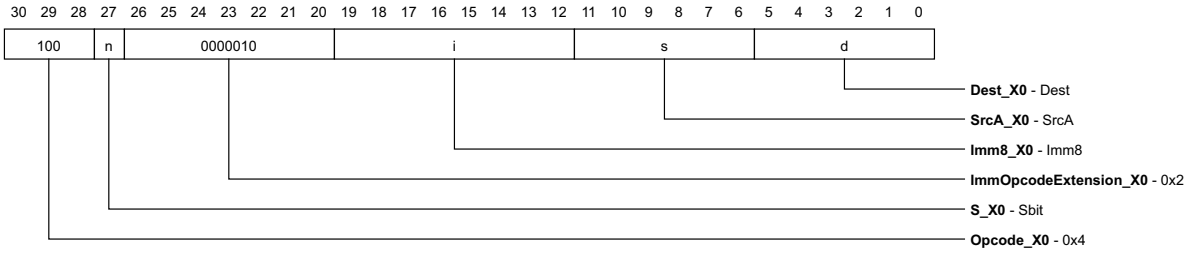

Figure 4-311: addih in X0 Bit Descriptions

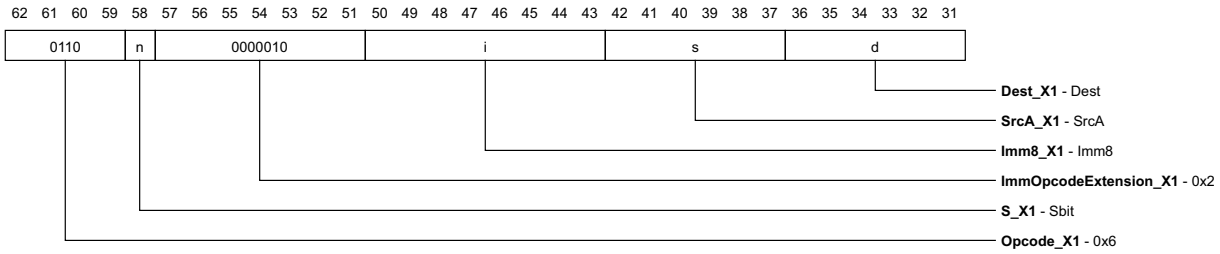

Figure 4-312: addih in X1 Bit Descriptions

adiffb_u: Absolute Difference Unsigned Bytes

Syntax

```
adiffb_u Dest, SrcA, SrcB
```

Example

```
adiffb_u r5, r6, r7
```

Description

Compute the absolute differences between the four bytes in the first source operand and the four bytes in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   output |=
      abs(((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK) -
          ((rf[SrcB]  >> (counter * BYTE_SIZE)) & BYTE_MASK)) <<
      (counter * BYTE_SIZE);
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

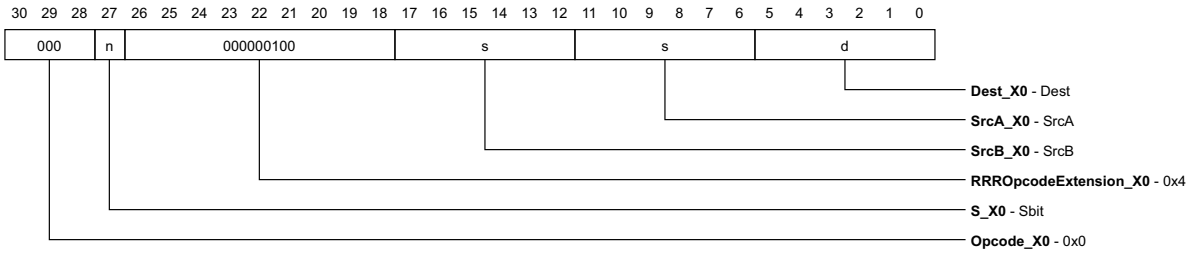

Figure 4-313: adiffb_u in X0 Bit Descriptions

adiffh: Absolute Difference Half Words

Syntax

```
adiffh Dest, SrcA, SrcB
```

Example

```
adiffh r5, r6, r7
```

Description

Compute the absolute differences between the pair of half words in the first source operand and the pair of half words in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output |=
      abs(signExtend16
          ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK) -
          signExtend16((rf[SrcB] >> (counter * HALF_WORD_SIZE)) &
                       HALF_WORD_MASK)) << (counter *
                                            HALF_WORD_SIZE);
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

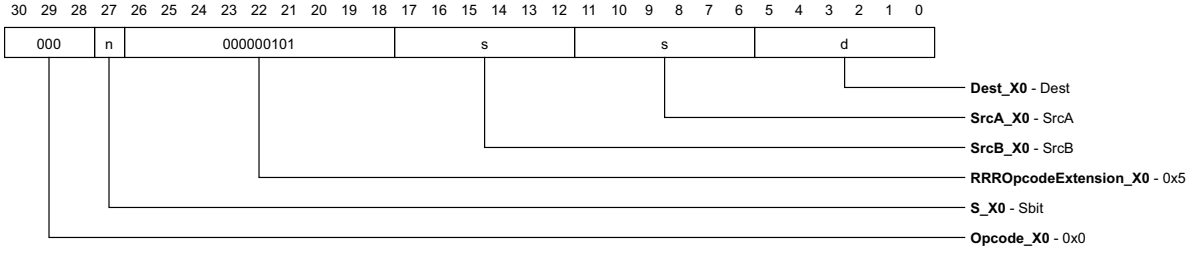

Figure 4-314: adiffh in X0 Bit Descriptions

avgb_u: Average Byte Unsigned

Syntax

```
avgb_u Dest, SrcA, SrcB
```

Example

```
avgb_u r5, r6, r7
```

Description

Compute the average of the four bytes in the first source operand and the four bytes in the second source operand, rounding upwards.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   UnsignedMachineWord srca =
      ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   UnsignedMachineWord srcb =
      ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
      ((((srca + srcb +
          1) >> 1) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

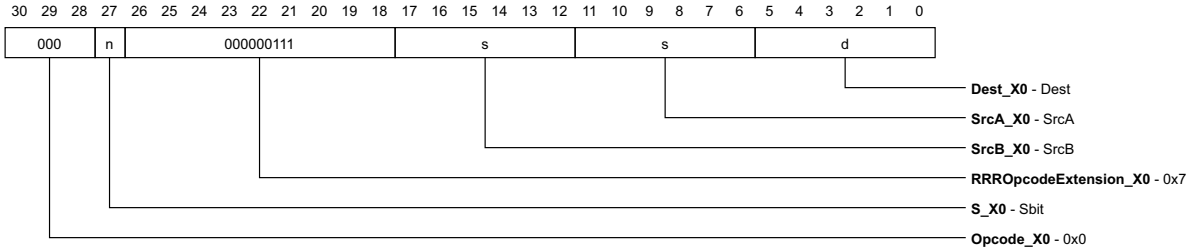

Figure 4-315: avgb_u in X0 Bit Descriptions

avgh: Average Half Words

Syntax

```
avgh Dest, SrcA, SrcB
```

Example

```
avgh r5, r6, r7
```

Description

Compute the average between the pair of half words in the first source operand and the pair of half words in the second source operand, rounding upwards.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   SignedMachineWord srca =
      signExtend16((rf[SrcA] >> (counter * HALF_WORD_SIZE)) &
                   HALF_WORD_MASK);
   SignedMachineWord srcb =
      signExtend16((rf[SrcB] >> (counter * HALF_WORD_SIZE)) &
                   HALF_WORD_MASK);
   output |=
      ((((srca + srcb +
          1) >> 1) & HALF_WORD_MASK) << (counter * HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

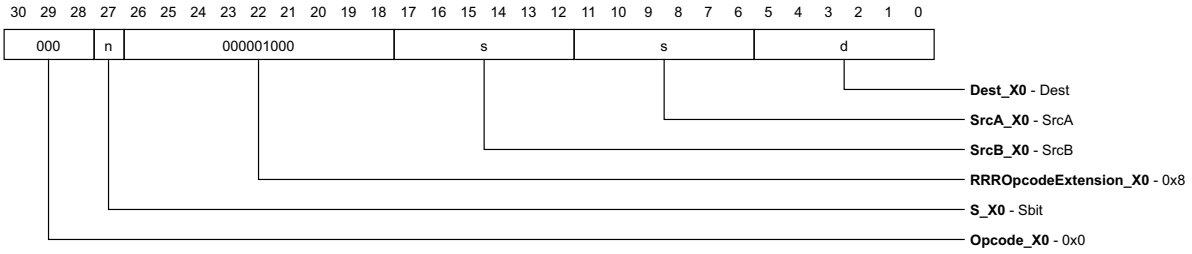

Figure 4-316: avgh in X0 Bit Descriptions

inthb: Interleave High Byte

Syntax

```
inthb Dest, SrcA, SrcB
```

Example

```
inthb r5, r6, r7
```

Description

Interleave the two high-order bytes of the first operand with the two 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 {A3,A2,A1,A0} and the second operand contains the packed bytes {B3,B2,B1,B0} then the result will be {A3,B3,A2,B2}.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   bool asel = ((counter & 1) == 1);
   int in_sel = 2 + counter / 2;
   int16_t srca = ((rf[SrcA] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
   int16_t srcb = ((rf[SrcB] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
   output |=
      (((asel ? srca : srcb) & BYTE_MASK) << (counter * BYTE_SIZE));
} rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

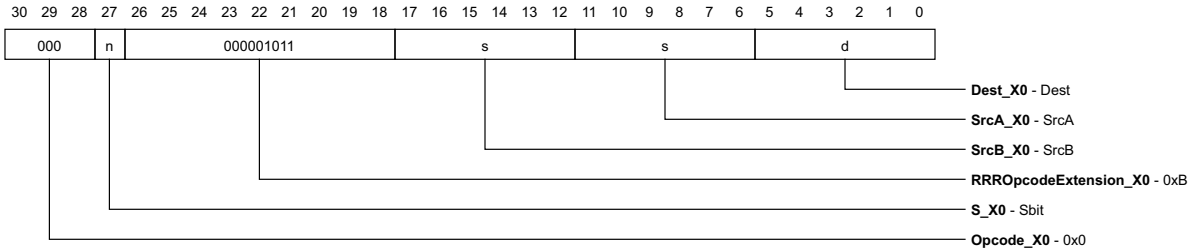

Figure 4-317: inthb in X0 Bit Descriptions

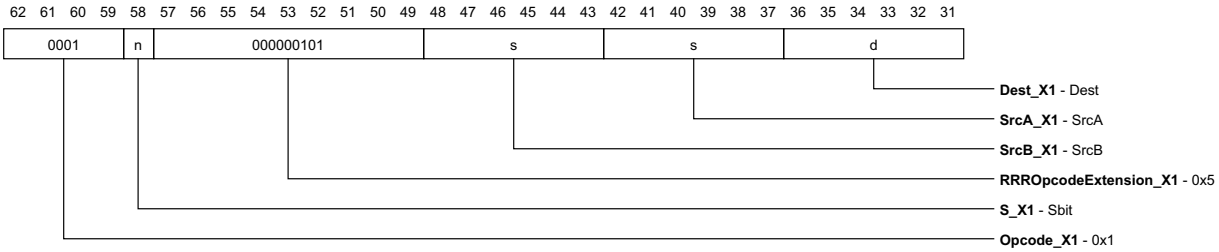

Figure 4-318: inthb in X1 Bit Descriptions

inthh: Interleave High Half Words

Syntax

```
inthh Dest, SrcA, SrcB
```

Example

```
inthh r5, r6, r7
```

Description

Interleave the high-order half word of the first operand with the high-order half word of the second operand. The high-order half word of the result will be the high-order half word of the first operand. For example if the first operand contains the packed half words {A1,A0} and the second operand contains the packed half words {B1,B0} then the result will be {A1,B1}.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   bool asel = ((counter & 1) == 1);
   int in_sel = 1 + counter / 2;
   int16_t srca =
      ((rf[SrcA] >> (in_sel * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb =
      ((rf[SrcB] >> (in_sel * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      (((asel ? srca : srcb) & HALF_WORD_MASK) <<
       (counter * HALF_WORD_SIZE));
} rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

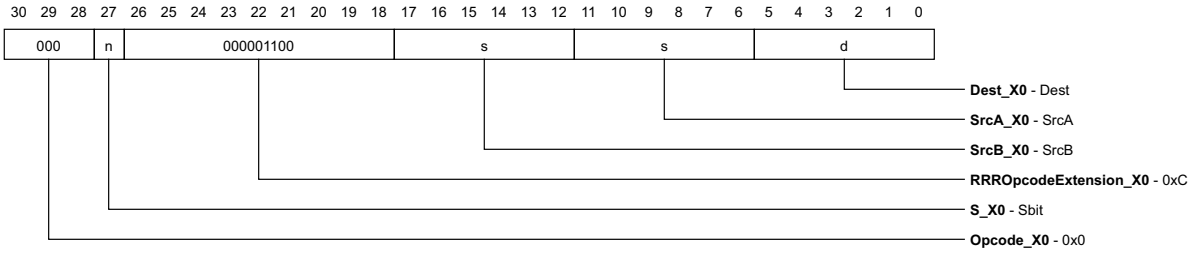

Figure 4-319: inthh in X0 Bit Descriptions

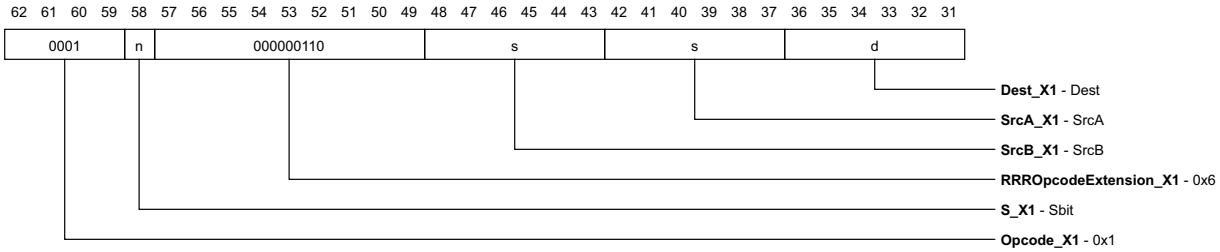

Figure 4-320: inthh in X1 Bit Descriptions

intlb: Interleave Low Byte

Syntax

```
intlb Dest, SrcA, SrcB
```

Example

```
intlb r5, r6, r7
```

Description

Interleave the two low-order bytes of the first operand with the two 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 {A3,A2,A1,A0} and the second operand contains the packed bytes {B3,B2,B1,B0} then the result will be {A1,B1,A0,B0}.

Functional Description

```
UnsignedMachineWord 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;
   int16_t srca = ((rf[SrcA] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
   int16_t srcb = ((rf[SrcB] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
   output |=
      (((asel ? srca : srcb) & BYTE_MASK) << (counter * BYTE_SIZE));
} rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

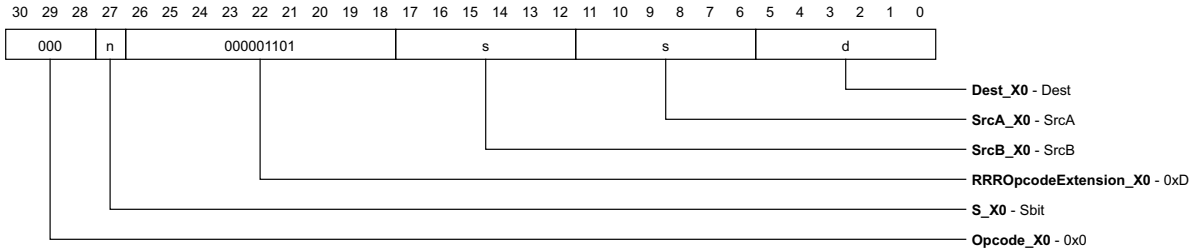

Figure 4-321: intlb in X0 Bit Descriptions

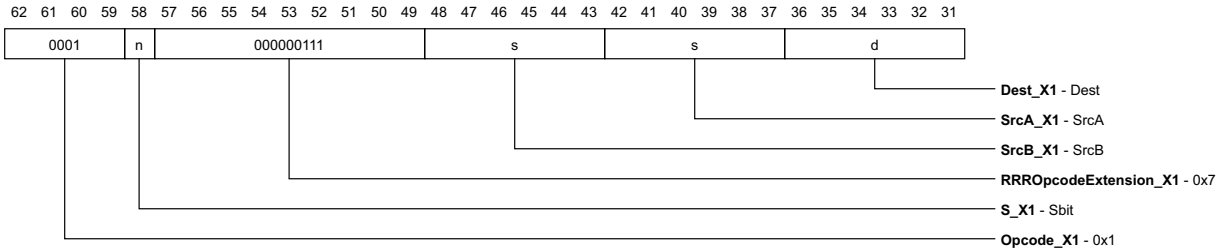

Figure 4-322: intlb in X1 Bit Descriptions

intlh: Interleave Low Half Words

Syntax

```
intlh Dest, SrcA, SrcB
```

Example

```
intlh r5, r6, r7
```

Description

Interleave the low-order half word of the first operand with the low-order half word of the second operand. The low-order half word of the result will be the low-order half word of the second operand. For example if the first operand contains the packed half words {A1,A0} and the second operand contains the packed half words {B1,B0} then the result will be {A0,B0}.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   bool asel = ((counter & 1) == 1);
   int in_sel = 0 + counter / 2;
   int16_t srca =
      ((rf[SrcA] >> (in_sel * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb =
      ((rf[SrcB] >> (in_sel * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      (((asel ? srca : srcb) & HALF_WORD_MASK) <<
       (counter * HALF_WORD_SIZE));
} rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

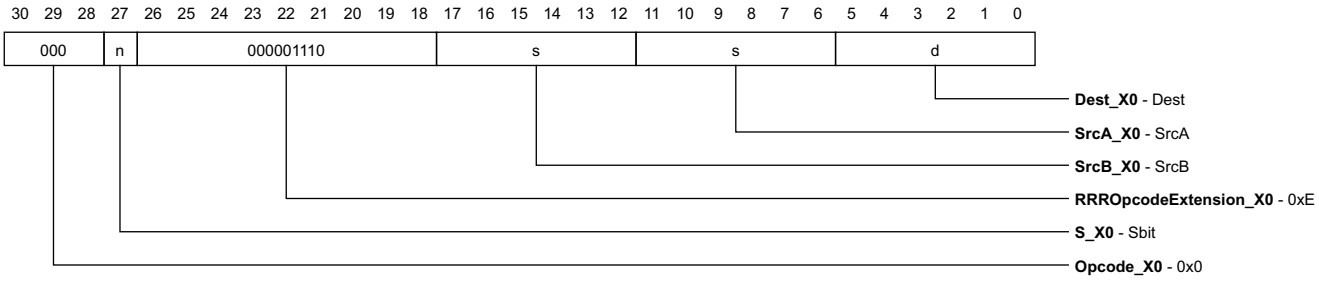

Figure 4-323: intlh in X0 Bit Descriptions

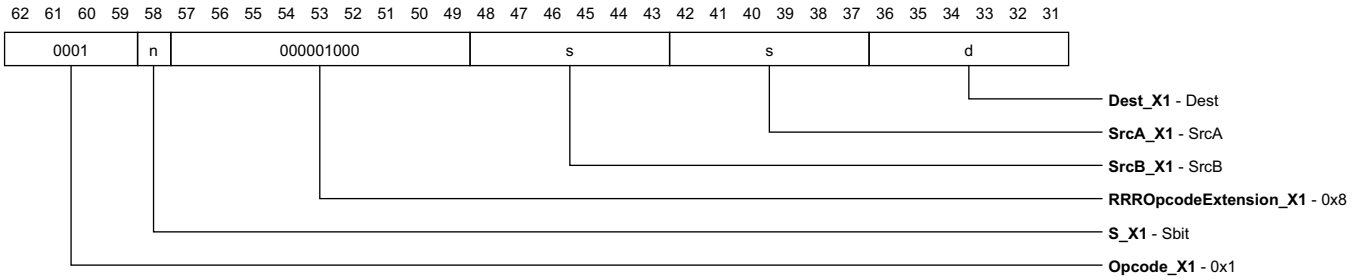

Figure 4-324: intlh in X1 Bit Descriptions

maxb_u: Maximum Byte Unsigned

Syntax

```
maxb_u Dest, SrcA, SrcB
```

Example

```
maxb_u r5, r6, r7
```

Description

Set each of the bytes 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   uint8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   uint8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
       ((((srca >
        srcb) ? srca : srcb) & BYTE_MASK) << (counter *
                                              BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

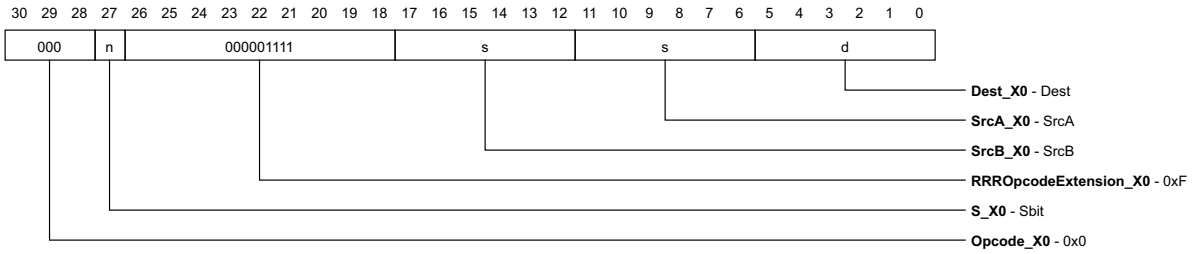

Figure 4-325: maxb_u in X0 Bit Descriptions

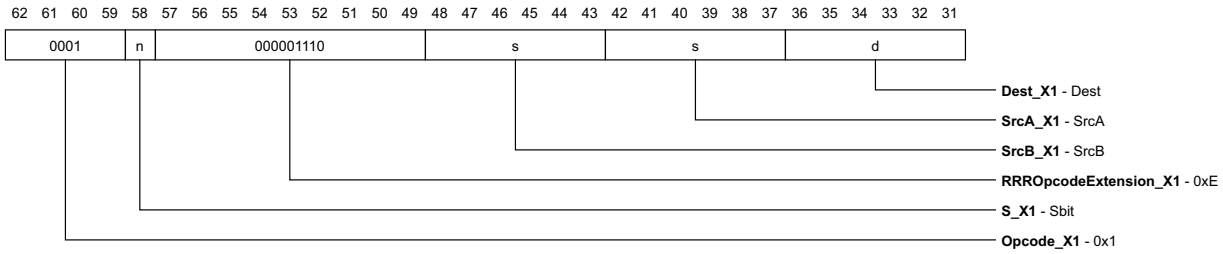

Figure 4-326: maxb_u in X1 Bit Descriptions

maxh: Maximum Half Words

Syntax

```
maxh Dest, SrcA, SrcB
```

Example

```
maxh r5, r6, r7
```

Description

Set each of the half words in the destination to the maximum of the corresponding half word in the first source operand and the corresponding half word in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb =
      ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      ((((srca >
          srcb) ? srca : srcb) & HALF_WORD_MASK) << (counter *
                                                     HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

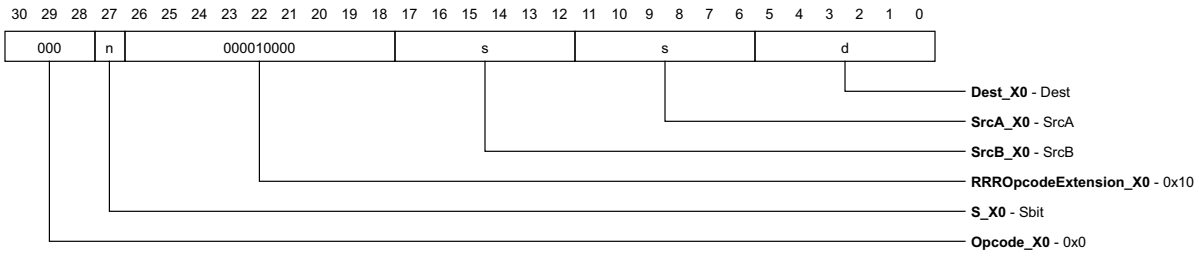

Figure 4-327: maxh in X0 Bit Descriptions

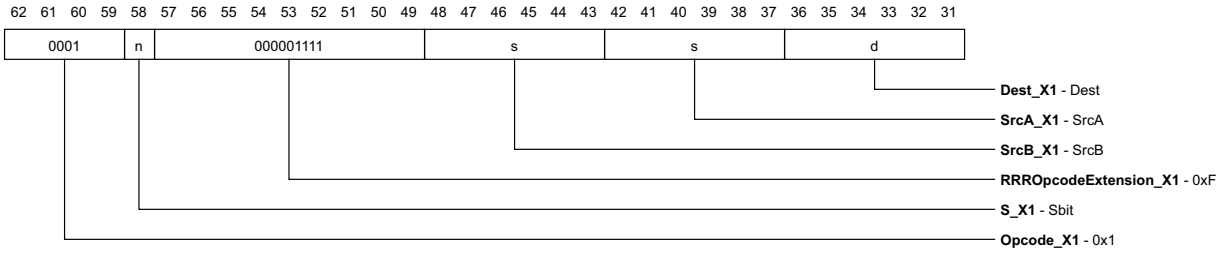

Figure 4-328: maxh in X1 Bit Descriptions

maxib_u: Maximum Immediate Byte Unsigned

Syntax

```
maxib_u Dest, SrcA, Imm8
```

Example

```
maxib_u r5, r6, 5
```

Description

Set each of the bytes in the destination to the maximum of the corresponding byte in the first source operand and the sign extended immediate.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
uint8_t immb = Imm8;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   uint8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
      (((srca >
         immb) ? srca : immb) & BYTE_MASK) << (counter *
                                                BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

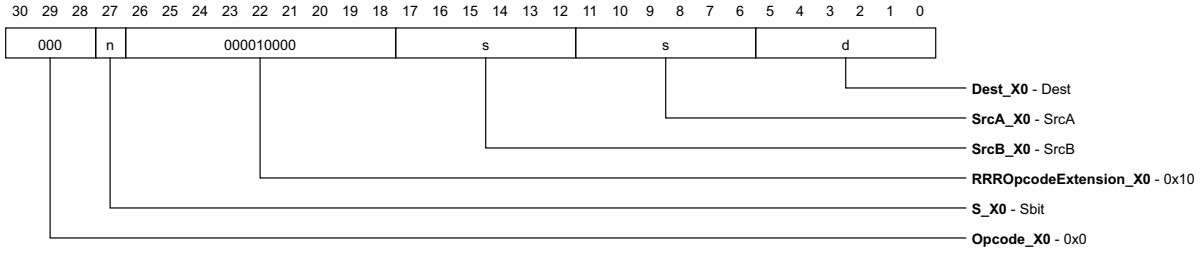

Figure 4-329: maxh in X0 Bit Descriptions

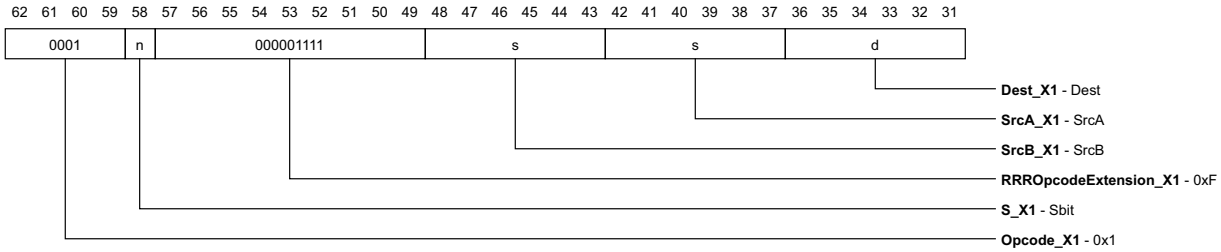

Figure 4-330: maxh in X1 Bit Descriptions

maxih: Maximum Immediate Half Words

Syntax

```
maxih Dest, SrcA, Imm8
```

Example

```
maxih r5, r6, 5
```

Description

Set each of the half words in the destination to the maximum of the corresponding half word in the first source operand and the sign extended immediate.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      ((((srca >
   signExtend8(Imm8)) ? srca : signExtend8(Imm8)) &
      HALF_WORD_MASK) << (counter * HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

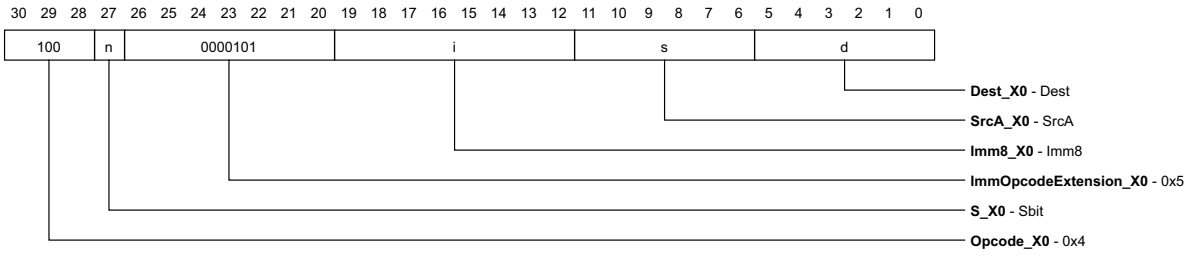

Figure 4-331: maxih in X0 Bit Descriptions

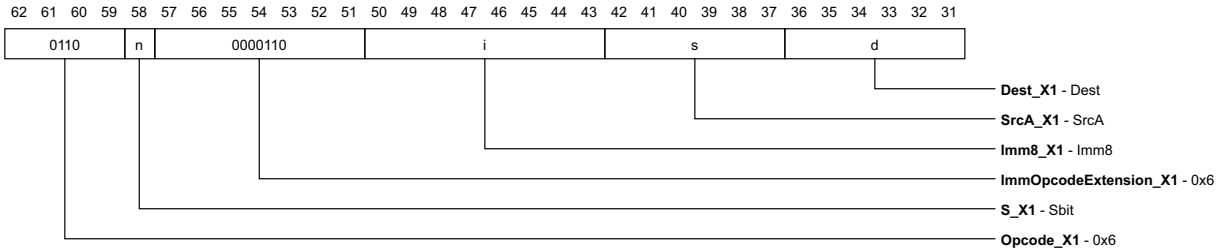

Figure 4-332: maxih in X1 Bit Descriptions

minb_u: Minimum Byte Unsigned

Syntax

```
minb_u Dest, SrcA, SrcB
```

Example

```
minb u r5, r6, r7
```

Description

Set each of the bytes 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   uint8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   uint8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
       ((((srca <
           srcb) ? srca : srcb) & BYTE_MASK) << (counter *
                                                 BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

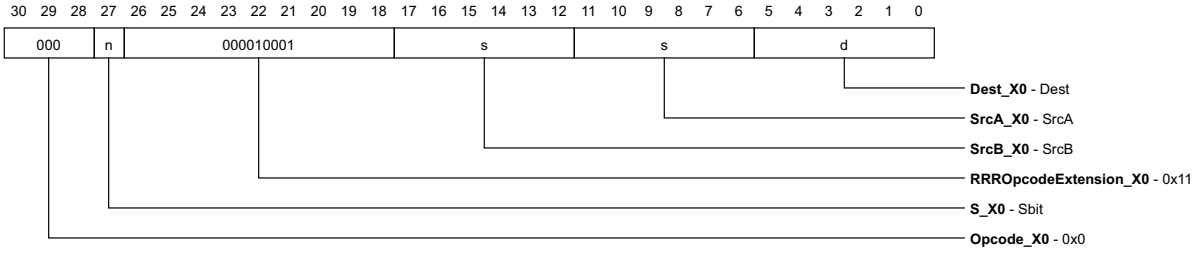

Figure 4-333: minb in X0 Bit Descriptions

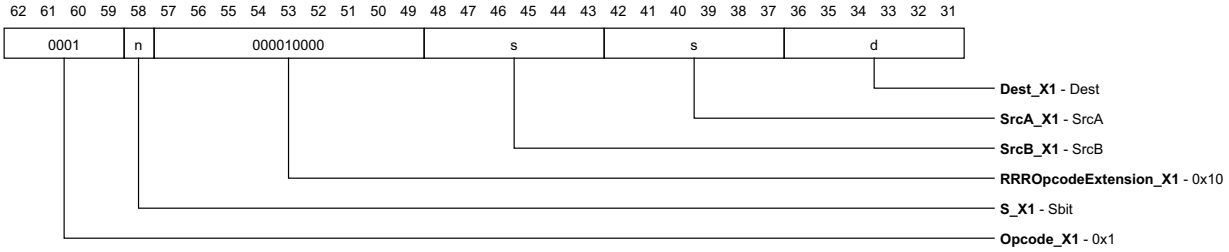

Figure 4-334: minb in X1 Bit Descriptions

minh: Minimum Half Words

Syntax

```
minh Dest, SrcA, SrcB
```

Example

```
minh r5, r6, r7
```

Description

Set each of the half words in the destination to the minimum of the corresponding half word in the first source operand and the corresponding half word in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb =
      ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      ((((srca <
          srcb) ? srca : srcb) & HALF_WORD_MASK) << (counter *
                                                     HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

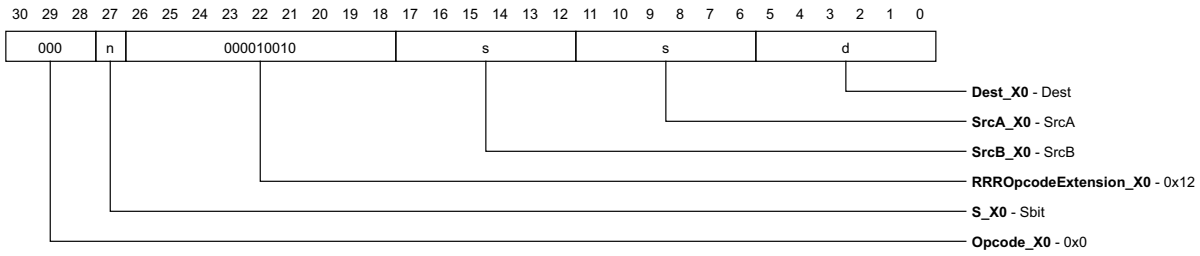

Figure 4-335: minh in X0 Bit Descriptions

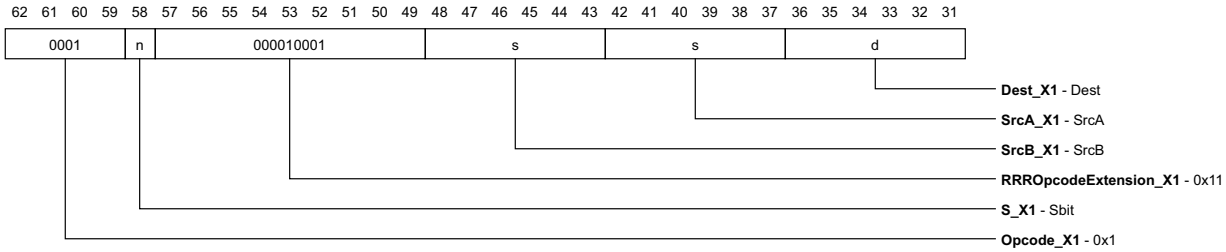

Figure 4-336: minh in X1 Bit Descriptions

minib_u: Minimum Immediate Byte Unsigned

Syntax

```
minib_u Dest, SrcA, Imm8
```

Example

```
minib_u r5, r6, 5
```

Description

Set each of the bytes in the destination to the minimum of the corresponding byte in the first source operand and the sign extended immediate.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
uint8_t immb = Imm8;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   uint8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
      ((((srca <
          immb) ? srca : immb) & BYTE_MASK) << (counter *
                                                BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

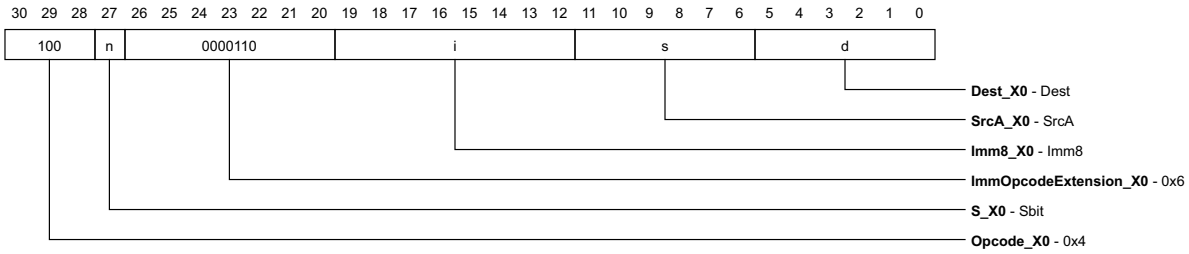

Figure 4-337: minib_u in X0 Bit Descriptions

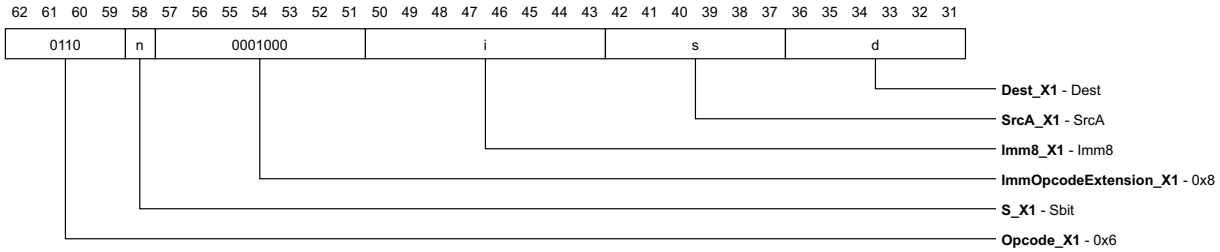

Figure 4-338: minib_u in X1 Bit Descriptions

minih: Minimum Immediate Half Words

Syntax

```
minih Dest, SrcA, Imm8
```

Example

```
minih r5, r6, 5
```

Description

Set each of the half words in the destination to the minimum of the corresponding half word in the first source operand and the sign extended immediate.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      ((((srca <
          signExtend8(Imm8)) ? srca : signExtend8(Imm8)) &
        HALF_WORD_MASK) << (counter * HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

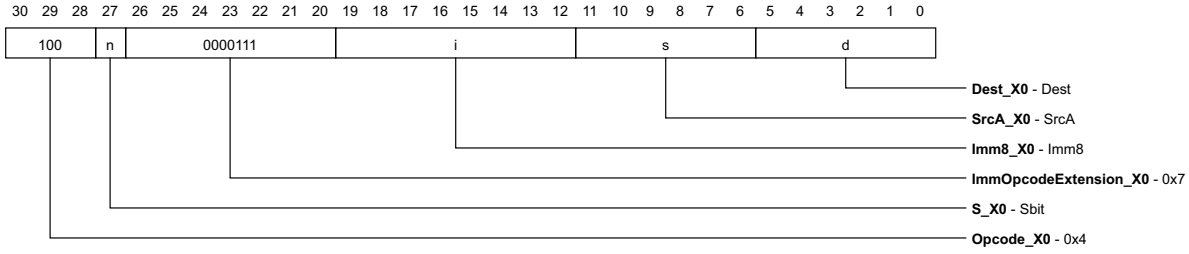

Figure 4-339: minih in X0 Bit Descriptions

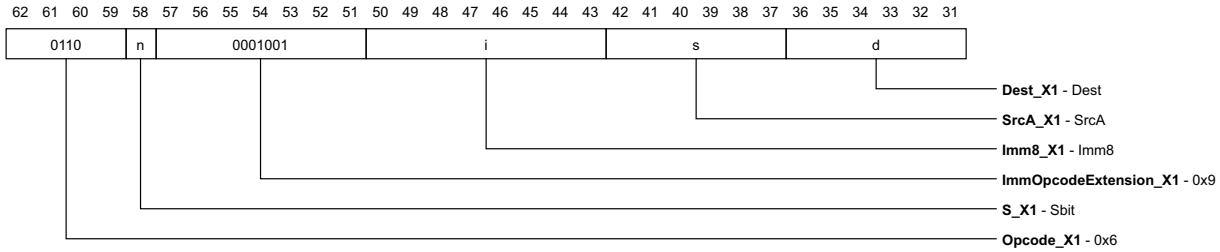

Figure 4-340: minih in X1 Bit Descriptions

mnzb: Mask Not Zero Byte

Syntax

```
mnzb Dest, SrcA, SrcB
```

Example

```
mnzb 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 0, otherwise set it to zero (0).

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   int8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
      ((((srca !=
          0) ? srcb : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

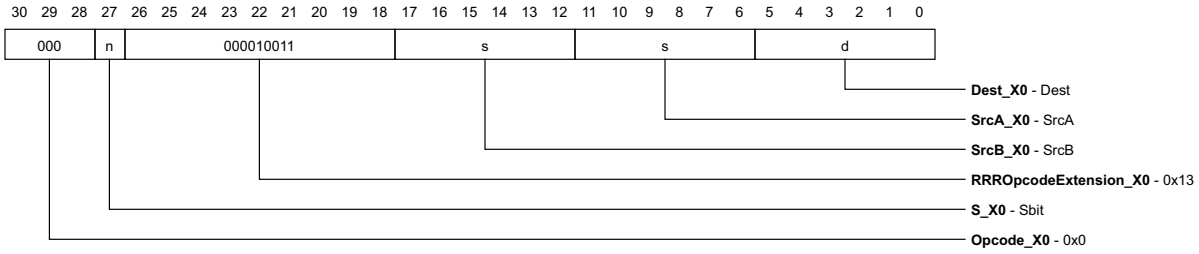

Figure 4-341: mnzb in X0 Bit Descriptions

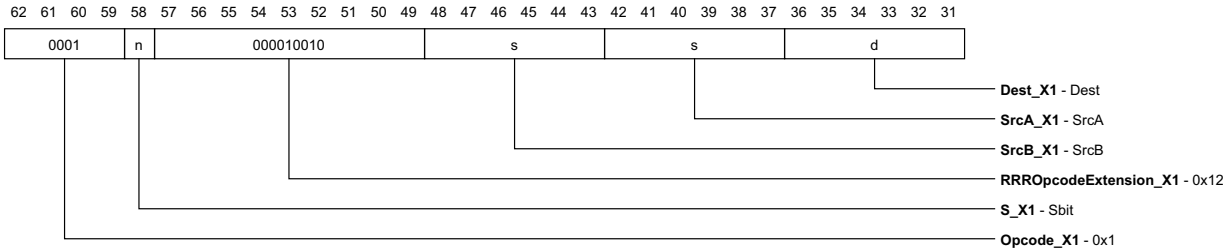

Figure 4-342: mnzb in X1 Bit Descriptions

mnzh: Mask Not Zero Half Words

Syntax

```
mnzh Dest, SrcA, SrcB
```

Example

```
mnzh r5, r6, r7
```

Description

Set each half word in the destination to the corresponding half word of the second operand if the corresponding half word of the first operand is not 0, otherwise set it to zero (0).

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb =
      ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      ((((srca !=
          0) ? srcb : 0) & HALF_WORD_MASK) << (counter *
                                               HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

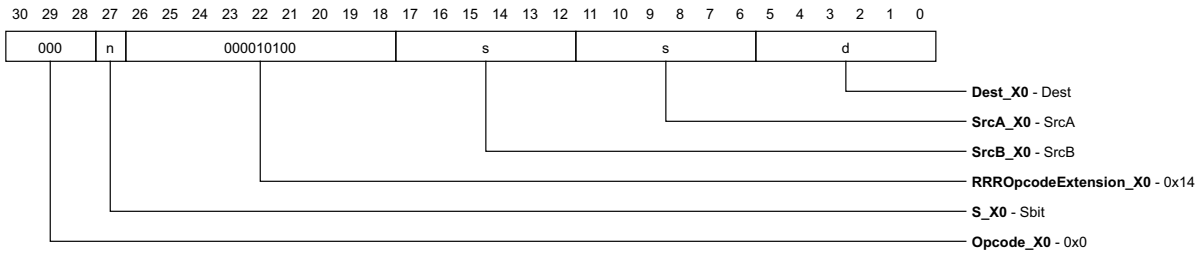

Figure 4-343: mnzh in X0 Bit Descriptions

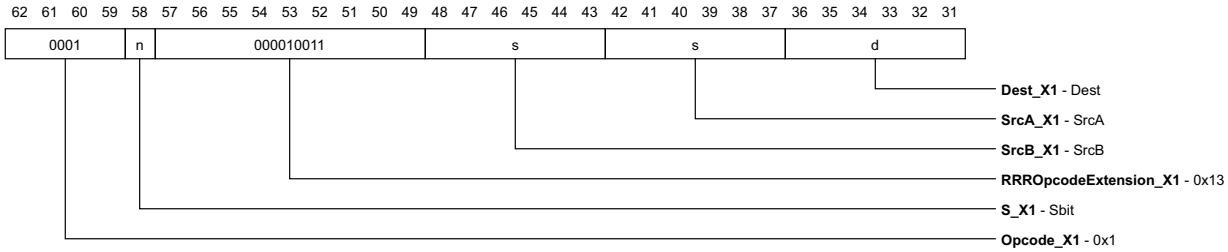

Figure 4-344: mnzh in X1 Bit Descriptions

mzb: Mask Zero Byte

Syntax

```
mzb Dest, SrcA, SrcB
```

Example

```
mzb 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 0, otherwise set it to zero (0).

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   int8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
      ((((srca ==
          0) ? srcb : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

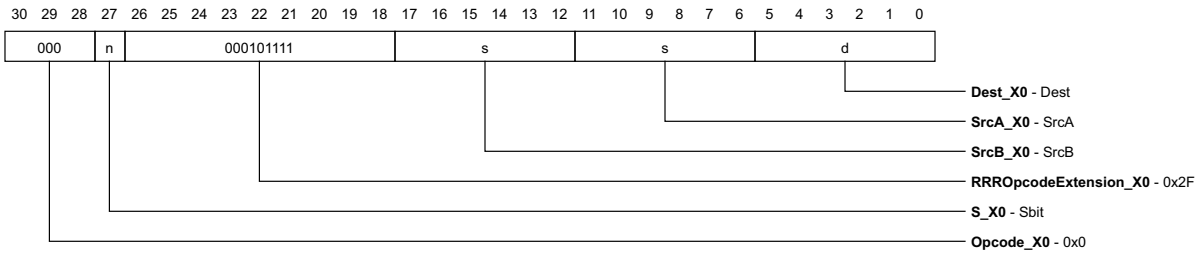

Figure 4-345: mzb in X0 Bit Descriptions

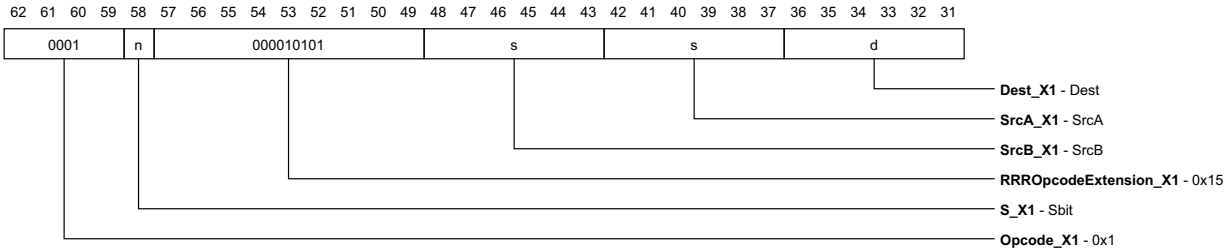

Figure 4-346: mzb in X1 Bit Descriptions

mzh: Mask Zero Half Words

Syntax

```
mzh Dest, SrcA, SrcB
```

Example

```
mzh r5, r6, r7
```

Description

Set each half word in the destination to the corresponding half word of the second operand if the corresponding half word of the first operand is 0, otherwise set it to zero (0).

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb =
      ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
      ((((srca ==
          0) ? srcb : 0) & HALF_WORD_MASK) << (counter *
                                               HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

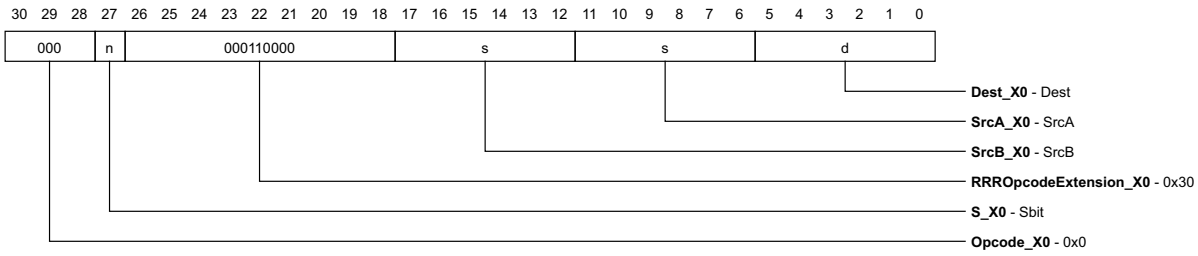

Figure 4-347: mzh in X0 Bit Descriptions

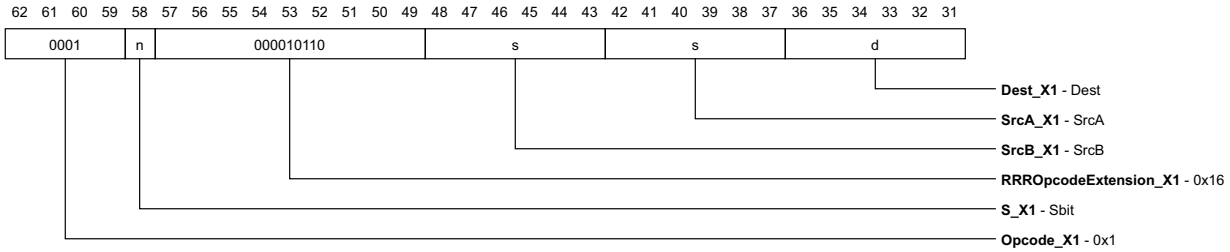

Figure 4-348: mzh in X1 Bit Descriptions

packbs_u: Pack Half Words Saturating

Syntax

```
packbs_u Dest, SrcA, SrcB
```

Example

```
packbs_u r5, r6, r7
```

Description

Saturate each half word of the two source registers to the maximum positive or 0 byte value, and then pack the results into the destination register. The high-order byte of the destination will be the saturated high-order half word of the first operand and the low-order byte of the destination will be the saturated low-order half word of the second operand. For example if the first operand contains the packed half words A1,A0 and the second operand contains the packed half word B1,B0 then the result will be sat A1,sat A0,sat B1,sat B0.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   bool asel = ((counter / 2) == 1);
   int in_sel = counter & 1;
   int16_t srca = signExtend16(getHalfWord(rf[SrcA], in_sel));
   int16_t srcb = signExtend16(getHalfWord(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

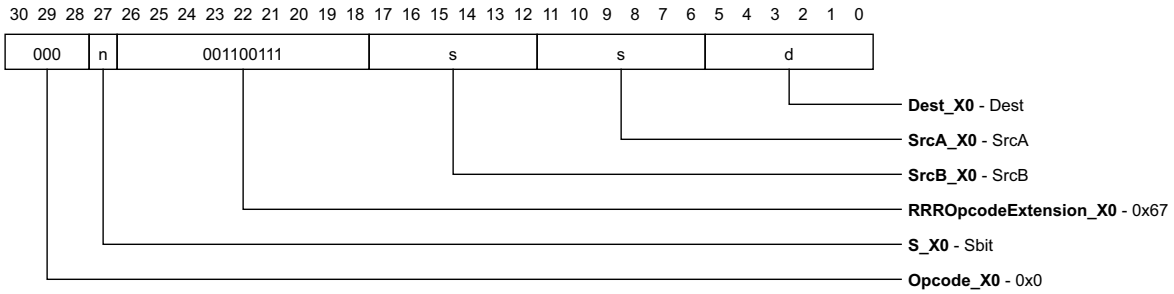

Figure 4-349: packbs_u in X0 Bit Descriptions

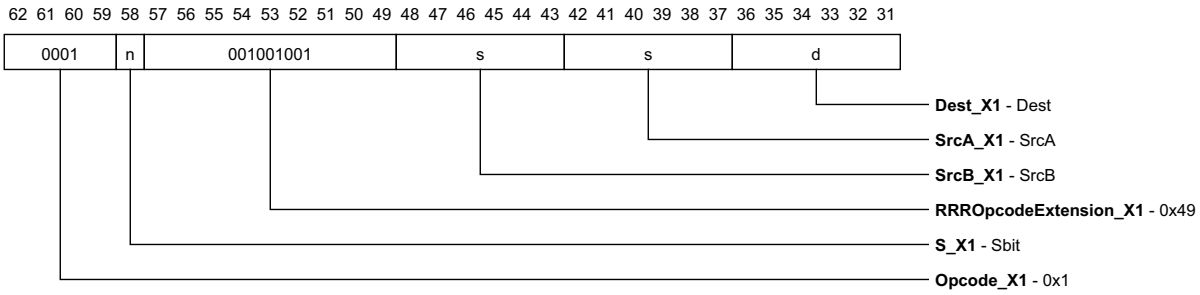

Figure 4-350: packbs_u in X1 Bit Descriptions

packhb: Pack High Byte

Syntax

```
packhb Dest, SrcA, SrcB
```

Example

```
packhb r5, r6, r7
```

Description

Pack the high-order byte of each of the packed half words 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 {A1_1,A1_0,A0_1,A0_0} and the second operand contains the packed bytes {B1_1,B1_0,B0_1,B0_0} then the result will be {A1_1,A0_1,B1_1,B0_1}.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   bool asel = ((counter / 2) == 1);
   int in_sel = 1 + (counter & 1) * 2;
   int8_t srca = ((rf[SrcA] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
   int8_t srcb = ((rf[SrcB] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
   output |=
      (((asel ? srca : srcb) & BYTE_MASK) << (counter * BYTE_SIZE));
} rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

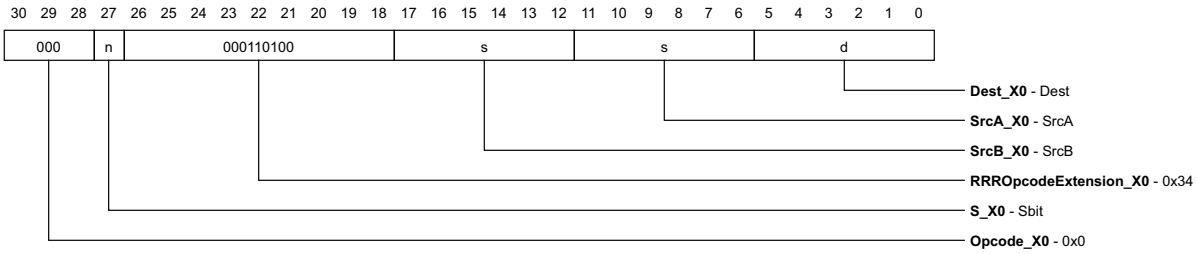

Figure 4-351: packhb in X0 Bit Descriptions

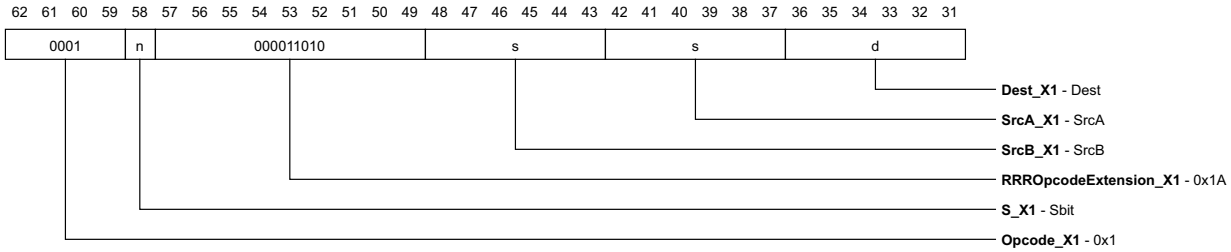

Figure 4-352: packhb in X1 Bit Descriptions

packhs: Pack Half Words Saturating

Syntax

```
packhs Dest, SrcA, SrcB
```

Example

```
packhs r5, r6, r7
```

Description

Saturate each of the two source registers to the maximum positive or minimum negative half word value, and then pack the results into the destination register. The low-order half word of the destination with be the saturated second operand.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   bool asel = counter & 1;
   int16_t srca = signed_saturate16(rf[SrcA]);
   int16_t srcb = signed_saturate16(rf[SrcB]);
   output = setHalfWord(output, counter, (asel ? srca : srcb));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

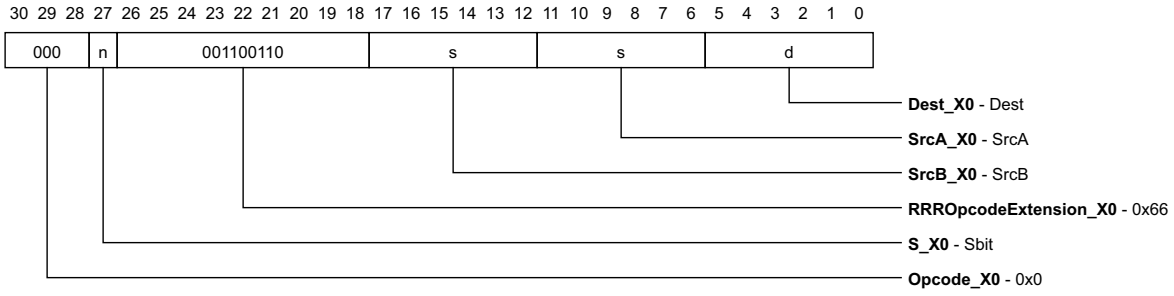

Figure 4-353: packhs in X0 Bit Descriptions

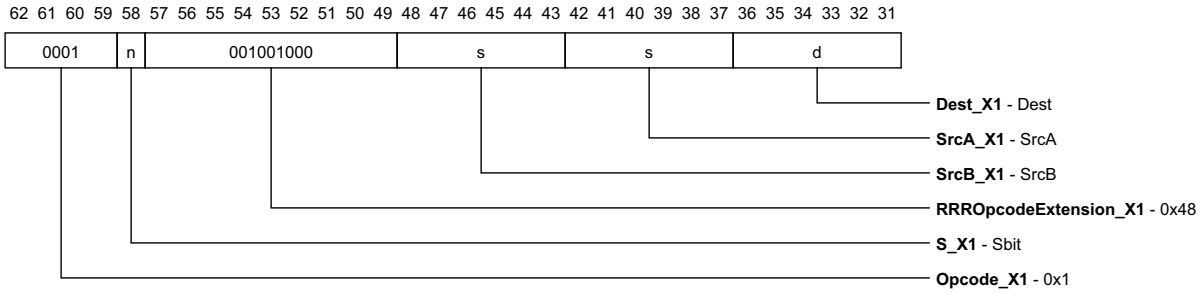

Figure 4-354: packhs in X1 Bit Descriptions

packlb: Pack Low Byte

Syntax

```
packlb Dest, SrcA, SrcB
```

Example

```
packlb r5, r6, r7
```

Description

Pack the low-order byte of each of the packed half words 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 {A1_1,A1_0,A0_1,A0_0} and the second operand contains the packed bytes {B1_1,B1_0,B0_1,B0_0} then the result will be {A1_0,A0_0,B1_0,B0_0}.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
     bool asel = ((counter / 2) == 1);
     int in_sel = 0 + (counter & 1) * 2;
     int8_t srca = ((rf[SrcA] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
     int8_t srcb = ((rf[SrcB] >> (in_sel * BYTE_SIZE)) & BYTE_MASK);
     output |=
         (((asel ? srca : srcb) & BYTE_MASK) << (counter * BYTE_SIZE));
} rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

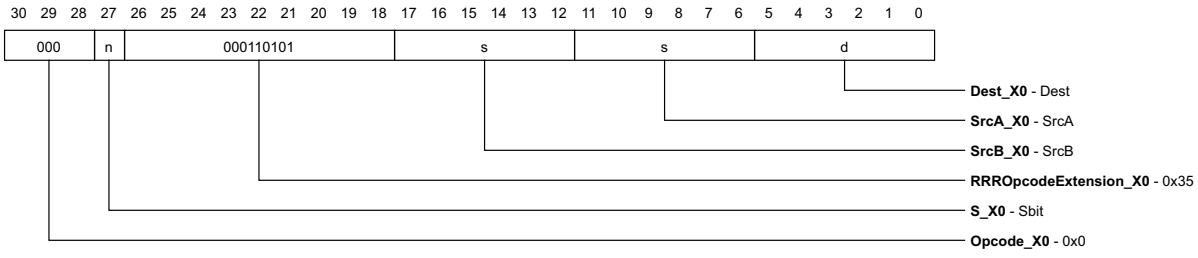

Figure 4-355: packlb in X0 Bit Descriptions

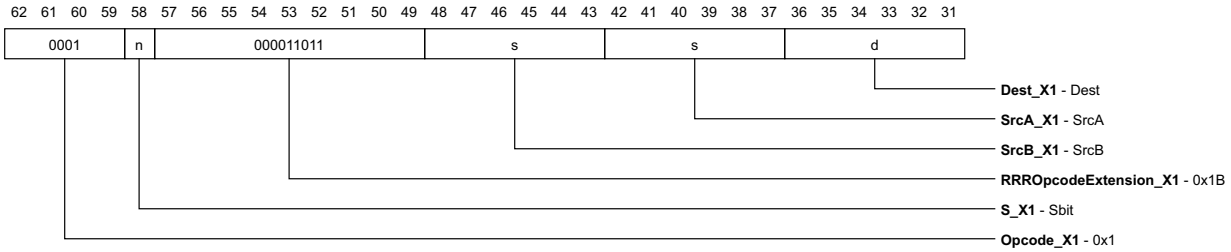

Figure 4-356: packlb in X1 Bit Descriptions

sadab_u: Sum of Absolute Difference Accumulate Unsigned Bytes

Syntax

```
sadab_u Dest, SrcA, SrcB
```

Example

```
sadab_u r5, r6, r7
```

Description

Sum the absolute differences between the four bytes in the first source operand and the four bytes in the second source operand and accumulate the sum into the destination register.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   output +=
       abs(((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK) -
           ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK));
}
rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

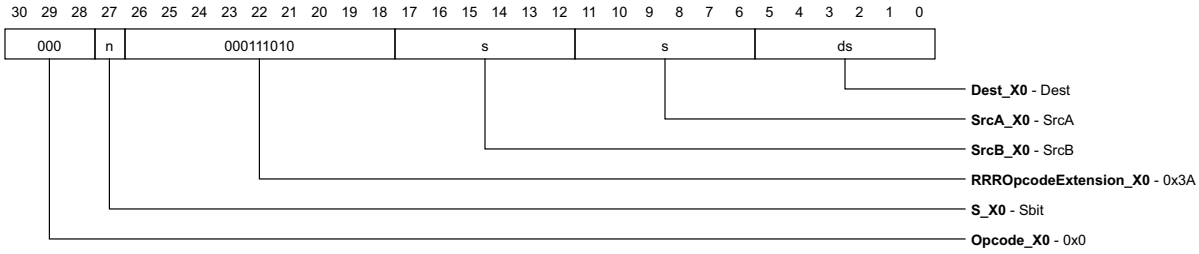

Figure 4-357: sadab_u in X0 Bit Descriptions

sadah: Sum of Absolute Difference Accumulate Half Words

Syntax

```
sadah Dest, SrcA, SrcB
```

Example

```
sadah r5, r6, r7
```

Description

Sum the absolute differences between the pair of half words in the first source operand and the pair of half words in the second source operand and accumulate the sum into the destination register.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    output +=
      abs(signExtend16(getHalfWord(rf[SrcA], counter)) -
   signExtend16(getHalfWord(rf[SrcB], counter)));
}
rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

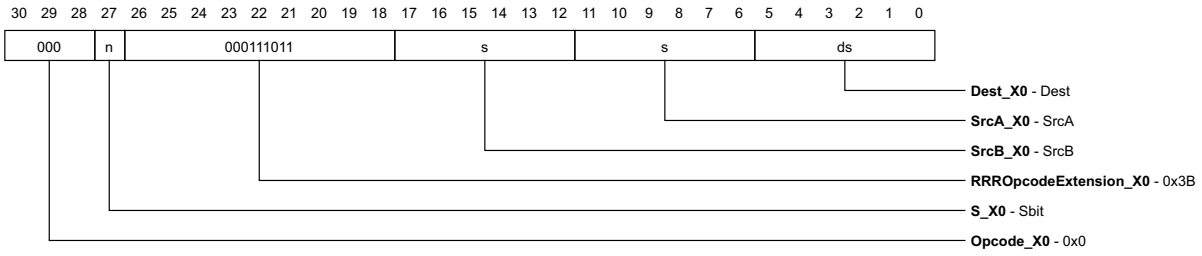

Figure 4-358: sadah in X0 Bit Descriptions

sadah_u: Sum of Absolute Difference Accumulate Unsigned Half Words

Syntax

```
sadah_u Dest, SrcA, SrcB
```

Example

```
sadah_u r5, r6, r7
```

Description

Sum the absolute differences between the pair of half words in the first source operand and the pair of half words in the second source operand and accumulate the sum into the destination register.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output +=
        abs(((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK)-
             ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) &
           HALF_WORD_MASK));
}
rf[Dest] = rf[Dest] + output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

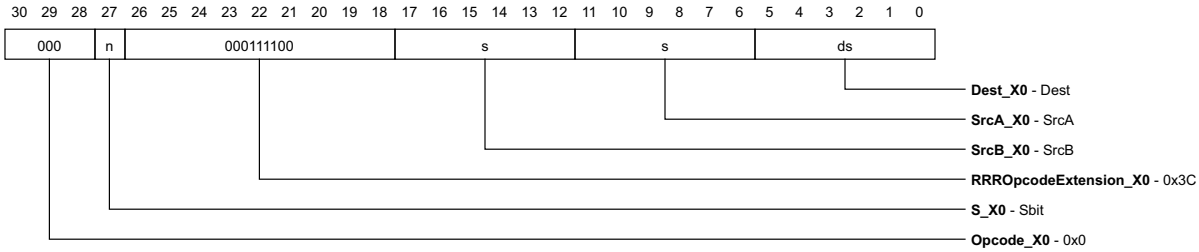

Figure 4-359: sadah_u in X0 Bit Descriptions

sadb_u: Sum of Absolute Difference Unsigned Bytes

Syntax

```
sadb_u Dest, SrcA, SrcB
```

Example

```
sadb_u r5, r6, r7
```

Description

Sum the absolute differences between the four bytes in the first source operand and the four bytes in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   output +=
        abs(((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK) -
            ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

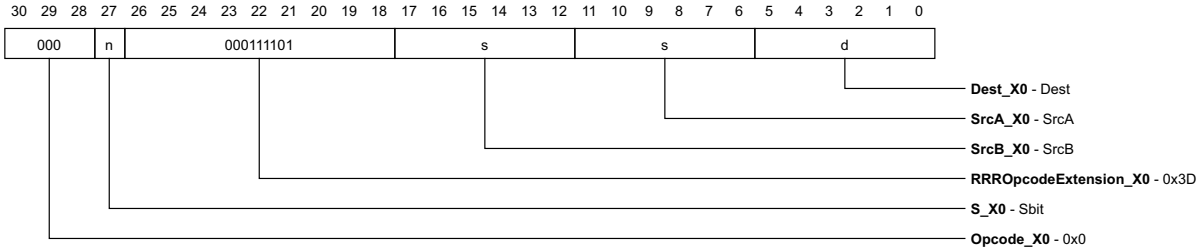

Figure 4-360: sadb_u in X0 Bit Descriptions

sadh: Sum of Absolute Difference Half Words

Syntax

```
sadh Dest, SrcA, SrcB
```

Example

```
sadh r5, r6, r7
```

Description

Sum the absolute differences between the pair of half words in the first source operand and the pair of half words in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output +=
       abs(signExtend16(getHalfWord(rf[SrcA], counter)) -
           signExtend16(getHalfWord(rf[SrcB], counter)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

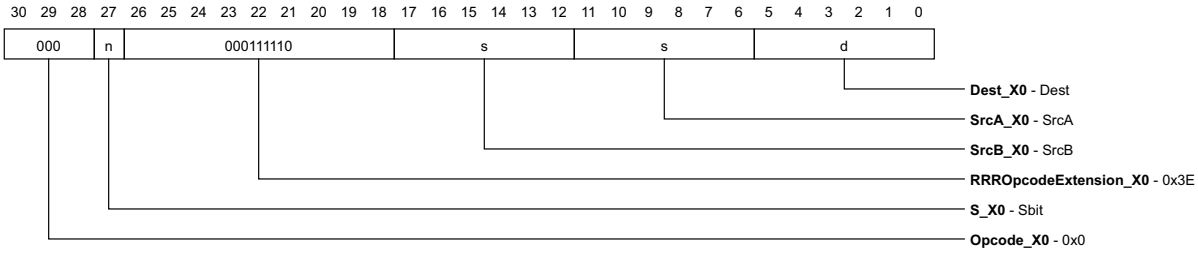

Figure 4-361: sadh in X0 Bit Descriptions

sadh_u: Sum of Absolute Difference Unsigned Half Words

Syntax

```
sadh_u Dest, SrcA, SrcB
```

Example

```
sadh_u r5, r6, r7
```

Description

Sum the absolute differences between the pair of half words in the first source operand and the pair of half words in the second source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output +=
        abs(((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK) -
             ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) &
           HALF_WORD_MASK));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X				

Encoding

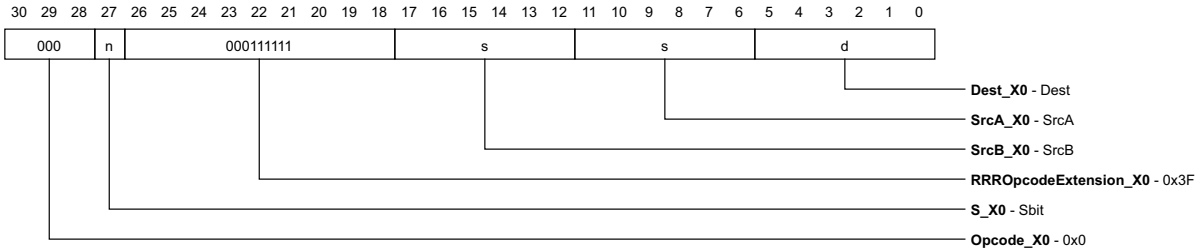

Figure 4-362: sadh_u in X0 Bit Descriptions

seqb: Set Equal to Byte

Syntax

```
seqb Dest, SrcA, SrcB
```

Example

```
seqb 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   int8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
       ((((srca ==
           srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

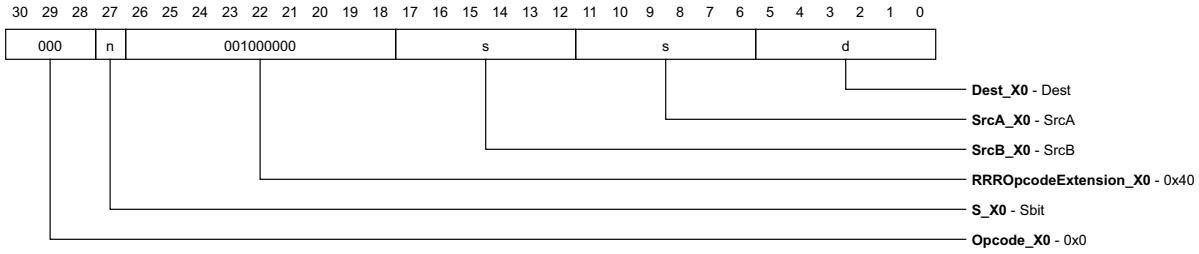

Figure 4-363: seqb in X0 Bit Descriptions

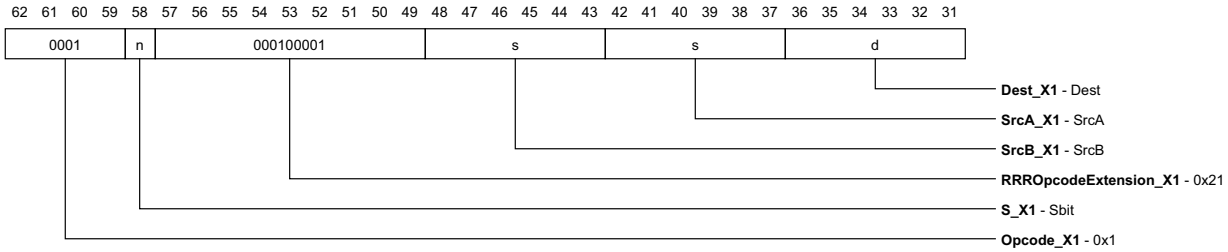

Figure 4-364: seqb in X1 Bit Descriptions

seqh: Set Equal To Half Words

Syntax

```
seqh Dest, SrcA, SrcB
```

Example

```
seqh r5, r6, r7
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is equal to the half word of the second source operand. Otherwise the result is set to 0. This instruction treats both source half words as signed values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb =
      ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
       ((((srca ==
           srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

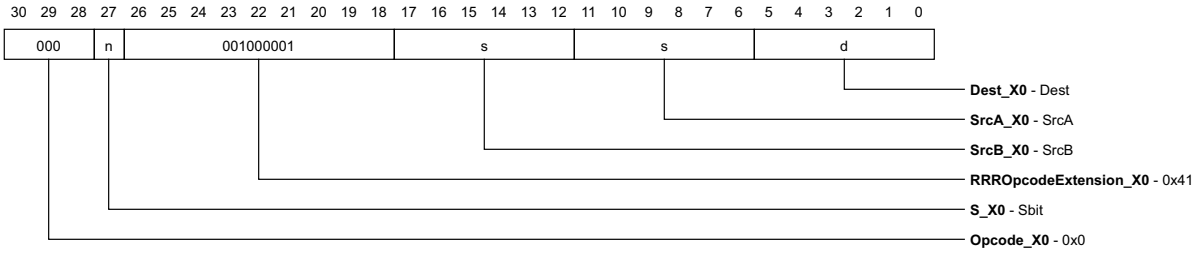

Figure 4-365: seqh in X0 Bit Descriptions

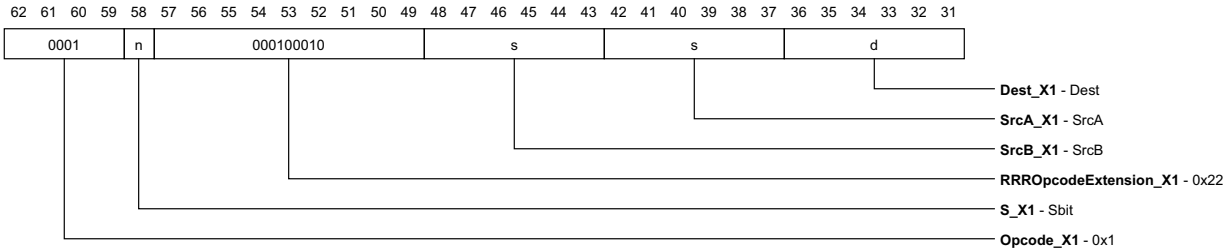

Figure 4-366: seqh in X1 Bit Descriptions

seqib: Set Equal To Immediate Byte

Syntax

```
seqib Dest, SrcA, Imm8
```

Example

```
seqib 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   int8_t srcb = signExtend8(Imm8) & BYTE_MASK;
   output |=
       ((((srca ==
           srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

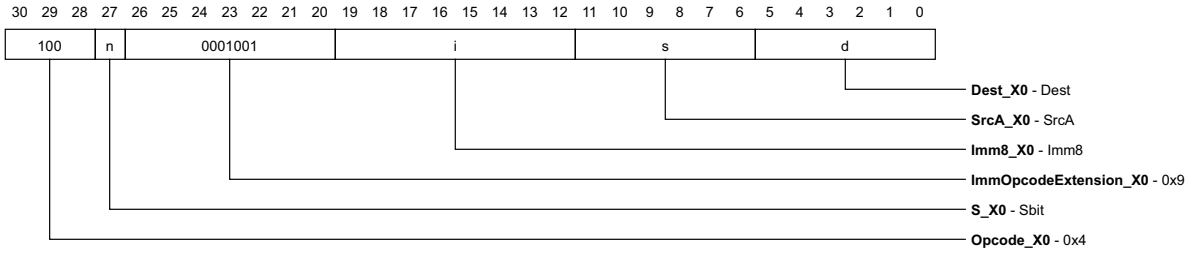

Figure 4-367: seqib in X0 Bit Descriptions

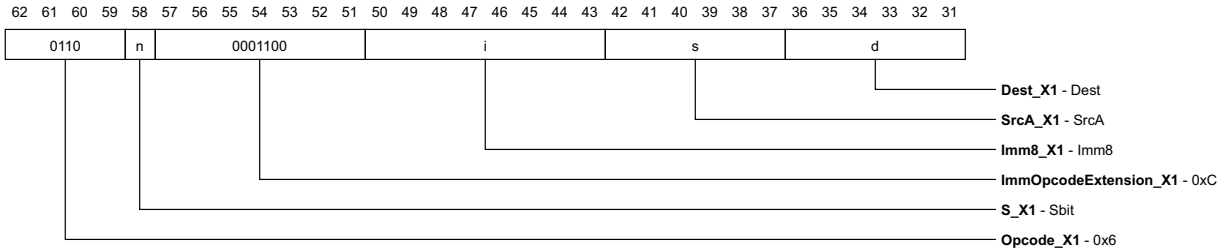

Figure 4-368: seqib in X1 Bit Descriptions

seqih: Set Equal To Immediate Half Words

Syntax

```
seqih Dest, SrcA, Imm8
```

Example

```
seqih r5, r6, 5
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is equal to a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source half words as signed values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   int16_t srcb = signExtend8(Imm8) & HALF_WORD_MASK;
   output |=
       ((((srca ==
           srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

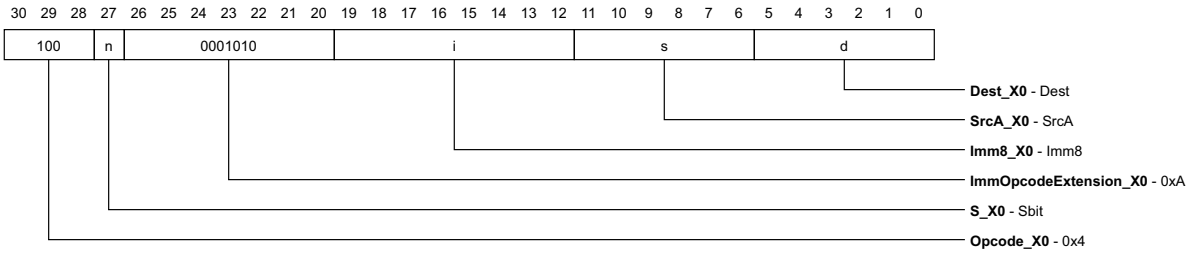

Figure 4-369: seqih in X0 Bit Descriptions

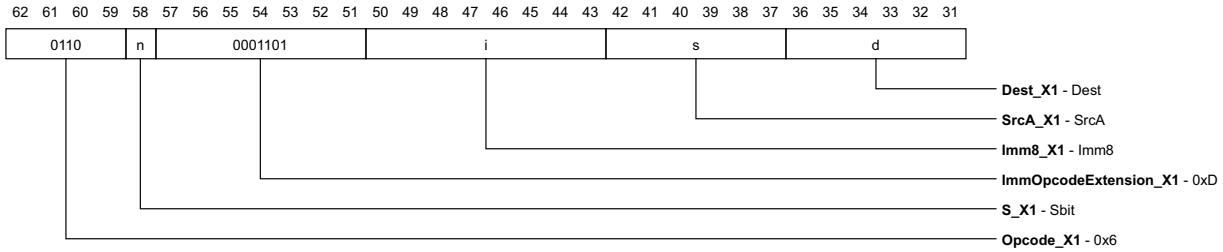

Figure 4-370: seqih in X1 Bit Descriptions

shlb: Logical Shift Left Bytes

Syntax

```
shlb Dest, SrcA, SrcB
```

Example

```
shlb r5, r6, r7
```

Description

Logically shift the four bytes in the first source operand to the left by the second source operand. If the shift amount is larger than the number of bits in a byte, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a byte. Logical left shift shifts zeros into the low ordered bits in a byte.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_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

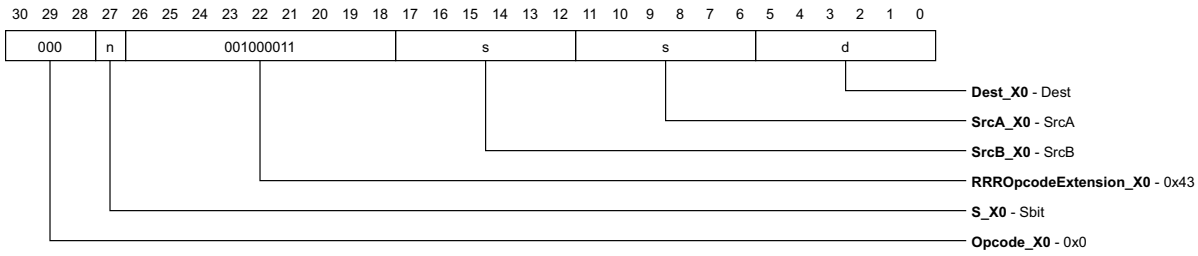

Figure 4-371: shlb in X0 Bit Descriptions

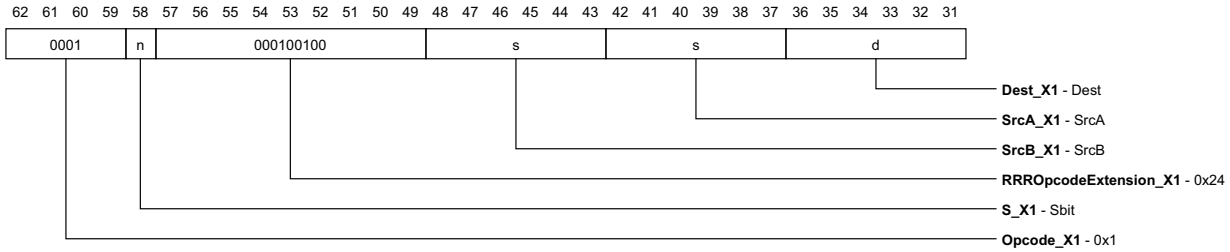

Figure 4-372: shlb in X1 Bit Descriptions

shlh: Logical Shift Left Half Words

Syntax

```
shlh Dest, SrcA, SrcB
```

Example

```
shlh r5, r6, r7
```

Description

Logically shift the pair of half words in the first source operand to the left by the second source operand. If the shift amount is larger than the number of bits in a half word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a half word. Logical left shift shifts zeros into the low ordered bits in a half word.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
   output =
      setHalfWord(output, counter,
                  (getHalfWord(rf[SrcA], counter) <<
                   (((UnsignedMachineWord) rf[SrcB]) % HALF_WORD_SIZE)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

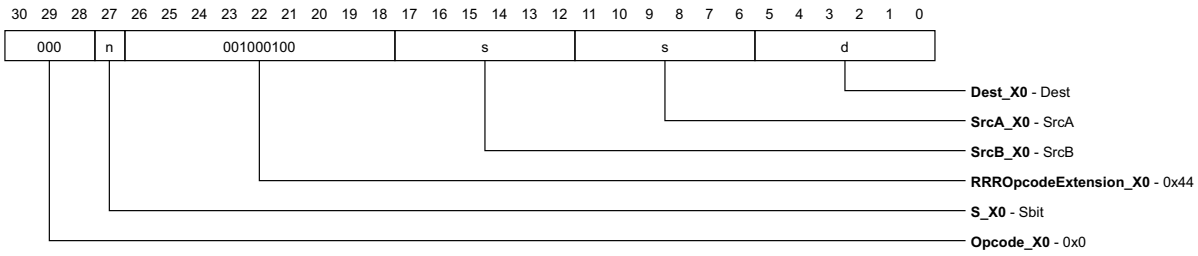

Figure 4-373: shlh in X0 Bit Descriptions

shlib: Logical Shift Left Immediate Bytes

Syntax

```
shlib Dest, SrcA, ShAmt
```

Example

```
shlib r5, r6, 5
```

Description

Logically shift the four bytes in the first source operand to the left by an immediate. If the shift amount is larger than the number of bits in a byte, the effective shift amount is computed to be the specified shift amount 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 2.

Functional Description

```
UnsignedMachineWord 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

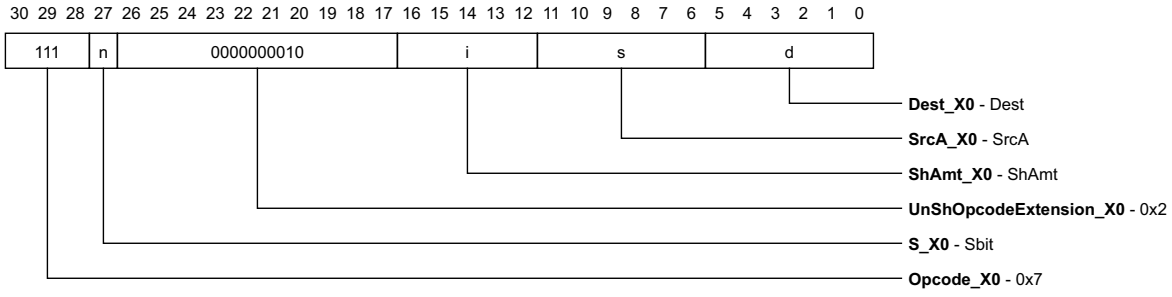

Figure 4-374: shlib in X0 Bit Descriptions

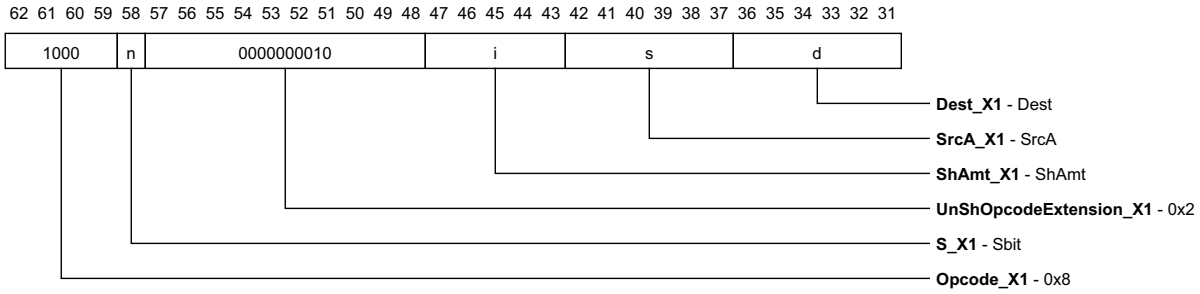

Figure 4-375: shlib in X1 Bit Descriptions

shlih: Logical Shift Left Immediate Half Words

Syntax

```
shlih Dest, SrcA, ShAmt
```

Example

```
shlih r5, r6, 5
```

Description

Logically shift the pair of half words in the first source operand to the left by an immediate. If the shift amount is larger than the number of bits in a half word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a half word. Left shifts shift zeros into the low ordered bits in a half word and are suitable to be used as unsigned multiplication by powers of 2.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
     output =
         setHalfWord(output, counter,
                     (getHalfWord(rf[SrcA], counter) <<
                      (((UnsignedMachineWord) ShAmt) %
                       HALF_WORD_SIZE)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

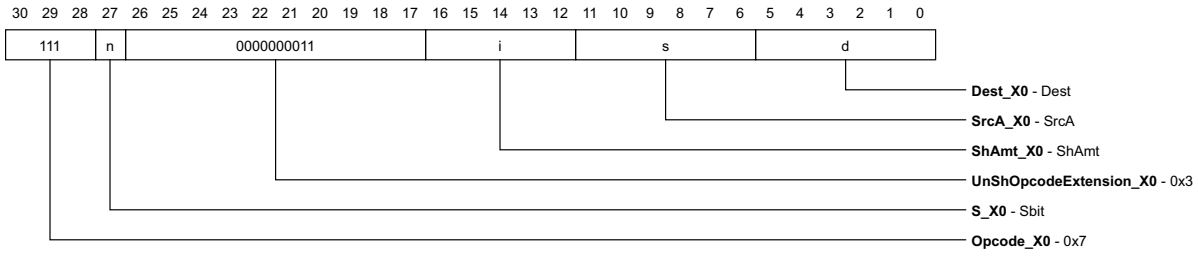

Figure 4-376: shlih in X0 Bit Descriptions

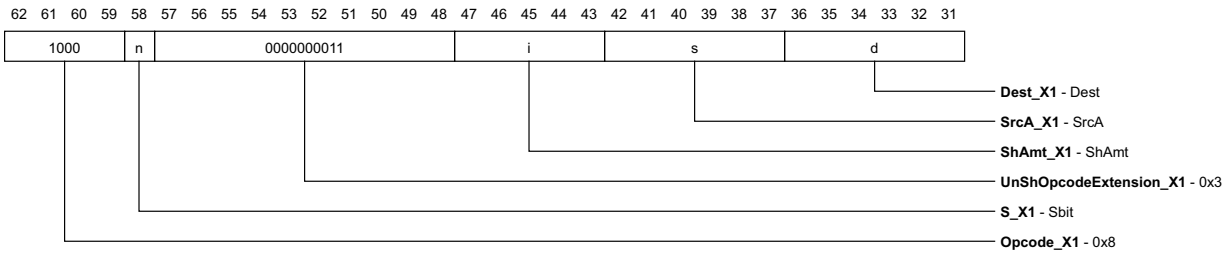

Figure 4-377: shlih in X1 Bit Descriptions

shrb: Logical Shift Right Bytes

Syntax

```
shrb Dest, SrcA, SrcB
```

Example

```
shrb r5, r6, r7
```

Description

Logically shift the four bytes in the first source operand to the right by the second source operand. If the shift amount is larger than the number of bits in a byte, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a byte. Logical right shift shifts zeros into the high ordered bits in a byte.

Functional Description

```
UnsignedMachineWord 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

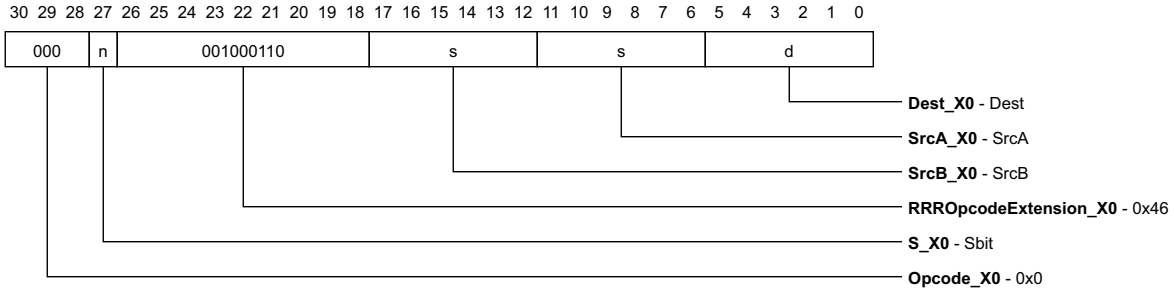

Figure 4-378: shrb in X0 Bit Descriptions

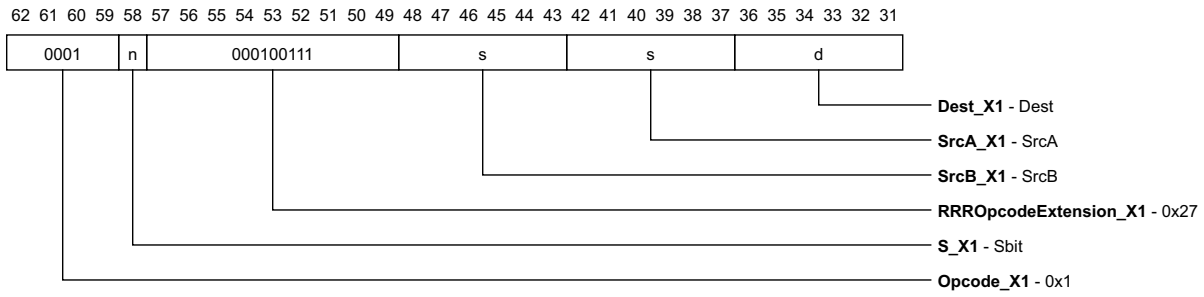

Figure 4-379: shrb in X1 Bit Descriptions

shrh: Logical Shift Right Half Words

Syntax

```
shrh Dest, SrcA, SrcB
```

Example

```
shrh r5, r6, r7
```

Description

Logically shift the pair of half words in the first source operand to the right by the second source operand. If the shift amount is larger than the number of bits in a half word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a half word. Logical right shift shifts zeros into the high ordered bits in a half word.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
     output =
         setHalfWord(output, counter,
                     (getHalfWord(rf[SrcA], counter) >>
                      (((UnsignedMachineWord) rf[SrcB]) %
                       HALF_WORD_SIZE)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

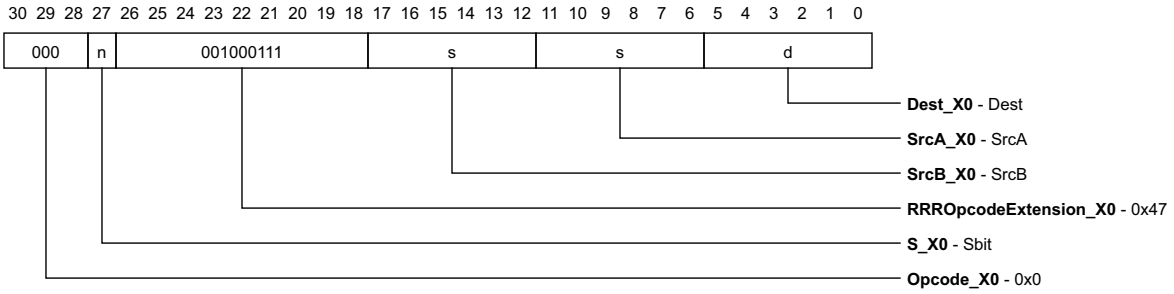

Figure 4-380: shrh in X0 Bit Descriptions

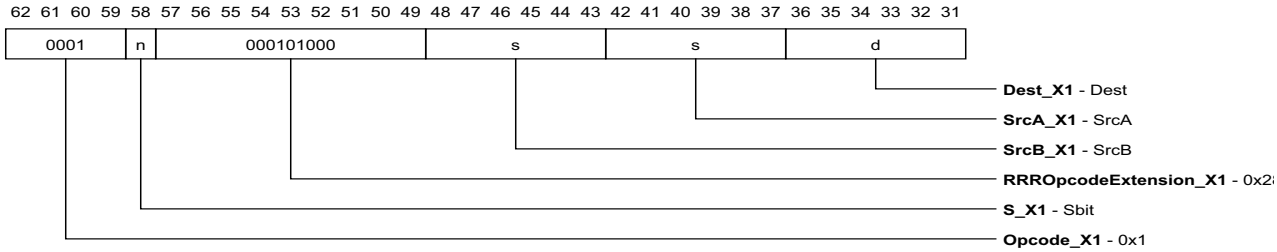

Figure 4-381: shrh in X1 Bit Descriptions

shrib: Logical Shift Right Immediate Bytes

Syntax

```
shrib Dest, SrcA, ShAmt
```

Example

```
shrib r5, r6, 5
```

Description

Logically shift the four bytes in the first source operand to the right by an immediate. If the shift amount is larger than the number of bits in a byte, the effective shift amount is computed to be the specified shift amount 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 2.

Functional Description

```
UnsignedMachineWord 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

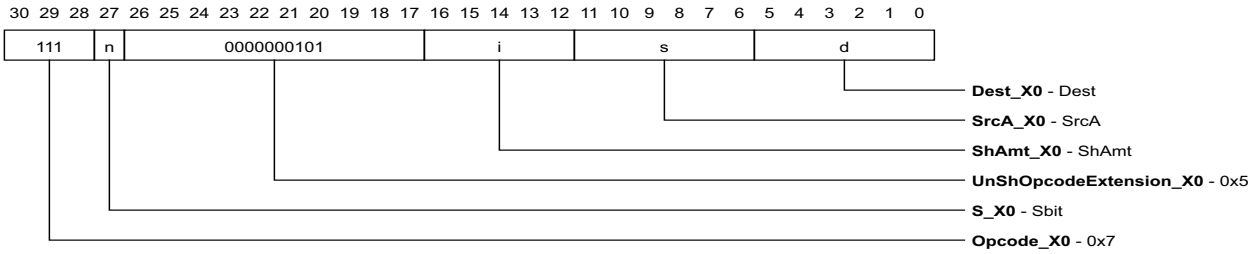

Figure 4-382: shrib in X0 Bit Descriptions

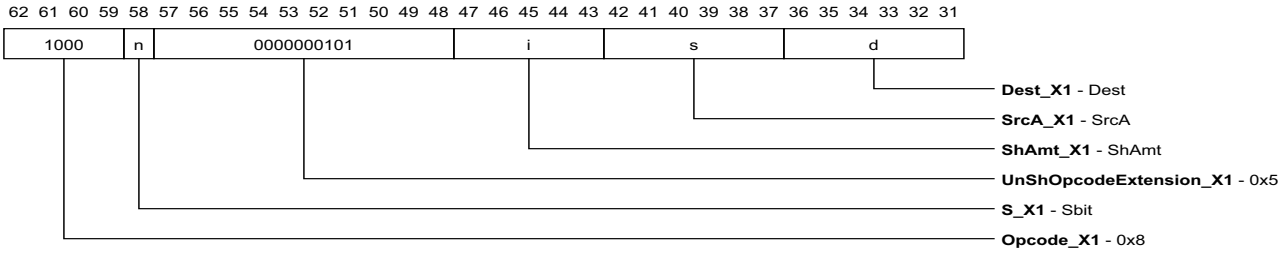

Figure 4-383: shrib in X1 Bit Descriptions

shrih: Logical Shift Right Immediate Half Words

Syntax

```
shrih Dest, SrcA, ShAmt
```

Example

```
shrih r5, r6, 5
```

Description

Logically shift the pair of half words in the first source operand to the right by an immediate. If the shift amount is larger than the number of bits in a half word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a half word. Logical right shifts shift zeros into the high ordered bits in a half word and are suitable to be used as unsigned integer division by powers of 2.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    output =
        setHalfWord(output, counter,
                    (getHalfWord(rf[SrcA], counter) >>
                     (((UnsignedMachineWord) ShAmt) %
                      HALF_WORD_SIZE)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

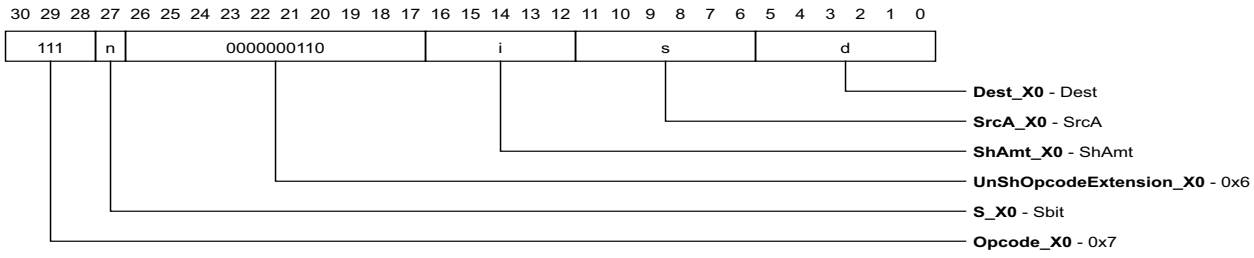

Figure 4-384: shrih in X0 Bit Descriptions

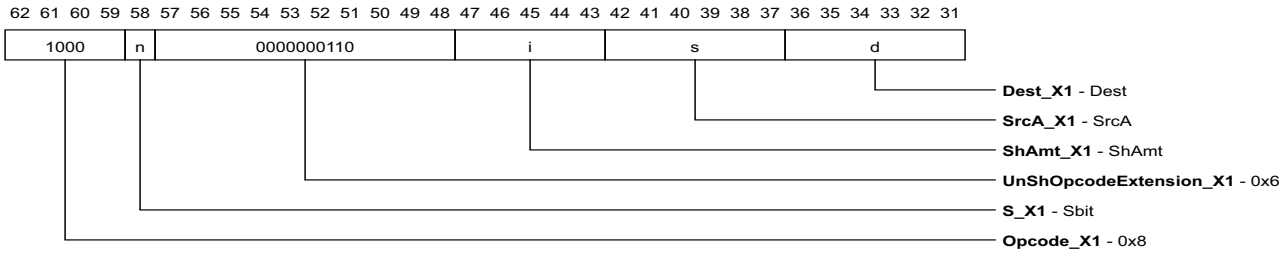

Figure 4-385: shrih in X1 Bit Descriptions

sltb: Set Less Than Byte

Syntax

```
sltb Dest, SrcA, SrcB
```

Example

```
sltb 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   int8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
        ((((srca <
            srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

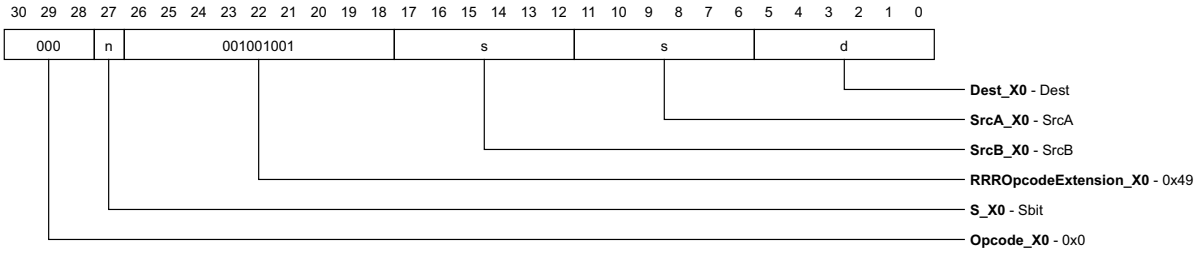

Figure 4-386: sltb in X0 Bit Descriptions

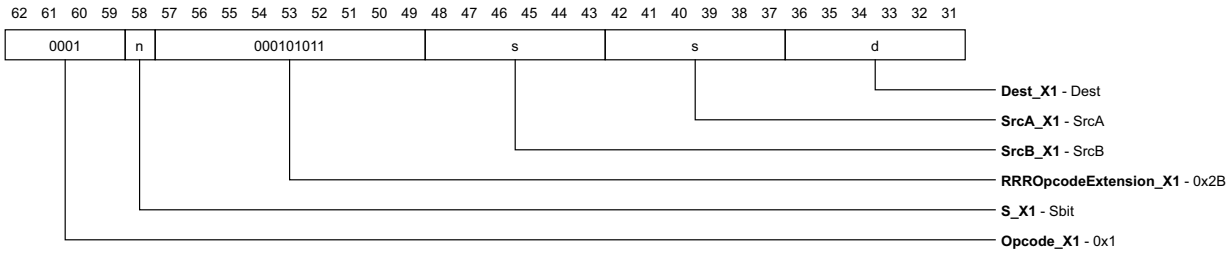

Figure 4-387: sltb in X1 Bit Descriptions

sltb_u: Set Less Than Unsigned Byte

Syntax

```
sltb_u Dest, SrcA, SrcB
```

Example

```
sltb u 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   uint8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   uint8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
       ((((srca <
           srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

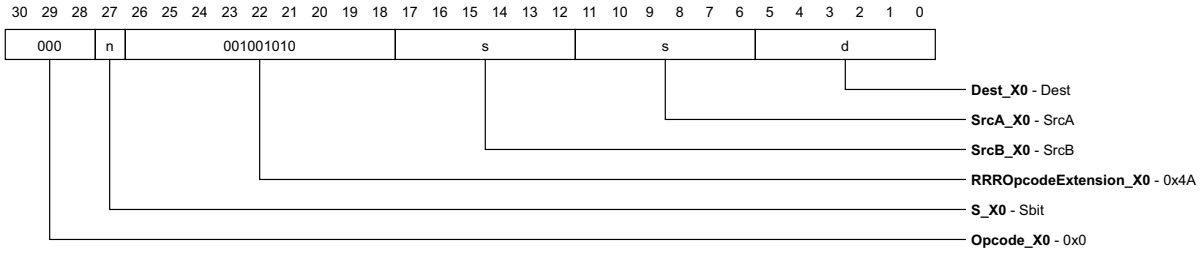

Figure 4-388: sltb_u in X0 Bit Descriptions

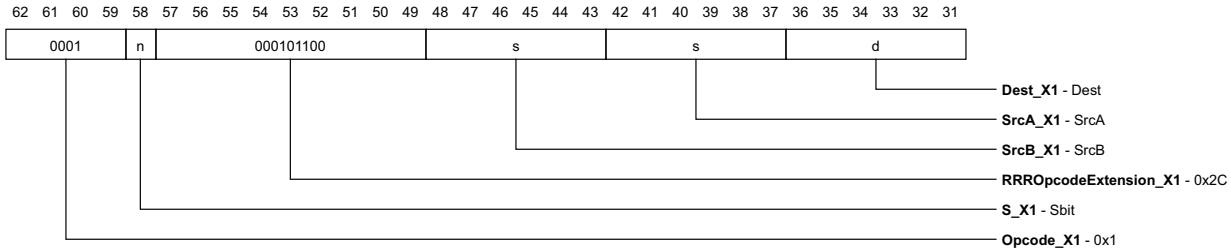

Figure 4-389: sltb_u in X1 Bit Descriptions

slteb: Set Less Than or Equal Byte

Syntax

```
slteb Dest, SrcA, SrcB
```

Example

```
slteb 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
    int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
    int8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
       ((((srca <=
           srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

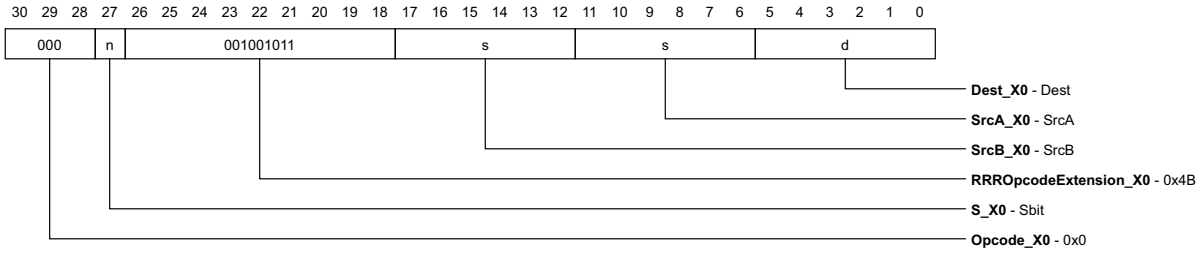

Figure 4-390: slteb in X0 Bit Descriptions

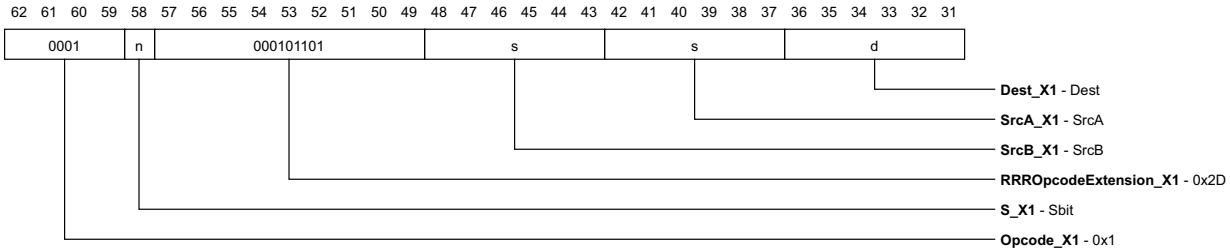

Figure 4-391: slteb in X1 Bit Descriptions

slteb_u: Set Less Than or Equal Unsigned Byte

Syntax

```
slteb_u Dest, SrcA, SrcB
```

Example

```
slteb_u 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
   uint8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   uint8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
   output |=
       ((((srca <=
           srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

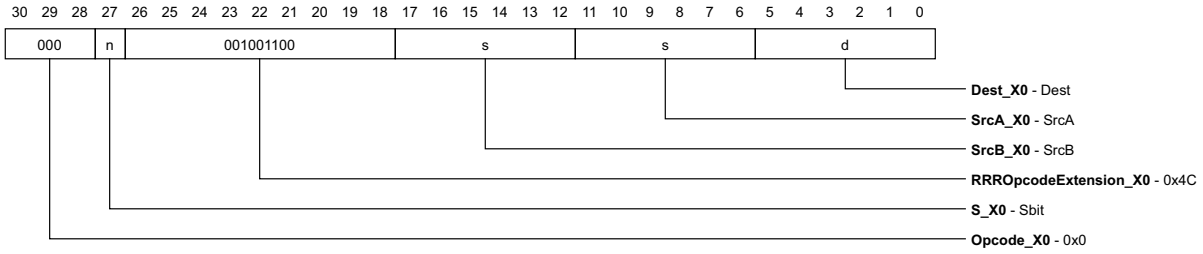

Figure 4-392: slteb_u in X0 Bit Descriptions

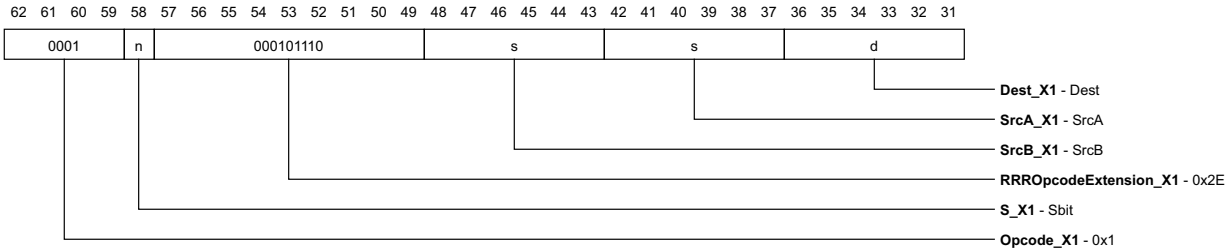

Figure 4-393: slteb_u in X1 Bit Descriptions

slteh: Set Less Than or Equal Half Words

Syntax

```
slteh Dest, SrcA, SrcB
```

Example

```
slteh r5, r6, r7
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is less than or equal to the half word of the second source operand. Otherwise the result is set to 0. This instruction treats both source half words as signed values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    int16_t srca =
      ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    int16_t srcb =
      ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    output |=
       ((((srca <=
           srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

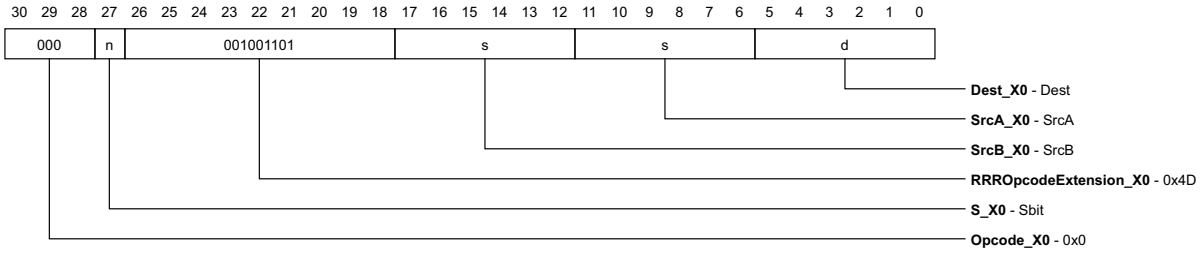

Figure 4-394: slteh in X0 Bit Descriptions

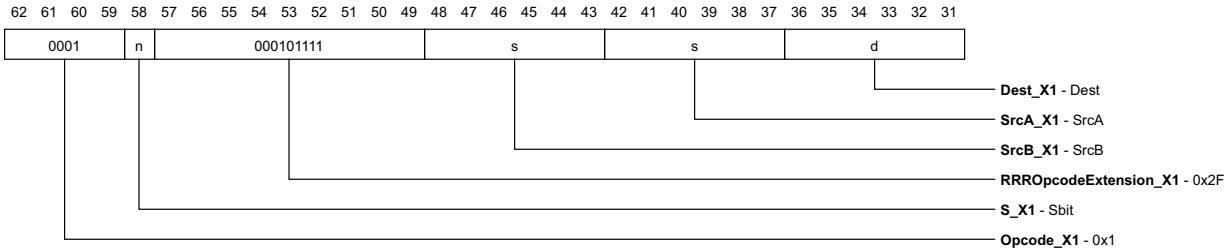

Figure 4-395: slteh in X1 Bit Descriptions

slteh_u: Set Less Than or Equal Unsigned Half Words

Syntax

```
slteh_u Dest, SrcA, SrcB
```

Example

```
slteh_u r5, r6, r7
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is less than or equal to the half word of the second source operand. Otherwise the result is set to 0. This instruction treats both source half words as unsigned values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    uint16_t srca =
       ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    uint16_t srcb =
       ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    output |=
       ((((srca <=
          srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                               HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

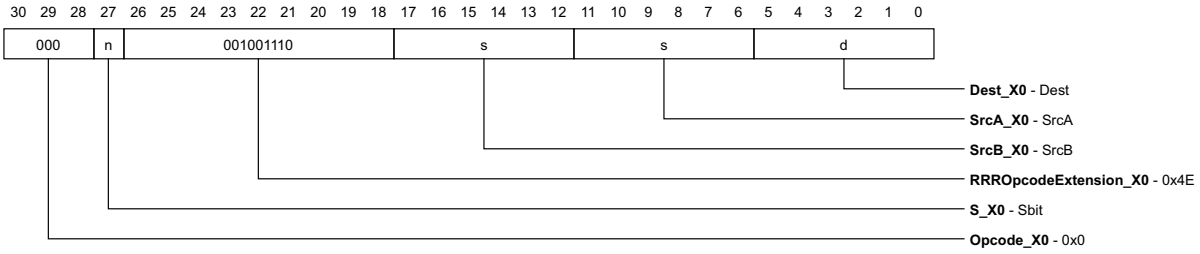

Figure 4-396: slteh_u in X0 Bit Descriptions

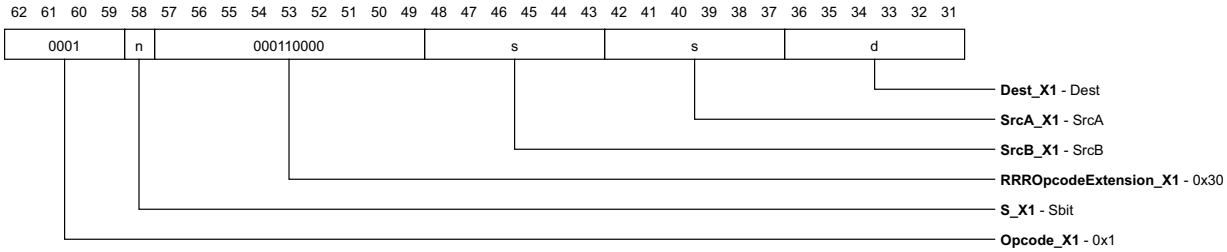

Figure 4-397: slteh_u in X1 Bit Descriptions

slth: Set Less Than Half Words

Syntax

```
slth Dest, SrcA, SrcB
```

Example

```
slth r5, r6, r7
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is less than the half word of the second source operand. Otherwise the result is set to 0. This instruction treats both source half words as signed values

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    int16_t srca =
        ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    int16_t srcb =
        ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
   output |=
        ((((srca <
            srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                 HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

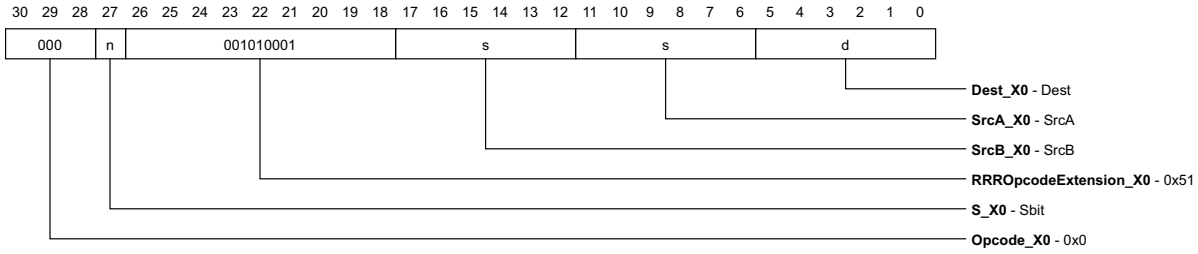

Figure 4-398: slth in X0 Bit Descriptions

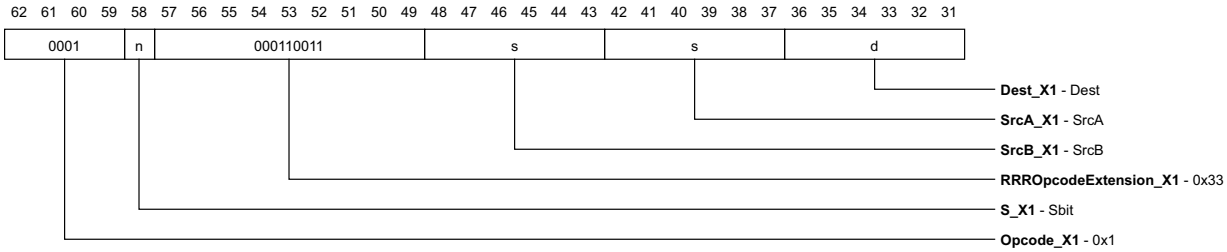

Figure 4-399: slth in X1 Bit Descriptions

slth_u: Set Less Than Unsigned Half Words

Syntax

```
slth_u Dest, SrcA, SrcB
```

Example

```
slth_u r5, r6, r7
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is less than the half word of the second source operand. Otherwise the result is set to 0. This instruction treats both source half words as unsigned values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    uint16_t srca =
        ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    uint16_t srcb =
        ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    output |=
        ((((srca <
            srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                 HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

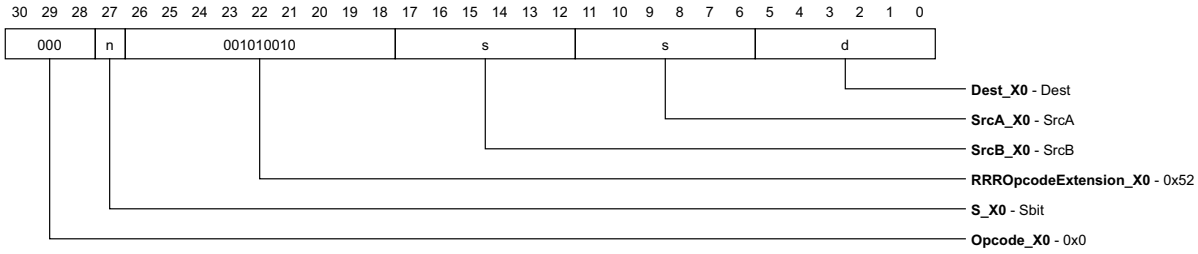

Figure 4-400: slth_u in X0 Bit Descriptions

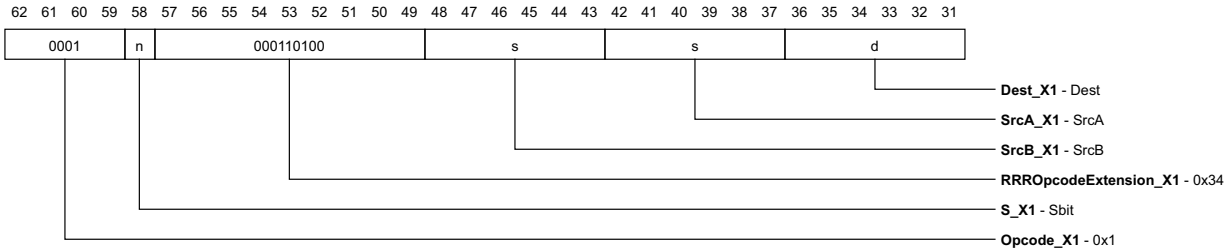

Figure 4-401: slth_u in X1 Bit Descriptions

sltib: Set Less Than Immediate Byte

Syntax

```
sltib Dest, SrcA, Imm8
```

Example

```
sltib 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
    int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
    int8_t srcb = signExtend8(Imm8) & BYTE_MASK;
    output |=
        ((((srca <
            srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

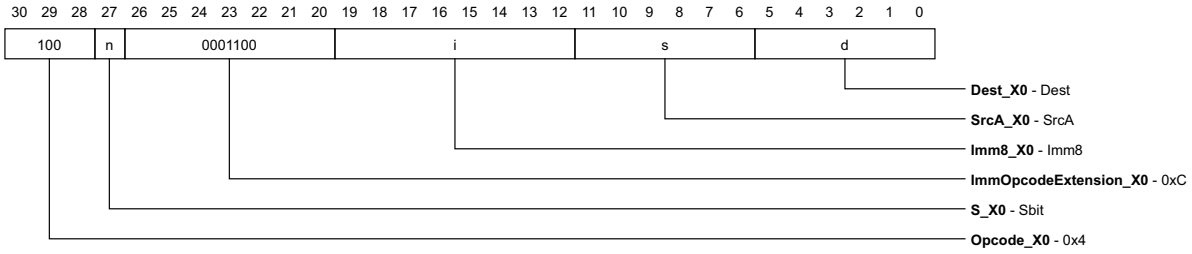

Figure 4-402: sltib in X0 Bit Descriptions

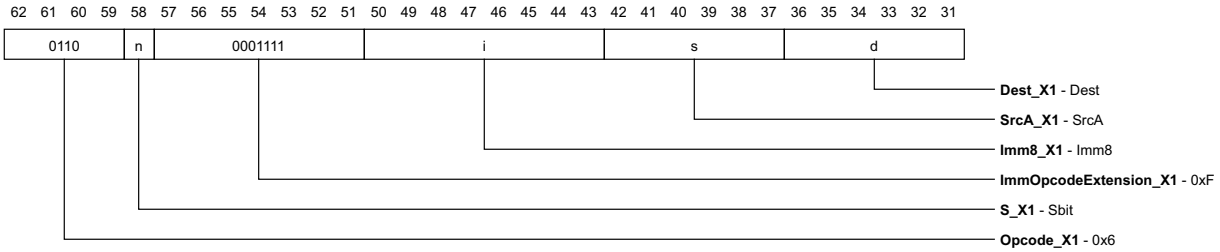

Figure 4-403: sltib in X1 Bit Descriptions

sltib_u: Set Less Than Unsigned Immediate Byte

Syntax

```
sltib_u Dest, SrcA, Imm8
```

Example

```
sltib_u 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
    uint8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
    uint8_t srcb = signExtend8(Imm8) & BYTE_MASK;
    output |=
        ((((srca <
            srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

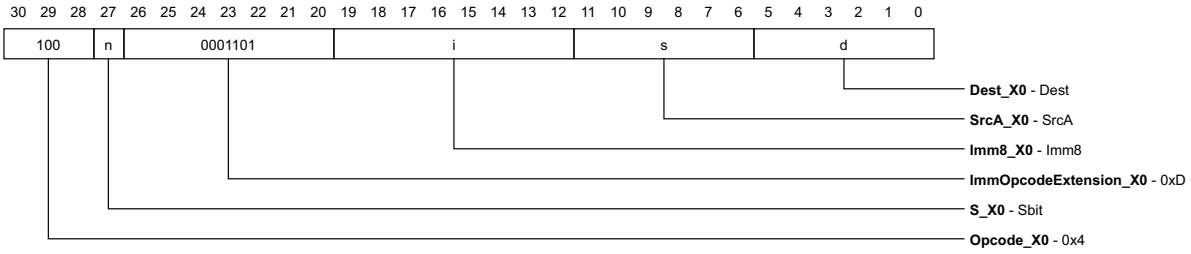

Figure 4-404: sltib_u in X0 Bit Descriptions

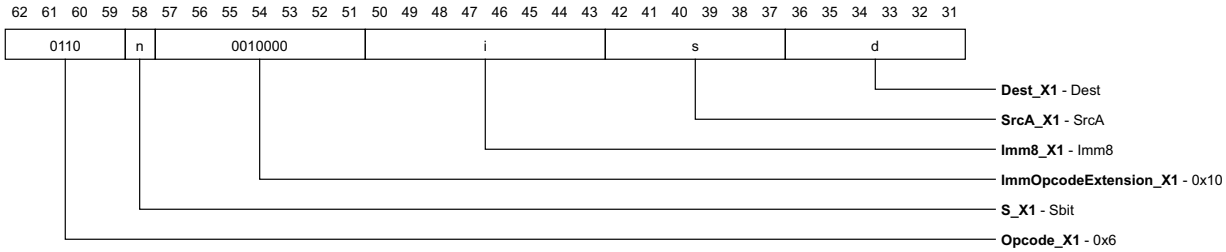

Figure 4-405: sltib_u in X1 Bit Descriptions

sltih: Set Less Than Immediate Half Words

Syntax

```
sltih Dest, SrcA, Imm8
```

Example

```
sltih r5, r6, 5
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source half words as signed values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    int16_t srca =
       ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    int16_t srcb = signExtend8(Imm8) & HALF_WORD_MASK;
    output |=
       ((((srca <
           srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

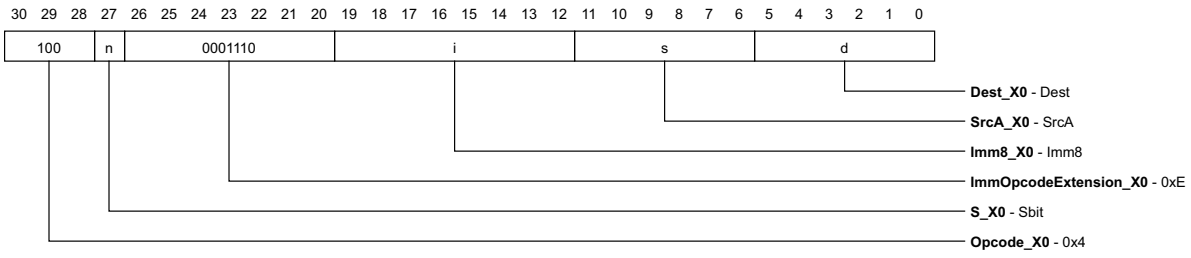

Figure 4-406: sltih in X0 Bit Descriptions

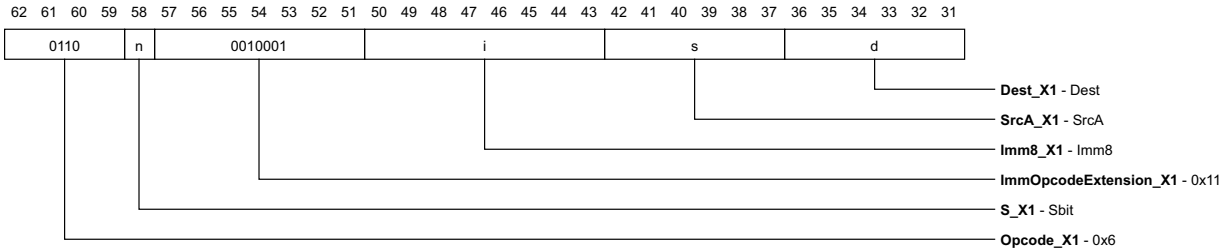

Figure 4-407: sltih in X1 Bit Descriptions

sltih_u: Set Less Than Unsigned Immediate Half Words

Syntax

```
sltih_u Dest, SrcA, Imm8
```

Example

```
sltih_u r5, r6, 5
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is less than a sign extended immediate. Otherwise the result is set to 0. This instruction treats both source half words as unsigned values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    uint16_t srca =
        ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    uint16_t srcb = signExtend8(Imm8) & HALF_WORD_MASK;
    output |=
        ((((srca <
            srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                 HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

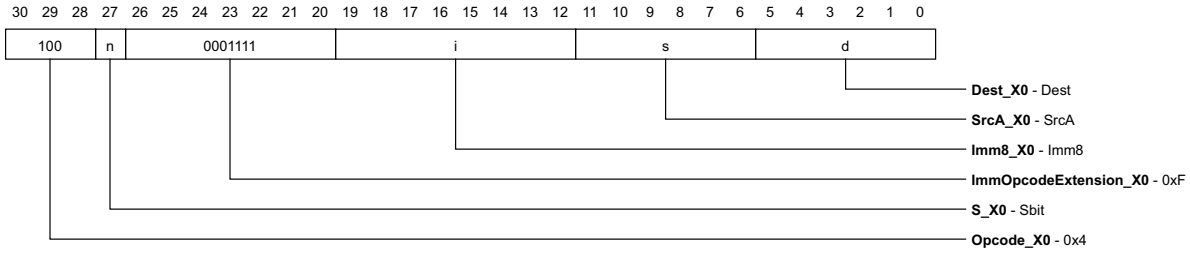

Figure 4-408: sltih_u in X0 Bit Descriptions

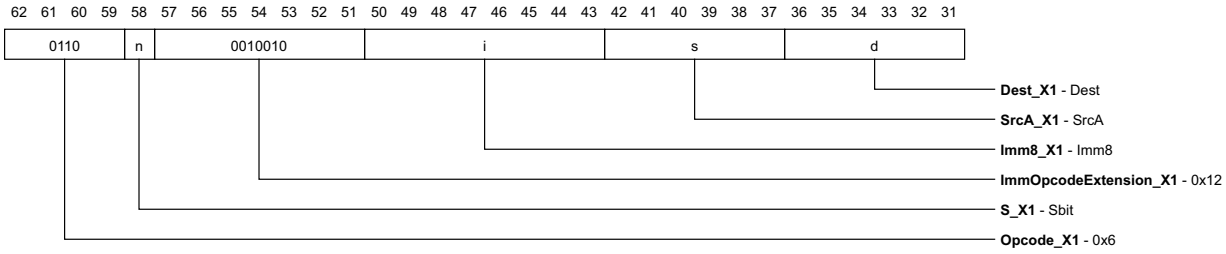

Figure 4-409: sltih_u in X1 Bit Descriptions

sneb: Set Not Equal To Byte

Syntax

```
sneb Dest, SrcA, SrcB
```

Example

```
sneb 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

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / BYTE_SIZE); counter++) {
    int8_t srca = ((rf[SrcA] >> (counter * BYTE_SIZE)) & BYTE_MASK);
    int8_t srcb = ((rf[SrcB] >> (counter * BYTE_SIZE)) & BYTE_MASK);
    output |=
        ((((srca !=
            srcb) ? 1 : 0) & BYTE_MASK) << (counter * BYTE_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

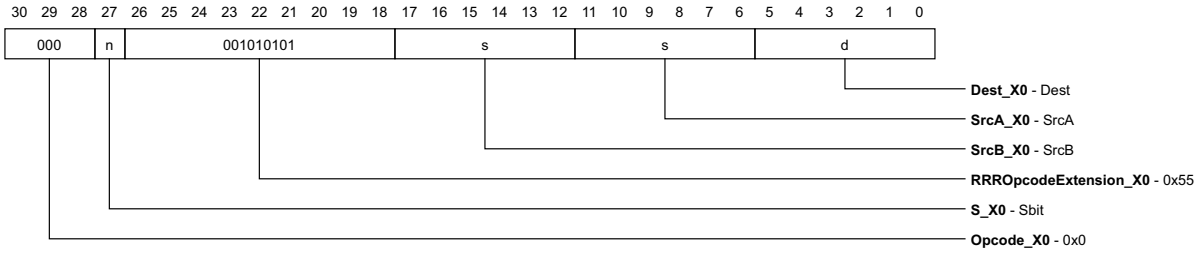

Figure 4-410: sneb in X0 Bit Descriptions

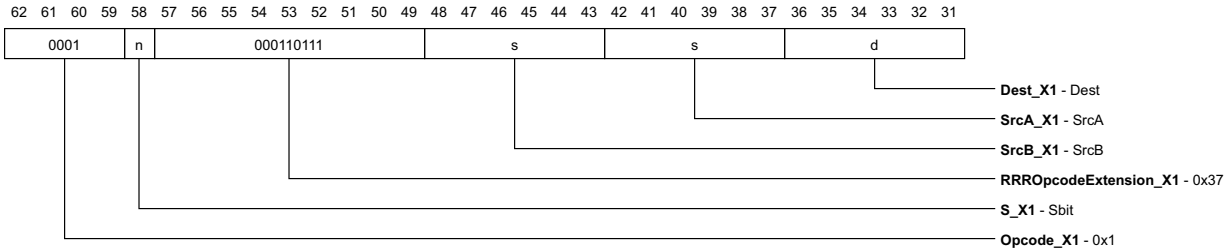

Figure 4-411: sneb in X1 Bit Descriptions

sneh: Set Not Equal To Half Words

Syntax

```
sneh Dest, SrcA, SrcB
```

Example

```
sneh r5, r6, r7
```

Description

Sets each result half word to 1 if the corresponding half word of the first source operand is not equal to the half word of the second source operand. Otherwise the result is set to 0. This instruction treats both source half words as signed values.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    int16_t srca =
       ((rf[SrcA] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    int16_t srcb =
       ((rf[SrcB] >> (counter * HALF_WORD_SIZE)) & HALF_WORD_MASK);
    output |=
        ((((srca !=
            srcb) ? 1 : 0) & HALF_WORD_MASK) << (counter *
                                                 HALF_WORD_SIZE));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

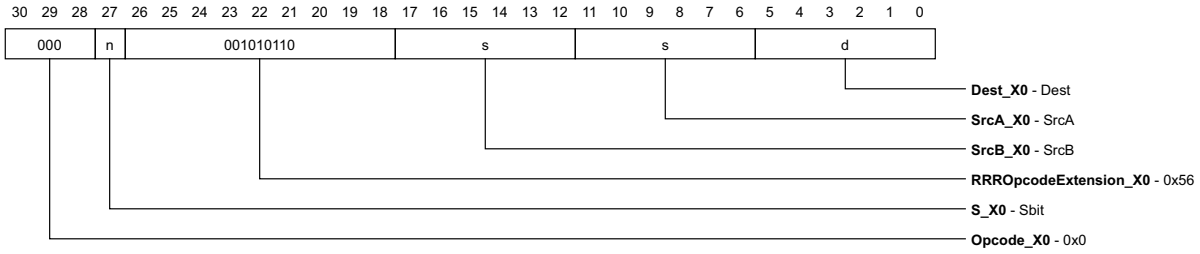

Figure 4-412: sneh in X0 Bit Descriptions

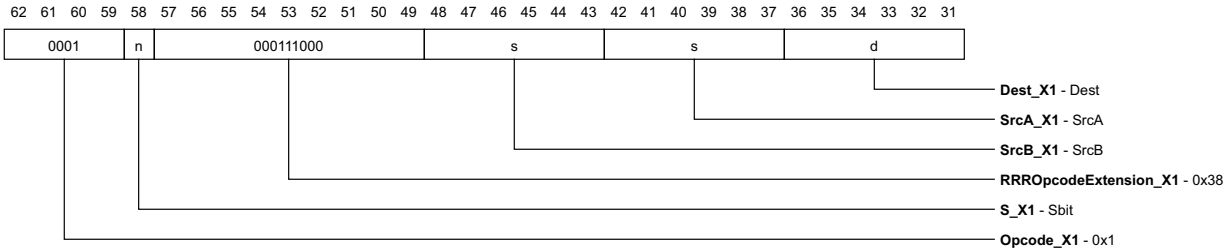

Figure 4-413: sneh in X1 Bit Descriptions

srab: Arithmetic Shift Right Bytes

Syntax

```
srab Dest, SrcA, SrcB
```

Example

```
srab r5, r6, r7
```

Description

Arithmetically shift the four bytes in the first source operand to the right by the second source operand. If the shift amount is larger than the number of bits in a byte, the effective shift amount is computed to be the specified shift amount 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

```
UnsignedMachineWord 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

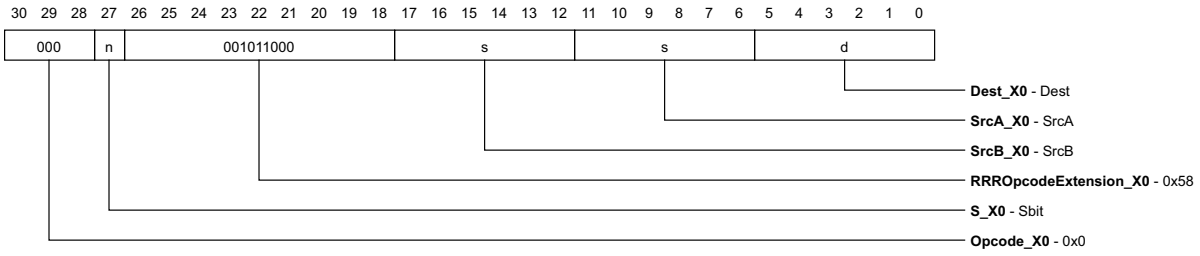

Figure 4-414: srab in X0 Bit Descriptions

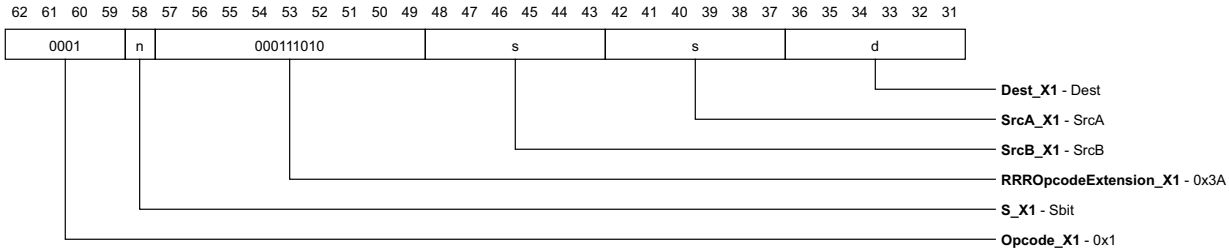

Figure 4-415: srab in X1 Bit Descriptions

srah: Arithmetic Shift Right Half Words

Syntax

```
srah Dest, SrcA, SrcB
```

Example

```
srah r5, r6, r7
```

Description

Arithmetically shift the pair of half words in the first source operand to the right by the second source operand. If the shift amount is larger than the number of bits in a half word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a half word. Arithmetic right shift shifts the high ordered bit into the high ordered bits in a half word.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    output =
        setHalfWord(output, counter,
                    (signExtend16(getHalfWord(rf[SrcA], counter)) >>
                     (((UnsignedMachineWord) rf[SrcB]) %
                      HALF_WORD_SIZE)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

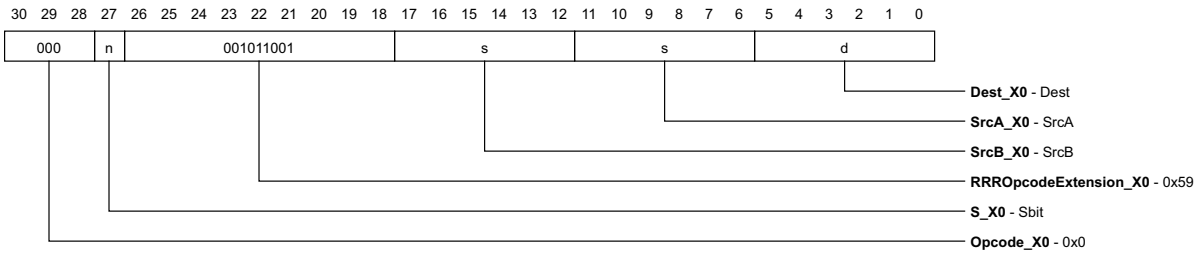

Figure 4-416: srah in X0 Bit Descriptions

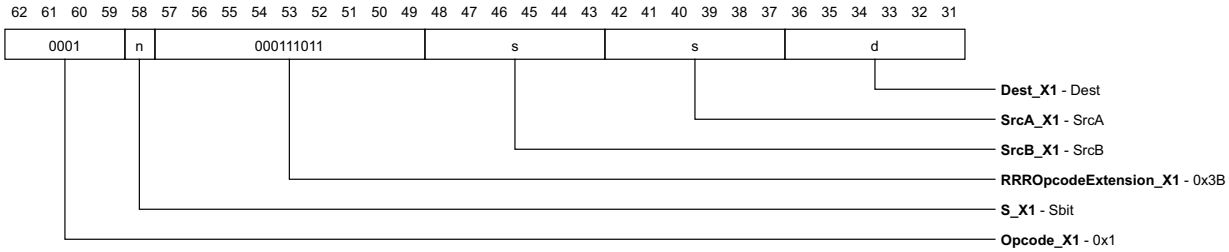

Figure 4-417: srah in X1 Bit Descriptions

sraib: Arithmetic Shift Right Immediate Bytes

Syntax

```
sraib Dest, SrcA, ShAmt
```

Example

```
sraib r5, r6, 5
```

Description

Arithmetically shift the four bytes in the first source operand to the right by an immediate. If the shift amount is larger than the number of bits in a byte, the effective shift amount is computed to be the specified shift amount 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

```
UnsignedMachineWord 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

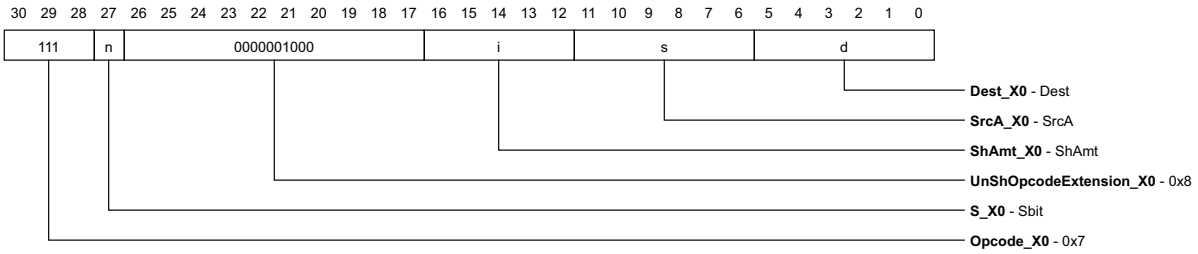

Figure 4-418: sraib in X0 Bit Descriptions

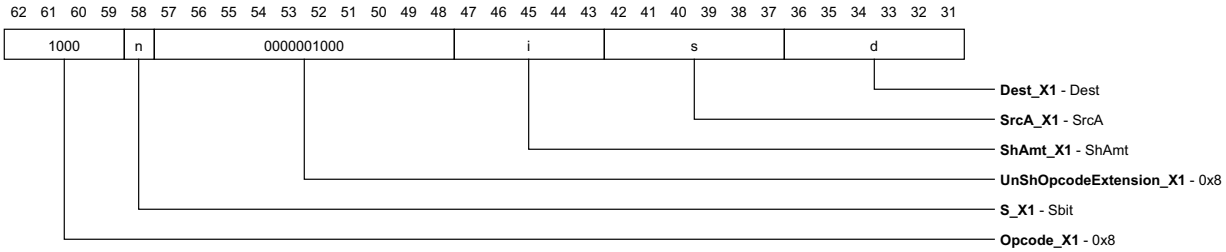

Figure 4-419: sraib in X1 Bit Descriptions

sraih: Arithmetic Shift Right Immediate Half Words

Syntax

```
sraih Dest, SrcA, ShAmt
```

Example

```
sraih r5, r6, 5
```

Description

Arithmetically shift pair of half words in the first source operand to the right by an immediate. If the shift amount is larger than the number of bits in a half word, the effective shift amount is computed to be the specified shift amount modulo the number of bits in a half word. Arithmetic right shifts shift the high ordered bit into the high ordered bits in a half word.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    output =
        setHalfWord(output, counter,
                    (signExtend16(getHalfWord(rf[SrcA], counter)) >>
                     (((UnsignedMachineWord) ShAmt) %
                      HALF_WORD_SIZE)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

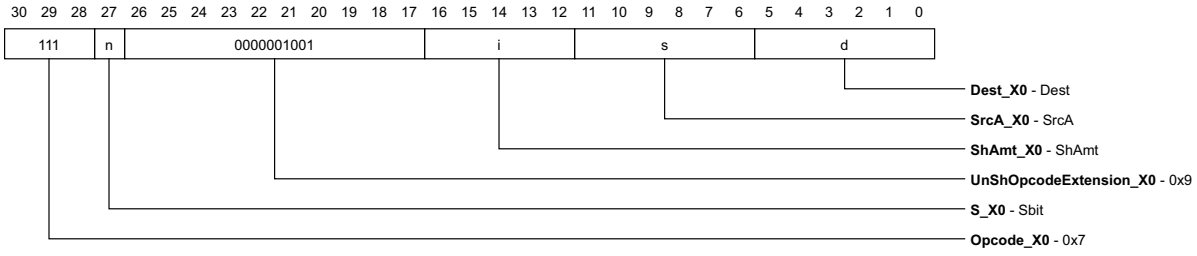

Figure 4-420: sraih in X0 Bit Descriptions

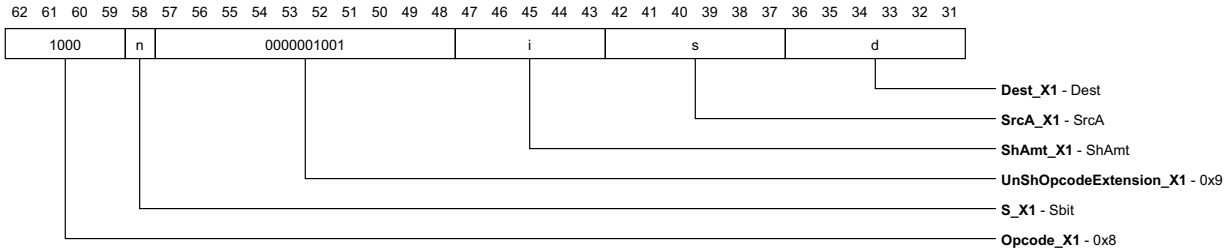

Figure 4-421: sraih in X1 Bit Descriptions

subb: Subtract Bytes

Syntax

```
subb Dest, SrcA, SrcB
```

Example

```
subb r5, r6, r7
```

Description

Subtract the four bytes in the second source operand from the four bytes in the first source operand.

Functional Description

```
UnsignedMachineWord 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

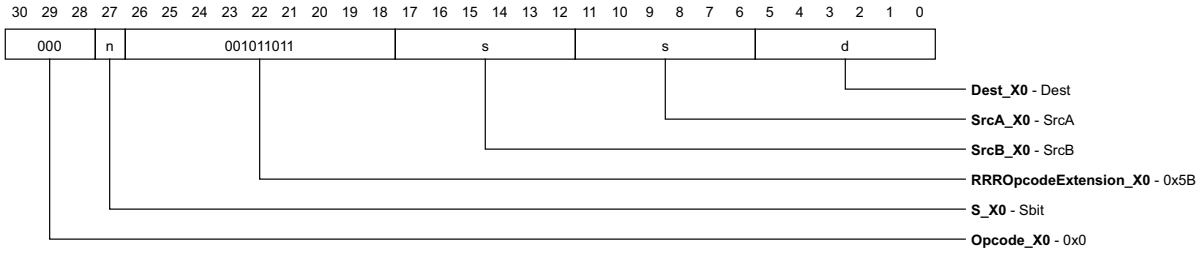

Figure 4-422: subb in X0 Bit Descriptions

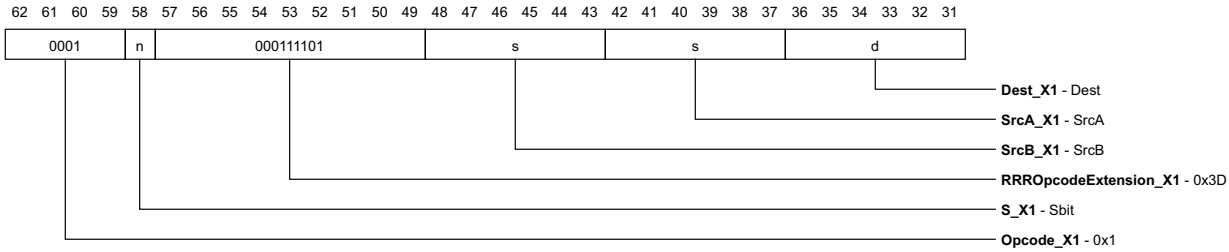

Figure 4-423: subb in X1 Bit Descriptions

subbs_u: Subtract Bytes Saturating Unsigned

Syntax

```
subbs_u Dest, SrcA, SrcB
```

Example

```
subbs_u r5, r6, r7
```

Description

Subtract the four bytes in the second source operand from the four bytes in the first source operand and saturate each result to 0 or the maximum positive value.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

```
UnsignedMachineWord 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

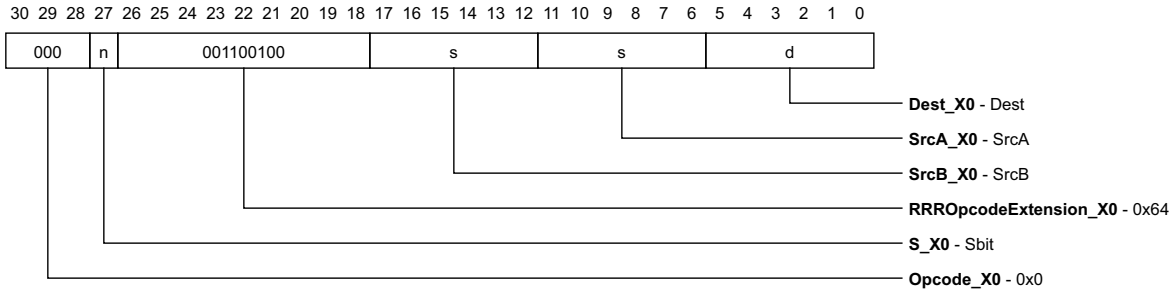

Figure 4-424: subbs_u in X0 Bit Descriptions

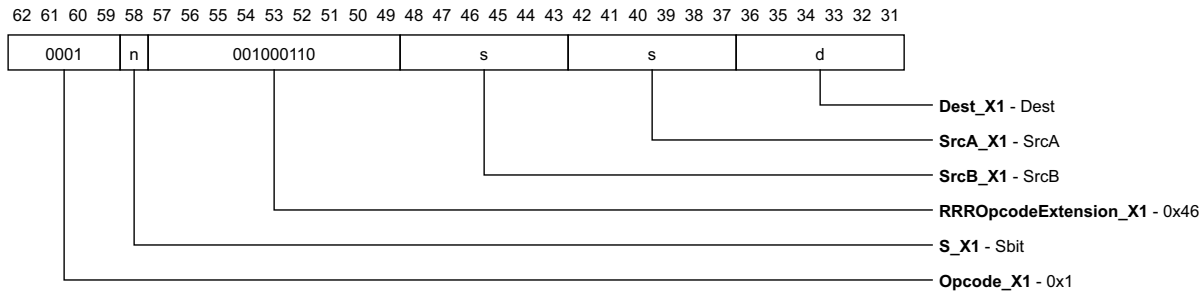

Figure 4-425: subbs_u in X1 Bit Descriptions

subh: Subtract Half Words

Syntax

```
subh Dest, SrcA, SrcB
```

Example

```
subh r5, r6, r7
```

Description

Subtract the pair of half words in the second source operand from the pair of half words in the first source operand.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    output =
        setHalfWord(output, counter,
                    (getHalfWord(rf[SrcA], counter) -
                     getHalfWord(rf[SrcB], counter)));
}
rf[Dest] = output;
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

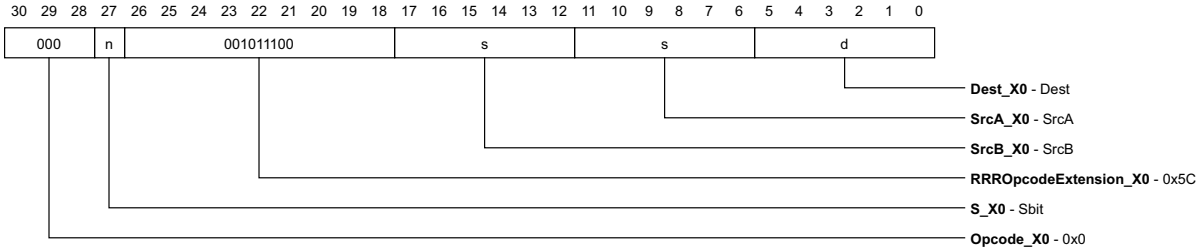

Figure 4-426: subh in X0 Bit Descriptions

subhs: Subtract Half Words Saturating

Syntax

```
subhs Dest, SrcA, SrcB
```

Example

```
subhs r5, r6, r7
```

Description

Subtract the pair of half words in the second source operand from the pair of half words in the first source operand and saturate each result to the minimum negative value or maximum positive value.

NOTE: This instruction is only supported in the TILE*Pro* family of products.

Functional Description

```
UnsignedMachineWord output = 0;
uint32_t counter;
for (counter = 0; counter < (WORD_SIZE / HALF_WORD_SIZE); counter++) {
    output =
        setHalfWord(output, counter,
                    signed_saturate16(signExtend16
                                      (getHalfWord(rf[SrcA], counter)) -
                                      signExtend16(getHalfWord
                                                   (rf[SrcB],
                                                    counter))));
}
rf[Dest] = output
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
X	X			

Encoding

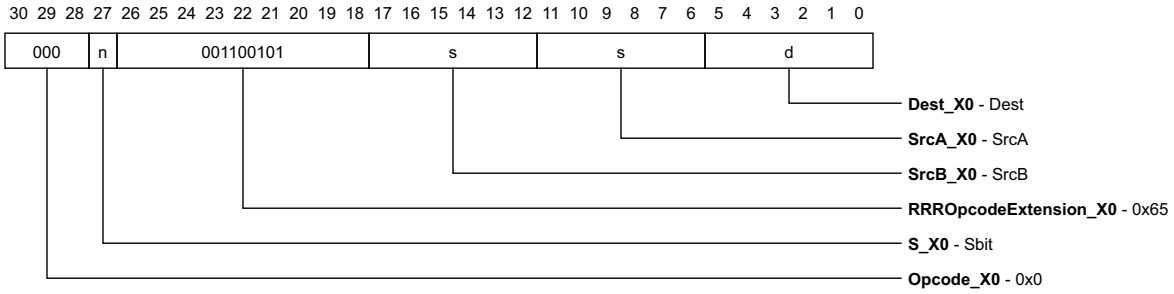

Figure 4-427: subhs in X0 Bit Descriptions

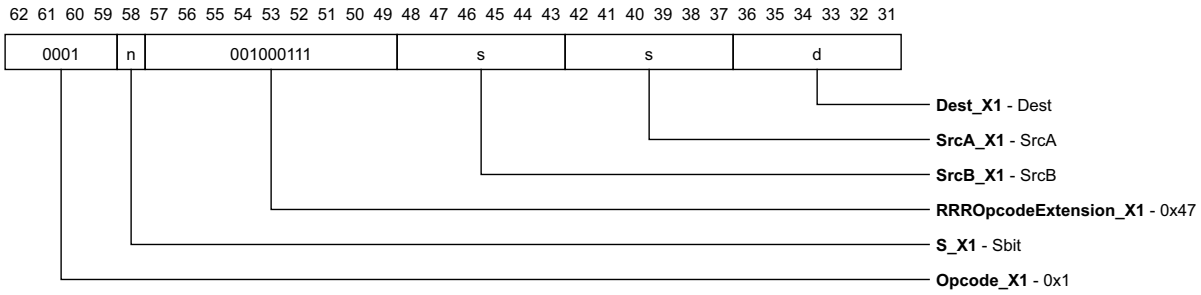

Figure 4-428: subhs in X1 Bit Descriptions

4.1.14 System Instructions

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 to Special Purpose Register Word
- nap: Nap
- swint0: Software Interrupt 0
- swint1: Software Interrupt 1
- swint2: Software Interrupt 2
- swint3: Software Interrupt 3

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 and 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

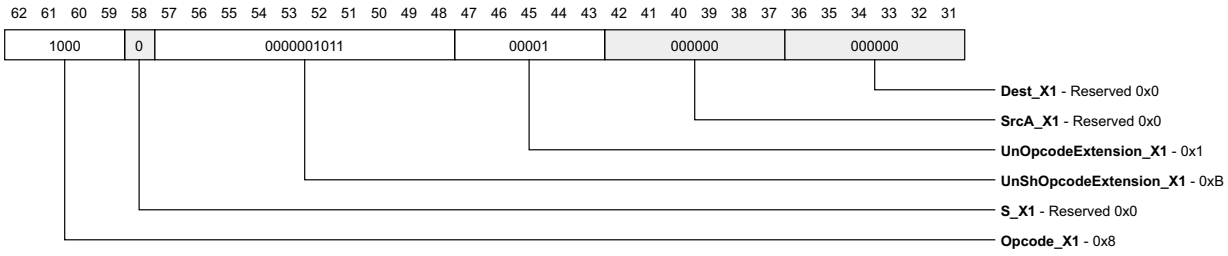

Figure 4-429: drain in X1 Bit Descriptions

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 minimally guaranteed to 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 Instruction Stream Coherence 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

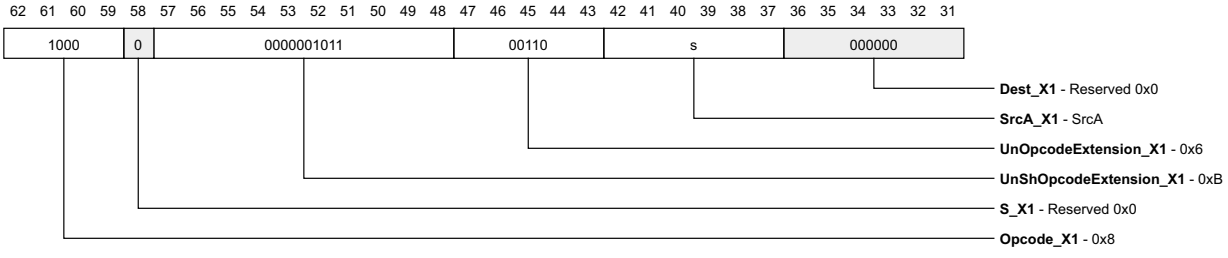

Figure 4-430: icoh in X1 Bit Descriptions

ill: Illegal Instruction

Syntax

```
ill
```

Example

```
ill
```

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.

Functional Description

```
illegalInstruction();
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X		X	

Encoding

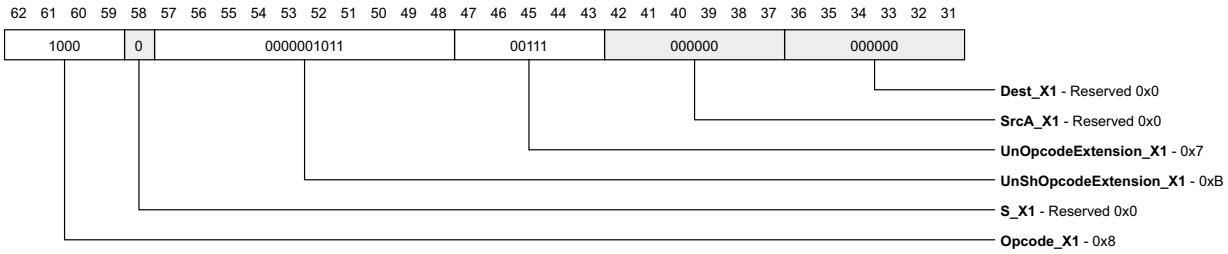

Figure 4-431: ill in X1 Bit Descriptions

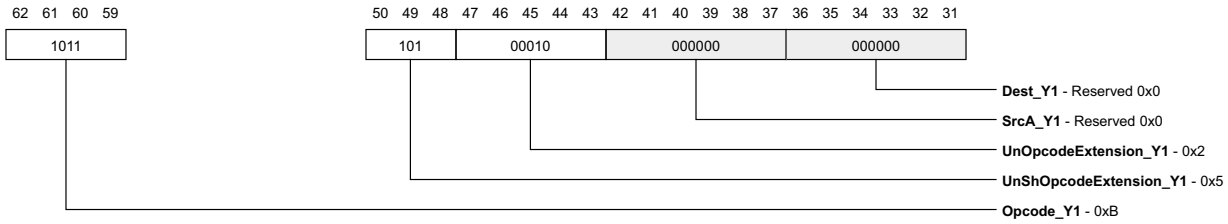

Figure 4-432: ill in Y1 Bit Descriptions

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

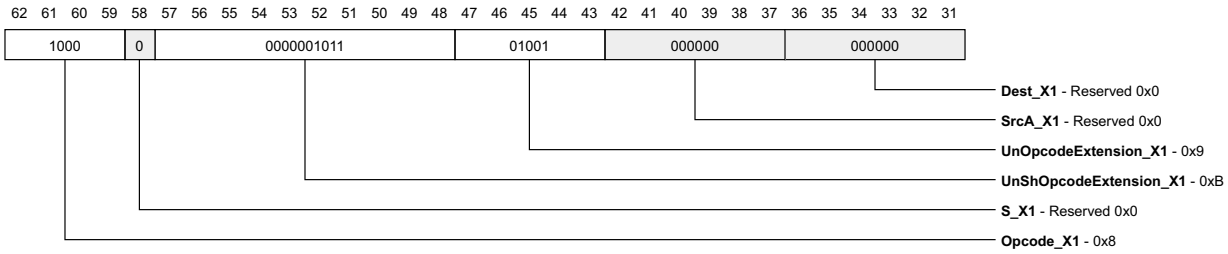

Figure 4-433: iret in X1 Bit Descriptions

mfspr: Move from Special Purpose Register Word

Syntax

```
mfspr Dest, Imm15
```

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^{15} possible special purpose registers.

Functional Description

```
rf[Dest] = sprf[Imm15];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

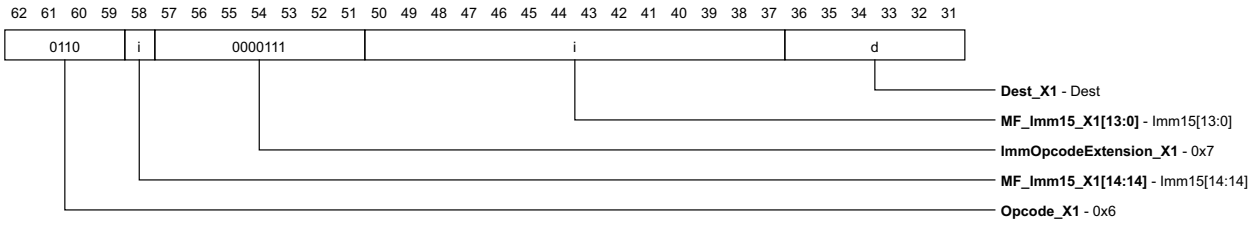

Figure 4-434: mfspr in X1 Bit Descriptions

mtspr: Move to Special Purpose Register Word

Syntax

```
mtspr Imm15, 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^{15} possible special purpose registers.

Functional Description

```
sprf[Imm15] = rf[SrcA];
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

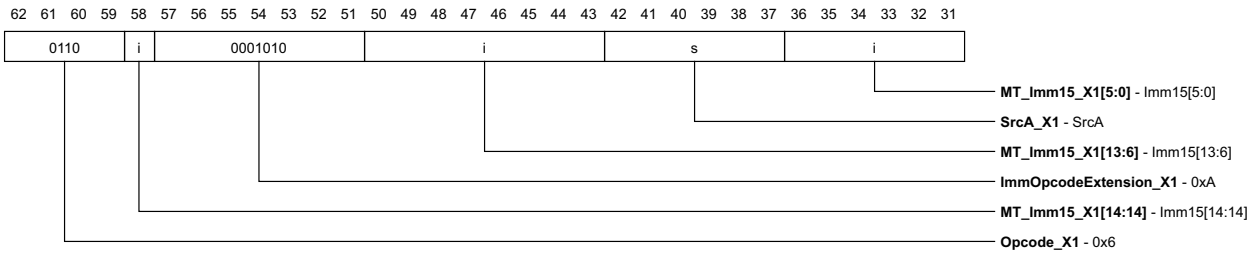

Figure 4-435: mtspr in X1 Bit Descriptions

nap: Nap

Syntax

```
nap
```

Example

```
nap
```

Description

Enters a lower power state. This instruction may or may not complete. To guarantee continued napping 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

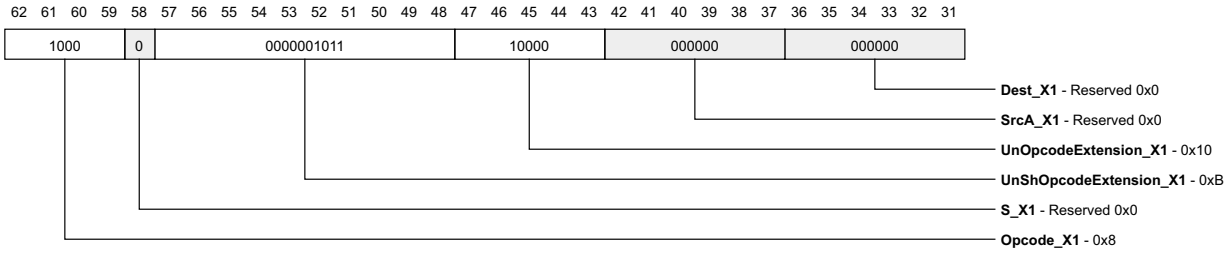

Figure 4-436: nap in X1 Bit Descriptions

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 Software Interrupt 0 instruction will produce unspecified results.

Functional Description

```
softwareInterrupt(0);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

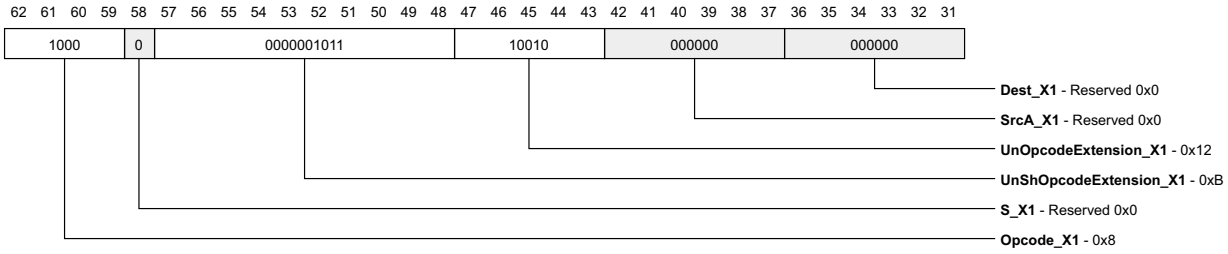

Figure 4-437: swint0 in X1 Bit Descriptions

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 Software Interrupt 1 instruction will produce unspecified results.

Functional Description

```
softwareInterrupt(1);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

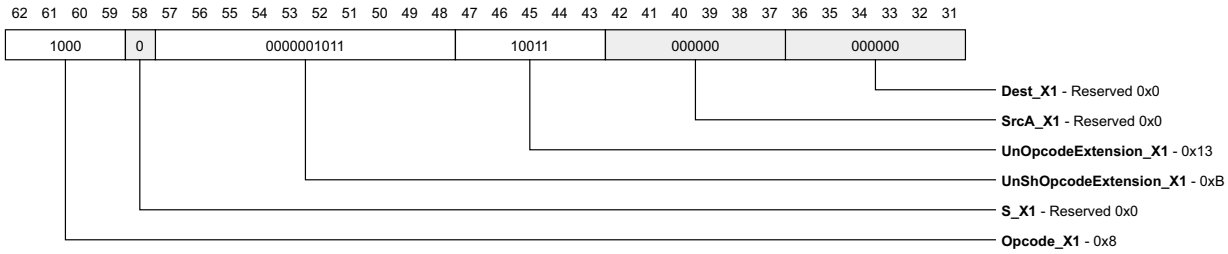

Figure 4-438: swint1 in X1 Bit Descriptions

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 Software Interrupt 2 instruction will produce unspecified results.

Functional Description

```
softwareInterrupt(2);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

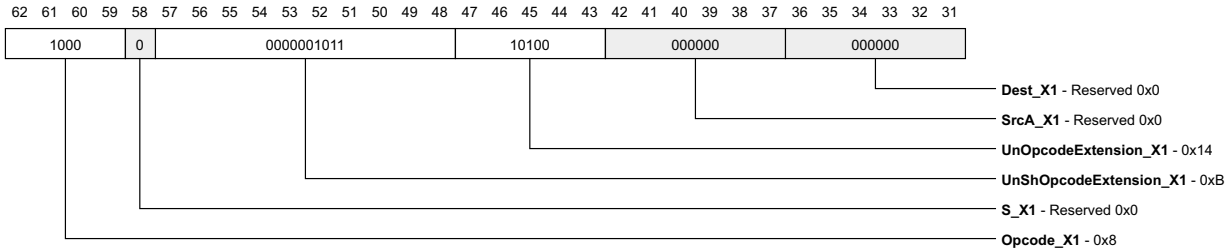

Figure 4-439: swint2 in X1 Bit Descriptions

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 Software Interrupt 3 instruction will produce unspecified results.

Functional Description

```
softwareInterrupt(3);
```

Valid Pipelines

X0	X1	Y0	Y1	Y2
	X			

Encoding

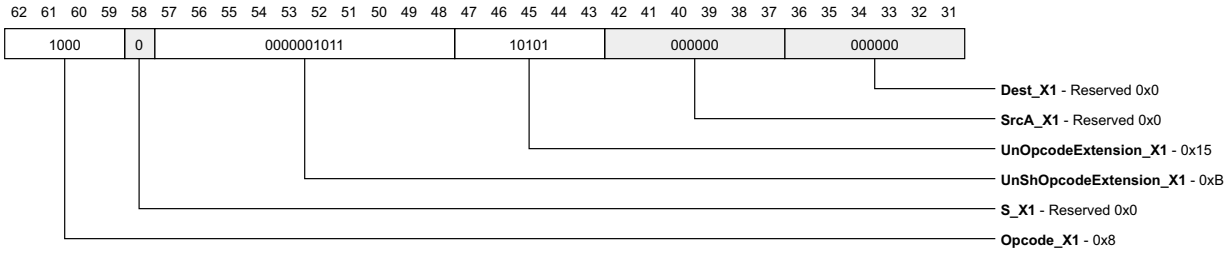

Figure 4-440: swint3 in X1 Bit Descriptions

4.1.15 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 4-4. Pseudo Instructions

Pseudo Instruction	Canonical Form
move dst, src	or dst, src, zero
movei dst, simm8	ori dst, zero, simm8
moveli dst, simm16	addli dst, zero, simm16
movelis dst, simm16	addlis dst, zero, simm16
j[1] target	jf target *or* jb target
jal[1] target	jalf target *or* jalb target
prefetch[2] src	lb_u zero, src
prefetch_L1 src	lb_u src, src
bpt[3]	ill
info simm8	andi zero, zero, simm8
infol simm16	auli zero, zero, simm16

1 Because of limitations in the instruction encoding space, forward-going direct jumps (`jf`, `jalf`) and backward-going direct jumps (`jb`, `jalb`) have different opcodes. If the programmer uses the pseudo-instruction `j` or `jal`, the assembler will generate the appropriate ISA instruction depending upon the target of the jump.

2 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.

3 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` 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.

5 MEMORY AND CACHE ARCHITECTURE

5.1 Memory Architecture

The Tile Processor™ architecture defines a flat, globally shared 64-bit physical address space and a 32-bit virtual address space. The TILE64™ and TILE*Pro*™ family of processors implement a 36-bit physical address space. The globally shared physical address space provides the mechanism by which processes and threads can share instructions and data. Data memory is byte, half-word, and word addressable.

By default, hardware provides a cache-coherent view of data memory to applications. That is, a read by a thread or process to a physical address P will return the value of the most recent write to address P. Instruction memory that is written by the process itself (self-modifying code) or by other processes is not kept coherent by hardware. Special software sequences using the icoh instruction must be used to enforce coherence between data and instruction memory. In the TILE64 implementation, IO writes are not kept coherent with on-chip caches. The TILE*Pro* implementation provides hardware cache coherence for IO accesses.

A non-coherent and a non-cacheable memory mode is also supported, as shown in Table 5-1. In addition to the memory modes, the architecture provides several memory attributes for controlling the allocation and distribution of cache lines. These are shown in Table 5-2.

The Tile Processor architecture memory attributes and modes are managed and configured through system software programming of page tables and enforced through TLB entries. Chapter 4 of the *Multicore Development Environment Optimization Guide* (UG105) provides the Application Programmer Interface (API) and details about memory allocation.

Table 5-1. Tile Processor Architecture Memory Modes

Memory Mode	Description
Coherent Memory	Hardware cache coherent memory.
Non-Coherent Memory	Hardware does not maintain coherence.
Non-Cacheable Memory	Data cache blocks are not cached in any on chip caches. Instruction cache blocks are not cached in the unified L2. Instruction cache blocks are always cached in the L1 instruction cache.

Table 5-2. Supported Allocation Control for Tile Architecture

Attributes	Description
No L1d allocation	Lines are not allocated in the L1d cache (TILE*Pro* only).
No L2 allocation	Remotely homed lines are not allocated in the L2 cache
Pinned memory	Hardware will lock the requested memory page in the L2 cache.
Hashed	Lines on page are distributed across cores according to a hardware hash function (TILE*Pro* only).

5.2 Cache Architecture

5.2.1 Overview

Due to the large difference between DRAM and processor speeds, the cache subsystem is critical for delivering high performance. The cache subsystem's primary role is to prevent the processor cores from stalling due to long memory latencies. To this end, the cache subsystem implements a high performance, non-blocking, two-level cache hierarchy. The two-level design isolates the timing-critical L1 caches from complexity, allowing the L1 data and instruction cache design to be simple, fast, and low power.

The execution engine does not stall on load or store cache misses. Rather, execution of subsequent instructions continue until the data requested by the cache miss is actually needed by another instruction. The cache subsystem is non-blocking and supports multiple concurrent outstanding memory operations. The cache subsystem supports *hit under miss* and *miss under miss*, allowing loads and stores to different addresses to be re-ordered to achieve high bandwidth and overlap miss latencies, while still ensuring that true memory dependencies are enforced.

The cache subsystem provides cache-coherent shared memory, atomic instructions (test-and-set), and memory fences (MF). The TILE*Pro* cache system maintains coherence with I/O DMA accesses to memory, and allows I/O to read and write the on-chip caches directly.

Finally, the cache subsystem implements a software-programmable hardware direct memory access engine (DMA) and supports using portions of the L2 cache as a scratchpad memory.

5.2.2 Cache Microarchitecture

Table 5-3 lists the most important characteristics of the TILE64 and TILE*Pro* cache subsystems.

Table 5-3. Cache Subsystems

	TILE64	**TILE*Pro***
L1 instruction (L1I) cache	8 KB, direct-mapped	16 KB, direct-mapped
L1 instruction translation lookaside buffer	8 entries, fully associative	16 entries, fully associative
L1 data (L1D) cache	8 KB, two-way associative	
L1 data translation lookaside buffer	16 entries, fully associative	
L2 unified cache	64 KB, two-way associative	64 KB, four-way associative
Latency (load to use)	2 cycles L1D hit, 8 cycles local L2 hit, 30-60 cycles remote L2 hit, 80 cycles L2 miss to memory	
Architecture	Non-blocking, out-of-order, stall-on-use	
DDC[a] technology	No	Yes
Line Size	L1I: 64B L1D: 16B L2: 64B	
Allocate Policy	L1I: Allocate on read miss L1D: Allocate on load miss only L2: Allocate on load or store miss	
Write Policy	L1I: N/A L1D: Write through, Store update on hit L2: Writeback	
Error Protection	L1I: 64-bit parity L1D: 8-bit parity L2: 8-bit parity	

a.Dynamic Distributed Cache

Figure 5-441 shows the top level block diagram for the Tile cache subsystem. The processor engine can issue one load or one store per cycle. The L1D cache is checked for the requested data. If the L1D does not have the requested data, the request is delivered to the L2 cache. Stores update the L1D if the targeted cache block is present, and always write thru to the L2 cache. The L1I cache is supported by a hardware prefetching engine that predicts and fetches the most likely next instruction cache line.Misses in the L2 cache on a given tile are satisfied by caches in other tiles or from external memory. If the other caches do not have the requested cache line, then they in turn fetch it from external memory and deliver it to the requesting core.

The cache subsystem supports out of order retirement, meaning instructions subsequent to a load or store miss can write the destination register before the load or store completes. Architectural state is kept consistent, due to the issue logic that blocks subsequent instructions from using stale data. The L2 cache subsystem supports multiple outstanding memory operations and cache misses. The L2 cache subsystem maintains an outstanding miss file to track transactions launched from this tile to memory or to other tiles. Each tile can have up to eight outstanding load misses to external memory as well as four (two for TILE64) outstanding L2 writebacks.

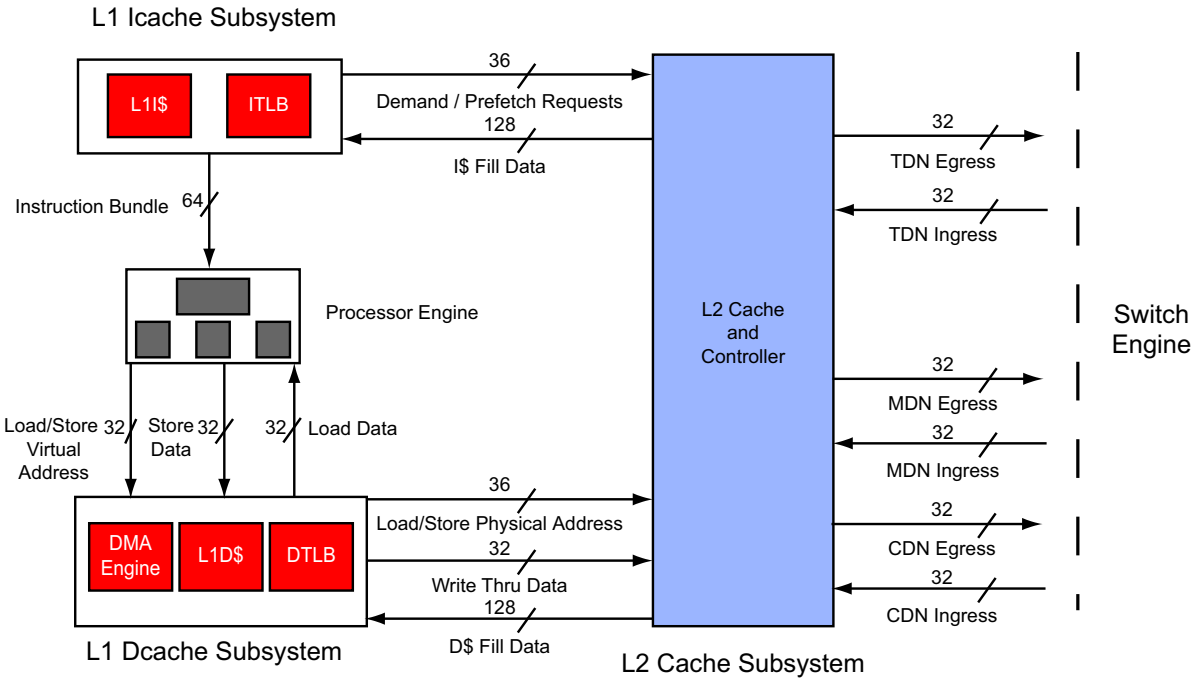

Figure 5-441. Cache Engine Block Diagram

5.2.2.1 Dynamic Distributed Cached Shared Memory

The TILE*Pro* uses the Dynamic Distributed Cache (DDC) to provide a hardware-managed, cache-coherent approach to shared memory. Applications normally access distributed coherent cached shared memory using loads and stores. DDC allows a page of shared memory to be homed on a specific tile (or distributed across many tiles), then cached remotely by other tiles. This mechanism allows a tile to view the collection of on-chip caches of all tiles as a large shared, distributed coherent cache. It promotes on-chip access and avoids the bottleneck of off-chip global memory. This form of shared memory access is particularly useful when processes read and write shared data in a fine-grained, interleaved manner — such as with locks and other synchronization objects.

Figure 5-443 shows a read from tile A (the remote requesting tile) to a cacheline X, where cacheline X is homed at tile B (the home tile):

1. Tile A first checks its local caches for the cacheline X, and on a miss, sends a request for cacheline X to tile B.
2. Tile B receives the request for cacheline X and retrieves cacheline X from its L2.
3. Tile B then sends the full cacheline X back to tile A. Tile A installs cacheline X in its local L1 and L2 caches.

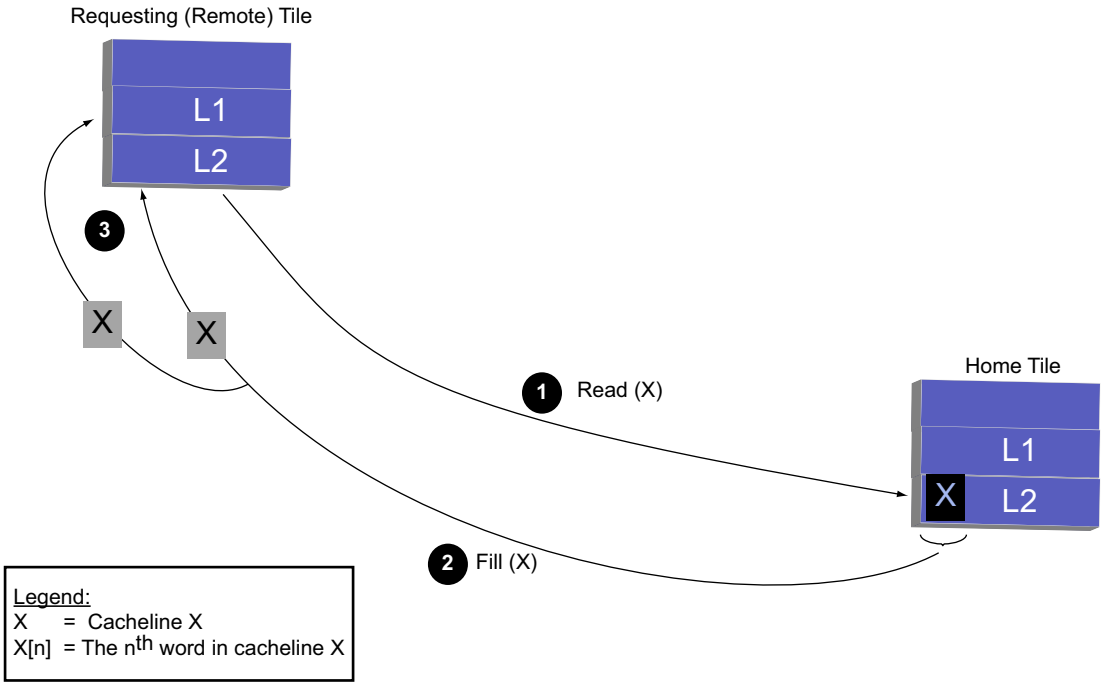

Figure 5-442. Request to Home Tile/Fill L2/L1 with Cacheline X

Figure 5-443. shows a write from tile A to a word (X[0]) in cacheline X, where cacheline X is again homed at tile B.

1. Tile A sends the write address and data to tile B.
2. Tile B receives the write address and data and checks the directory information for cacheline X. The directory indicates that tile C (the sharing tile) has a copy of cacheline X. Tile B updates cacheline X with the new value for word X[0].
3. Tile B sends an invalidate message to tile C.
4. Tile C receives the invalidation and invalidates cacheline X from its caches.
5. Tile C then sends an invalidation acknowledgement back to tile B.
6. Tile B receives the invalidation acknowledgement and sends a write acknowledgement back to tile A.
7. Tile A receives the write acknowledgement message and thus knows that the write to word X[0] has completed.

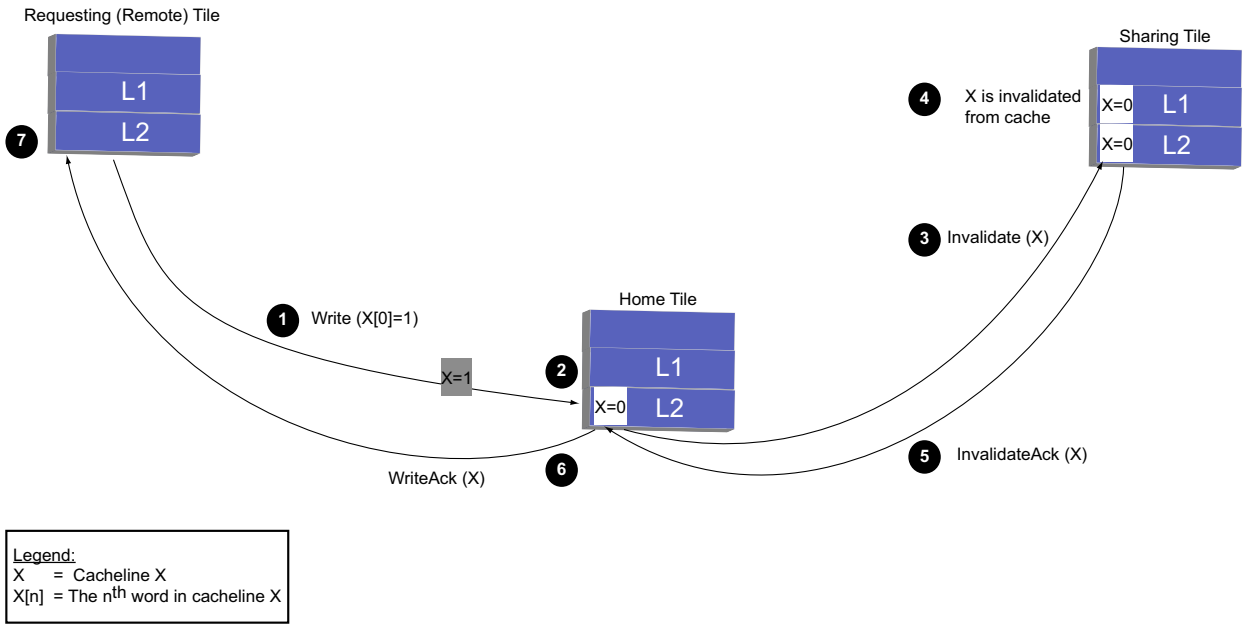

Figure 5-443. Write from Tile A to Word [0] in Cacheline X

5.2.2.2 Coherent and Direct-to-Cache I/O

TILE*Pro* provides hardware cache coherence for I/O DMA accesses. On a write to memory from an I/O DMA engine, the hardware invalidates any cached copies of the line, and updates the cache with the newly written data.

Similarly, on a read to memory from an I/O DMA engine, the hardware checks the on-chip caches for the line and supplies it from there if found. The *System Architecture Manual* (UG103) describes these mechanisms in detail.

5.2.2.3 Striped Memory

TILE*Pro* provides a boot time option to enable a "striped main memory" mode of operation. Striped main memory mode overrides the default mapping of physical memory pages to the four main memory controllers. In striped main memory mode, a physical page of memory is "striped" across the four controllers at an 8KB granularity. That is, a 64KB page would have the first quarter of the page located at memory controller 0, the second quarter at memory controller 1, the third quarter at memory controller 2, and the last quarter at memory controller 3. The striped main memory mode of operation uniformly spreads all physical memory pages across the controllers, thus balancing the load among the four controllers.

5.2.3 Direct Memory Access

The Tile Processor architecture provides a direct memory access (DMA) engine in each tile. This engine can be configured by the application programmer to move data to and from main memory and the L2 cache, and between cores.

The DMA engine operates autonomously from the processor core, issuing DMA load and DMA store operations during cycles in which the cache pipeline is not being used by the processor engine. The DMA source and destination addresses need not be word or cacheline-aligned. The application programmer can specify different source and destination strides, with which the

DMA can perform complex memory transformations such as “shape changes”, in addition to simple copy operations. Each read or write operation performed by the DMA engine executes through the data Translation Lookaside Buffers (TLBs); therefore DMA operations are fully protected and inherit memory attributes for the memory page being accessed. As a result, the DMA engine can be used to move data, for example, from an uncacheable buffer in main memory to a pinned, cacheable buffer. The DMA engine can move data from one tile’s L2 cache to another tile’s L2 cache in the background. Completion of a DMA transfer can be signaled via an interrupt (DMA_NOTIFY) or by polling a special-purpose register (SPR).

The application programmer configures the DMA engine by writing to several SPRs. To perform a DMA request, the DMA transfer description registers (DMA_BYTE, DMA_CHUNK_SIZE, DMA_DST_ADDR, DMA_DST_CHUNK_ADDR, DMA_SRC_ADDR, DMA_SRC_CHUNK_ADDR, and DMA_STRIDE) are set appropriately, and then the REQUEST bit in DMA_CTR register is set. Figure 5-444 illustrates how a 2D-to-1D DMA transfer is handled.

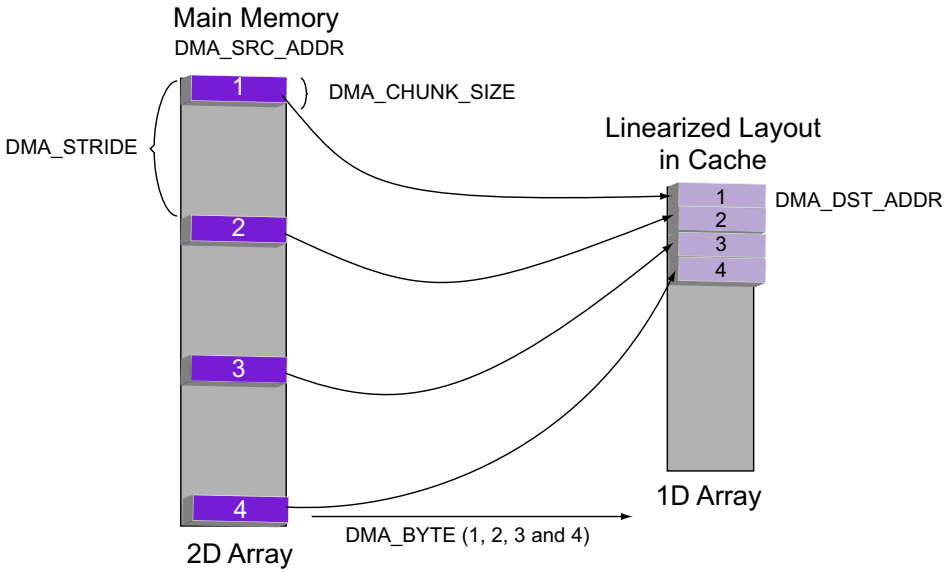

Figure 5-444: 2D-to-1D DMA Transfer

Table 5-4. DMA Registers

DMA Registers	Description
DMA_BYTE	**DMA Byte Register.** This register serves two functions. It contains the size (in bytes) to be transferred in the first chunk and the number of chunks to be transferred. For a detailed description of this register, see page 448.
DMA_CHUNK_SIZE	**DMA Chunk Size Register.** For a detailed description of this register, see page 449.
DMA_CTR	**DMA Control Register.** This register controls the DMA engine. For a detailed description of this register, see page 450.
DMA_DST_ADDR	**DMA Destination Address Register.** This register holds the address of the first byte to be written when the next DMA operation is started. For a detailed description of this register, see page 451.

Table 5-4. DMA Registers (continued)

DMA Registers	Description
DMA_DST_CHUNK_ADDR	**DMA Destination Chunk Address Register.** This register holds the address of the first byte in the first destination chunk for the next DMA operation. For a detailed description of this register, see page 452.
DMA_SRC_ADDR	**DMA Source Address Register.** This register contains the address of the first byte in Main Memory to be read when the next DMA operation is started. For a detailed description of this register, see page 453.
DMA_SRC_CHUNK_ADDR	**DMA Source Chunk Address Register.** This register holds the address of the first byte in the first source chunk in Main Memory for the next DMA operation. For a detailed description of this register, see page 454.
DMA_STRIDE	**DMA Source And Destination Strides Register.** This register specifies the DMA source and destination strides.[a] For a detailed description of this register, see page 455.
DMA_USER_STATUS	**DMA User Status Register.** This register records the current user DMA operation status. For a detailed description of this register, see page 456.

a.SOURCE stride field and DEST stride field in the DMA_STRIDE SPR must be specified.

5.3 Memory Consistency Model

The Tile Processor architecture's memory consistency model specifies the order in which memory operations from a processor become visible to other processors in the coherence domain.

There are two main properties, P1 and P2, defined by the memory consistency model: instruction reordering rules and store atomicity. The Tile Processor architecture defines a relaxed memory consistency model in which:

P1: Instruction Reordering

Non-overlapping memory accesses from a given processor that reference shared pages can be reordered and can become visible to other processors sharing that page in an order different from the original program order, with the following restrictions:

- Data dependencies through memory accesses from a single processor are enforced (RAW, WAW, and WAR)
- Data dependencies through registers or memory determines local visibility order
- Local ordering established by memory data dependencies or register dependencies does not determine global visibility order. See Data writes (including *test-and-set* and flushes) must observe control dependencies.

P2: Store Atomicity

Stores performed by a processor appear to become visible simultaneously to all remote processors, but can become visible to the issuing processor before becoming globally visible (for example, by bypassing to a subsequent load through a write buffer). Test-and-set operations are atomic to *all* processors: bypassing to or from test-and-set operations is not allowed.

The Tile Processor architecture provides the memory fence (MF) instruction to establish ordering among otherwise unordered instructions when such ordering is needed for correctness. Data memory operations in the program prior to the memory fence instruction are made globally visible before ANY operation after the memory fence.

The Tile Processor architecture provides a test-and-set (TNS) instruction to read and write a memory location atomically.

The following code sequences illustrate the properties of the tile memory consistency model. In the examples that follow, memory addresses are denoted by x and y, are word aligned, and are assumed to contain the value 0 initially. All loads and stores are word-sized. The notation A → B indicates that operation A becomes visible to all processors in the coherence domain before operation B becomes visible. Examples Listing 5-1. through Listing 5-5. below illustrate property P1—instruction reordering. Examples Listing 5-6. through Listing 5-8. illustrate property P2—store atomicity and write bypassing.

Listing 5-1. Property P1—Instruction Reordering. Stores can reorder with stores to different locations and loads can reorder with loads to different locations.

```
Tile 0              | Tile 1
sw [x] = 1          | lw r1 = [y]
sw [y] = 1          | lw r2 = [x]
```

All outcomes for r1 and r2 are possible.

The stores can be made visible in any order. Implementations are free to reorder data memory operations to different locations. Program order does not imply visibility order.

Listing 5-2. Property P1—Instruction Reordering. Ordering is enforced through the memory fence instruction.

```
Tile 0              | Tile 1
sw [x] = 1 //M1     | lw r1 = [y] // M4
MF // M2            | MF // M5
sw [y] = 1 // M3    | lw r2 = [x] // M6
```

The only illegal outcome is r1 == 1 and r2 == 0.

Notice that this example is the same as in Listing 5-1., except that here we have an MF instruction inserted between the pair of stores on Tile 0 and also between the pair of loads on Tile 1. The use of the MF instruction ensures that M1→M3 and M4→M6. Therefore, if M3 is visible to M4, then M1 is visible to M6.

Listing 5-3. Property P1—Instruction Reordering. Loads can reorder with stores to different locations.

```
Tile 0              | Tile 1
sw [x] = 1  //M1    | sw [y] = 1// M3
lw r1 = [y] // M2   | lw r2 = [x]// M4
```

This example is similar to Listing 5-1., in that the loads and stores on each tile have no dependence and can be freely reordered. All outcomes are legal.

Listing 5-4. Property P1—Instruction Reordering. Preventing loads from passing stores to different locations.

```
Tile 0              | Tile 1
sw [x] = 1  //M1    | sw [y] = 1// M3
MF                  | MF
lw r1 = [y] // M2   | lw r2 = [x]// M4
```

The only illegal outcome is `r1 == r2 == 0`.

This example is similar to the one shown in Listing 5-3., except we now have MF instructions between the memory operations. The MF on Tile 0 causes M1→M2, and the MF on Tile 1 causes M3→M4. Therefore:

If r1 == 0, we have M2→M3, so we have M1→M2→M3→M4, so r2 == 1.

If r2 == 0, we have M4→M1, so we have M3→M4→M1→M2, so r1 == 1.

If r1 == 1, we have M3→M2, but M4 is not ordered with M1, so r2 == 0 OR r2 == 1.

If r2 == 1, we have M1→M4, but M2 is not ordered with M3, so r1 == 0 OR r1 == 1.

Listing 5-5. Property P1-Instruction Reordering.

```
Tile 0              | Tile 1
sw [x]=1 //M1       | lw r2 = [y]//M4
MF //M2             | bbs r5, foo
sw [y] = 1 // M3    | lw r3 = [x]//M6
```

Here, r2 == 1, r3 == 0 is a legal outcome. M6 is dependant on the branch, however the branch is not dependent on M4. Therefore, there is no dependency between M4 and M6 and they can be reordered. Specifically, M4 may miss in the cache. While the miss is outstanding, the branch and M6 both execute, and M6 hits in the cache, writing r3 == 0. Then, the stores on Tile 0 execute and M4 gets the new value of y (1).

Listing 5-6. Property P2—Store Atomicity and Write Bypassing. Local data dependencies do not establish global visibility ordering: processors can see their own writes early.

```
Tile 0              | Tile 1
sw [x] = 1 //M1     | lw r2 = [y]//M4
lw r1 = [x] //M2    | MF      //M5
sw [y] = r1 // M3   | lw r3 = [x]//M6
```

The following is a legal outcome: r1 == r2 == 1, r3 == 0.

In this case, true data dependencies on Tile 0 cause M1, M2, and M3 to EXECUTE on Tile 0 in order. However, this *does not* imply that they become globally visible to Tile 1 in this order.

The above outcome could occur if Tile 0 bypassed the sw to x to the lw x through a write buffer or local cache. Now, operation M3 writes memory, and operation M4 observes the write M3, but operation M6 gets to memory before operation M1 has become globally visible. To avoid the local bypass, Tile 0 should issue a MF instruction between M1 and M2. This forces M1 to become globally visible before M3.

Listing 5-7. Property P2—Store Atomicity and Write Bypassing. Local data dependencies establish local ordering.

```
Tile 0              | Tile 1
sw [x] = 1 //M1     | lw r1 = [y] // M4
MF //M2             | lw r2 = [r1] //M5
sw [y] = x //M3
```

r1 == x and r2 == 0 is an illegal outcome.

M5 is data dependent on M4 and thus executes (and becomes locally visible) after M4.

Listing 5-8. Property P2—Store Atomicity and Write Bypassing. Stores have a single order as observed by remote

processors.

```
Tile 0              | Tile 1            | Tile 2            | Tile 3
sw [x] = 1 //M1     | lw r1 = [x] //M2  | sw [y] = 1 //M4   | lw r3 = [y] //M5
                    | MF                |                   | MF
                    | lw r2 = [y] //M3  |                   | lw r4 = [x] //M6
```

`r1 == 1, r3 == 1, r2 == 0, r4 == 0` is an illegal outcome.

If the above outcome were legal, this would imply that Tile 3 observes M4 occurring before M1 and Tile 1 observes M1 occurring before M4. More formally, Tile 1 observes: `M1` → `M2` → `M3` → `M4`. While Tile 3 observes: `M4` → `M5` → `M6` → `M1`. Recalling property P2 of the consistency model, it should be noted that because a store from a given processor occurs atomically as observed by remote processors, the above outcome is illegal.

6 ON-CHIP NETWORK ARCHITECTURE

6.1 Overview

The TILE*Pro*™ and TILE64™ family of chips utilize multiple two-dimensional mesh networks for communication between tiles and I/O devices. Memory System traffic, Cache System traffic, I/O traffic and Software based messaging all travel over the Tilera mesh networks. Each switch point in a given network contains a dedicated link to/from the Tile Processor™, as well as four bidirectional links in the cardinal directions (north, south, east and west) to neighboring switch points. The networks run at the same frequency as the Tile Processor core, providing a single cycle latency for the head of a message to "hop" from one network switch point to a neighboring switch point. The networks can be classified into two groups, the *Memory Networks*, which handle all memory traffic such as cache misses, DDR2 requests, and so forth; and the *Messaging Networks,* which allow software to have control of the network and manually send messages between tiles and I/O devices. The Memory Networks consist of the Memory Dynamic Network (MDN), the Tile Dynamic Network (TDN), and the Coherence Dynamic Network (CDN, TILE*Pro* only). The Messaging Networks consist of the User Dynamic Network (UDN) and the I/O Dynamic Network (IDN).

6.2 Network Properties

6.2.1 Switches

Each switch point in the Tilera® networks is implemented as a full crossbar, shown in Figure 6-445. Any input port can arbitrate for any output port, excluding itself (north cannot route north).

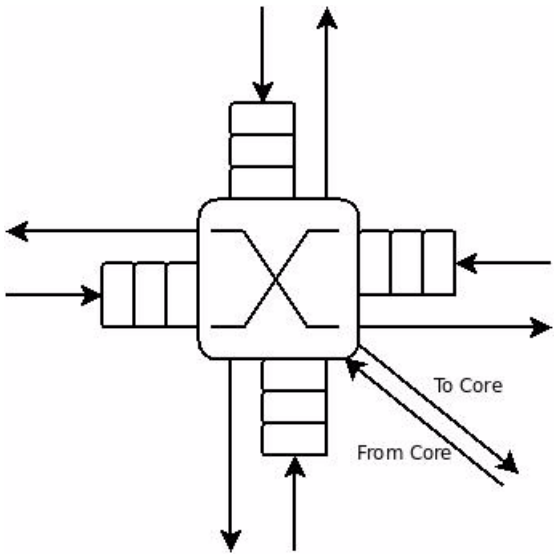

Figure 6-445: Crossbar Switch

6.2.2 Packets

Data is transmitted over the Tilera networks via "packets". Each packet is divided into multiple *N* bit "flits", where *N* is the width of the network. Each packet contains a header flit designating the destination of the packet and the size of the packet, and a payload of data flits.

6.2.3 Routing

The Tilera networks are "wormhole" networks. In a wormhole network, the header flit arbitrates for a given output port at a switch, and, once granted, locks down that output port until the final flit in the packet has successfully traversed the switch. For large packets, this type of routing may result in the reservation of multiple output ports simultaneously for the same packet. The Tilera networks use a *dimension-ordered* routing policy, where packets always travel in the X direction first, then the Y-direction. The TilePro family of processors allow each network to be configured to either route X first, or Y first.

6.2.4 Flow Control

Flow control between neighboring switch points is implemented via a credit scheme. Each switch point has an input buffer that may hold three flits. Each output port contains a credit count corresponding to how many available entries the neighboring input port has available. When a flit is routed through an output port, the credit count is decremented. If the credit count is zero, the flit is blocked and cannot proceed. When an input port consumes a flit, a credit is returned to the corresponding output port.

6.2.5 Fairness and Arbitration

The switch points implement round-robin output port arbitration, providing equivalent fairness for all input ports.

6.2.6 Timing

The Tilera networks operate at the same frequency as the processor cores. The latency for a flit to be read from an input buffer, traverse the crossbar, and reach the storage at the input of a neighboring switch is a single cycle.

6.2.7 Link Width

All of the on-chip network links are 32 bits wide.

6.3 Memory Networks

The Memory Networks carry all requests and responses belonging to the cache system and the memory system. The TDN is responsible for carrying tile-to-tile requests, such as read/write requests. The MDN is responsible for carrying requests from a Tile to/from the DDR2s, as well as carrying all acknowledgments and responses to TDN requests. The CDN (only present in TILE*Pro*) carries invalidate messages needed for the cache coherency protocol in the TILE*Pro* series of processors.

6.3.1 Packet Sizes

The following tables contain a breakdown of the type of requests on the different networks and the size of each request in terms of network flits.

Table 6-5. TDN Packets

TDN Opcode	Size in Flits (TILE64)	Size in Flits (TILE*Pro*)
Read Requests	4	3
Write Requests	5	4
Test-and-Set Requests	4	3

Table 6-6. MDN Packets

MDN Opcode	Size in Flits (TILE64 and TILE*Pro*)
Read Requests	3
Write Requests	4-19
Read Responses	3-18
Acknowledgments	2

Table 6-7. CDN Packets

CDN Opcode	Size in Flits (TILE*Pro* Only)
Invalidate Request	3

6.3.2 Deadlock

The Memory Networks are completely managed by hardware and are deadlock free by design.

6.4 Messaging Networks

6.4.1 Register Mapping

Inside the tile, the UDN has a direct connection to the ALU. This allows tiles to communicate with very low latency. The UDN is register-mapped such that any operation can directly write or read the network. For example:

```
add udn0, r5, r6  // Add r5 to r6 and send the result to the UDN
add r5, r6, udn0  // Read a word from the UDN, add to r6 and put the result in r5
```

The UDN can be initialized and accessed via the TMC library. See the *Applications Libraries Reference Guide* (UG227) for details.

Access to the UDN is fully interlocked. This allows an application to read the network port and sleep until data arrives providing a low power wait state with zero latency wake up.

Similarly, on network send, if the network is not able to consume the packet word immediately, the processor will automatically wait until buffer space is available thus saving considerable power and latency over a polling or interrupt driven scheme.

Special Purpose Registers (SPRs) and interrupts are available to monitor the status of the incoming and outgoing network ports in order to provide alternate usage models.

6.4.2 Packet Format

A packet consists of a *route header*, *tag*, and a *variable length payload*. The route header is created by the sender and contains the X,Y coordinates of the target. It is examined at each switchpoint to route the packet through the tile fabric. The second packet word is a *tag* and is also created by the sender. The tag word is used to differentiate between flows at the receiver.

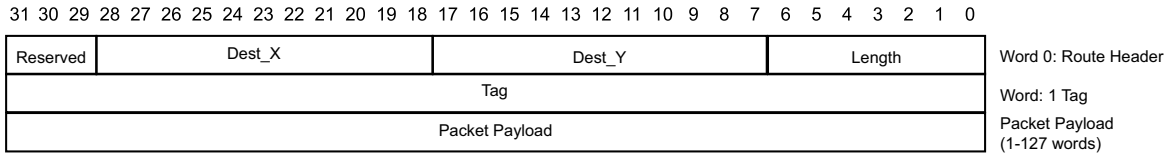

Figure 6-446: UDN Packet Format

Table 6-8. UDN Packet Description

Bits	Name	Description
Word[0]: Route Header		
6:0	Length	**Length of packet in 32-bit words (7 bits).** The length includes the tag word, which all UDN packets must have, but does not include the route header. So, a value of 2 for length indicates a packet with only a route header, a tag word, and one word of payload. A value of zero indicates a 128-word packet.
17:7	Dest_Y	**Destination tile's Y location.** This field is 11 bits.
28:18	Dest_X	**Destination tile's X location.** This field is 11 bits.
31:29	Reserved	**Reserved.** Unused bits reserved for future use. Must be zero. This field is 3 bits.
Word[1]: Tag		
31:0	Tag	Thirty-two-bit value indicating this packet's flow. The tag is used by the demux logic to sort packets. The remaining words are user payload and are not interpreted by hardware
Packet Payload (0-127 words)		

6.4.3 Demux

The UDN receive logic includes demultiplexing (demux) hardware in order to provide high performance flow detection and independent buffering. Based on the tag, an incoming packet is placed in one of four demux queues, as shown in Figure 6-447.

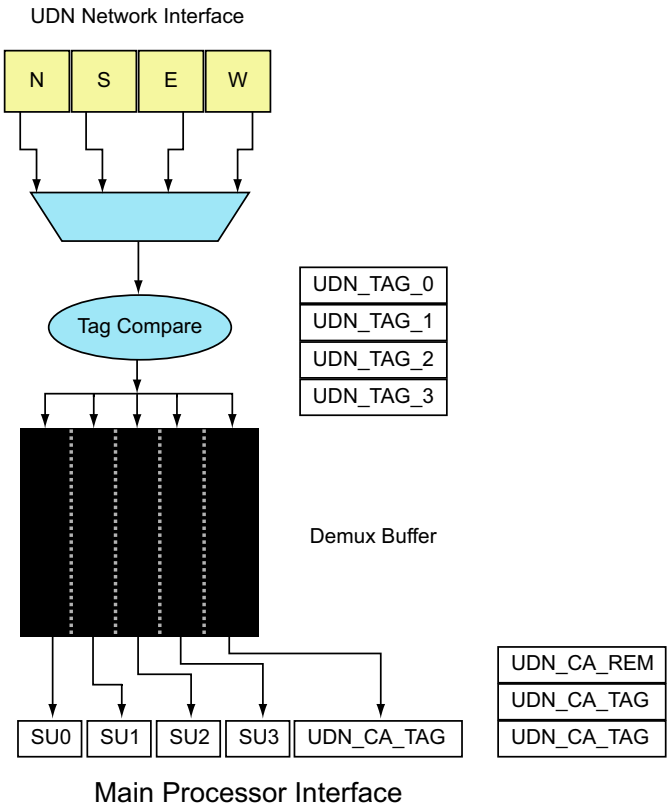

Figure 6-447: Demux Logic

The tag of the incoming packet is compared against four SPRs, (UDN_TAG_0, UDN_TAG_1, UDN_TAG_2, and UDN_TAG_3). If it matches one of the resident tags, the route header and tag words are removed and the payload words are placed in the corresponding demux queue. These queues are accessible individually through register-mapped access via udn0, udn1, udn2 and udn3. This allows differently tagged flows to be serviced out of order with respect to each other.

If the incoming packet does not match any of the programmed tags, it is placed in the catch-all queue with the length field and tag left intact.

The catch-all queue is mapped to the following SPRs:

1. UDN_CA_TAG – the tag of the packet at the head of the catch-all queue
2. UDN_CA_REM – the number of words remaining in the current packet at the head of the catch-all queue
3. UDN_CA_DATA – the SPR that returns the payload data (one word per read SPR read)

   Note that UDN_CA_TAG and UDN_CA_REM are always valid if catch-all is not empty, even when the beginning of the packet has been partially read.

When data is available on one of the queues, it is indicated by the SPR UDN_DATA_AVAIL. Bits 0-3 of this register correspond to the four tagged demux queues, and bit 4 indicates if any data is available in the catch-all queue. An application can poll this register if it needs to wait until data is available on a specific queue.

Interrupts may be enabled to signal when data is available on a queue. The interrupt UDN_CA is signaled when the catch-all queue has data available. In order for a tagged queue to signal an interrupt, it must also be enabled in the UDN_AVAIL_EN SPR (in addition to the system level interrupt enable). The four tagged queues share a data available interrupt. The Interrupt Service Routine (ISR) can check the UDN_DATA_AVAIL register to determine which of the four channels caused the interrupt.

In addition to the data available bits, SPRs are also provided that give the number of words available in each queue. UDN_DEMUX_COUNT_0, UDN_DEMUX_COUNT_1, UDN_DEMUX_COUNT_2, AND UDN_DEMUX_COUNT_3 provide the count for the four tagged demux queues, and UDN_DEMUX_CA_COUNT gives the count of payload words in the catch-all queue. For information about these SPRs, refer to the *System Architecture Manual* (UG103).

The physical buffering for all these queues is implemented as small dedicated FIFOs backed by a larger shared RAM. Space for each queue in the shared RAM is allocated and de-allocated dynamically as needed. This shared buffering provides great flexibility to the message passing system.

The large RAM is also shared with the Input/Output Dynamic Network (IDN), which is only used by system software. The buffering allocated to the IDN and the UDN in the shared buffer is hard partitioned by system software and cannot be modified by the user. There is no interaction between the UDN and the IDN and the UDN will neither block nor corrupt the IDN.

6.4.4 Deadlock

If you are not using iLib Standard Channels, care must be taken to avoid deadlock by software buffering for received packet flows and management of dependences between the outgoing and incoming packet flows.

6.4.5 Hardwall

The UDN hardwall mechanism is used to prevent unwanted communication between user applications running on adjacent tiles. The hardwall mechanism consists of an SPR-programmable protection bit on each output port of the UDN switch point and an interrupt triggered by any attempted violation of a hardwall.

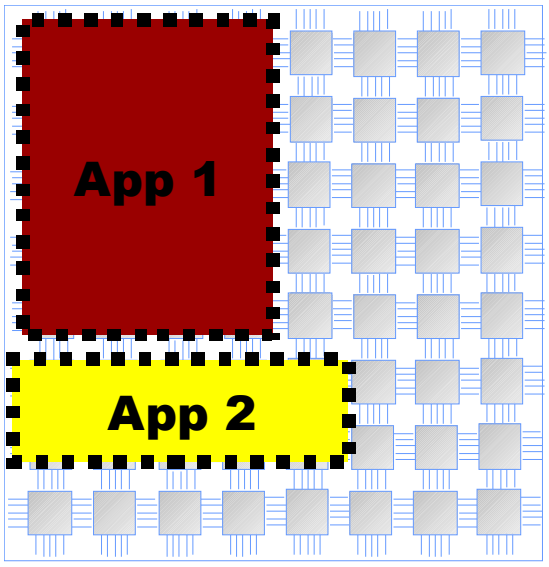

Figure 6-448: UDN Hardwall Mechanism

When an output port is protected, no data can be sent out of the associated port. Attempting to send a packet word to a protected port will trigger an interrupt on the Tile Processor. Software can then inspect the packet and take any appropriate action.

This hardwall also provides a powerful virtualization tool. For example, the hardwall could be used to emulate the behavior of a much larger fabric by detecting messages that cross a hardwall boundary and tunneling them to another group of Tiles or another process running on the same group of tiles.

7 STATIC NETWORK

7.1 Overview

The purpose of the static network is to allow applications to transport scalar operands between tiles efficiently. Instead of using a header to specify the destination, the static network uses routing specifications at each intermediate tile to determine the direction the data should take.

The static network is composed of a crossbar switch that connects to its nearest neighbors in a two-dimensional mesh network, as well as to that tile's processor engine. Each connection is 32-bits, full duplex, and flow controlled. The time required for a word to travel in the network is just one cycle for each hop, or intermediate tile, plus one more cycle at the destination tile to get from the network to the main processor.

The static network crossbar switch can route from five different *directions*: north, south, east, west, and the processor engine. The crossbar is fully connected—every output can be routed from any input (except back to itself) in each cycle, including broadcast and multicast operations.

Data movement is controlled by a static route that is setup with an special purpose registers (SPR) write from the main processor. Static routes remain in force until changed by another SPR write.

7.2 Static Routing

The desired routing is specified statically by writing the SPR SNSTATIC. This SPR has five fields, corresponding to the five possible output ports, as listed in Table 7-9.

Table 7-9. Special Purpose Register Fields

Bits	Output Port
14:12	Main Processor
11:9	West
8:6	South
5:3	East
2:0	North

As shown in Table 7-10, each field contains a number that specifies which input port will route to that output port:

Table 7-10. Port Designations

Numbers	Input Port
0	None
1	North
2	East
3	South
4	West
5	Main Processor

For example if `03214` (in octal) is written to `SNSTATIC`, the following routes remain in effect until `SNSTATIC` is written to again:

- south to west
- east to south
- north to east
- west to north

Multicast routing is supported. For example, writing 00033 (octal) to `SNSTATIC` will cause any word from the south to be routed to both the east and the north.

NOTE: Specifying that an input port routes back to the same output port (for example `00001`, which specifies north routed to north) is illegal and results in undefined behavior.

Each input port in a static route is considered individually, and as soon as the input port has a word available and all output ports have room, the word is moved.

7.3 Data Flow Control

Every port in the static network is flow-controlled, which allows it to tolerate delays introduced by unpredictable events. The TILE64 implements the flow control using a credit-based flow control system. Each link buffers at least three words of storage, and the sender therefore begins with three credits. A sender decrements its credit count when it sends a word, and increments the credit count when it receives acknowledgement from the receiver. A sender can only send when its count is non-0.

7.4 Hardwall Protection

The STN hardwall mechanism is used to prevent unwanted communication between user applications running on adjacent tiles. The hardwall mechanism consists of an SPR-programmable protection bit on each output port of the STN switch point and an interrupt triggered by any attempted violation of a hardwall.

When an output port is protected, no data can be sent out of the associated port. Attempting to send a word to a protected port will trigger an interrupt on the Tile Processor. Software can then inspect the word and take any appropriate action.

When a static route specifies a multicast route, and just one of the many output directions causes a protection violation, the word will not be routed to any of the output ports.

7.5 User-Accessible Special Purpose Registers

The list of all user-accessible SPRs follows. Please see the appendix for more details.

- Static network control register (SNCTL)

  Contains bits to freeze the crossbar switch.

- Static Network FIFO Data register (SNFIFO)

  Used to save or restore static network state, or to extract words blocked by a routing violation.

- Static Network FIFO Select register (SNFIFO_SEL)

  Controls which FIFO is read/written when accessing SNFIFO_DATA.

  - 0 – North Input FIFO
  - 1 – East Input FIFO
  - 3 – South Input FIFO
  - 4 – West Input FIFO
  - 5 – Main Processor Input FIFO
  - 6 – Main Processor Output FIFO

- Static Network Input State register (SNISTATE)

  Used to save or restore static network state. Indicates how many words are in each port's input buffer.

- Static Network Output State register (SNOSTATE)

  Used to save or restore static network state. Indicates how many credits each output port has for sending. Also contains how many words are in the Main Processor Output FIFO.

- Static Network Static Route register (SNSTATIC)

  Used to setup a static route (see "Static Routing" on page 381)

- Static Network Data Available register (SN_DATA_AVAIL)

  Indicates if data is available to be read from the static network by the processor engine.

8 USER-LEVEL SYSTEM CONCERNS

8.1 Overview

User-level programs need to interact with the greater system where they are executed. In order to interact with the system, user-level programs need to be able to execute system calls, interact with I/O, and control in-tile devices of a system nature. This section describes system interactions from a user-level viewpoint.

8.2 System Calls

A system call is a mechanism whereby a user-level program voluntarily passes control flow to a more privileged piece of software. A system call typically involves passing some information along with the program control flow. The system software may elect to pass return data to the user-level program after the system call completes. System calls are typically executed in response to a user-level program requiring some functionality that is provided by system software. Access to system calls are typically done through library code not directly implemented by end users.

The Tile Processor Architecture supports the ability for user-level programs to call system software via the `swint0`, `swint1`, `swint2`, and `swint3` instructions[1]. The architecture includes four "`swint`" interrupt handlers with each one corresponding to one of the swint instructions. When a swint instruction is executed, an interrupt is signaled to the respective swint interrupt handler. There are four swint interrupt levels because there are four protection levels in the Tile Processor Architecture. Therefore it is possible to choose the level of system software in which a program wants to request services. The control of the protection level to which a swint instruction vectors is not hard coded, but as a software convention, the swint number matches the protection level, where 0 is user-level, 1 is supervisor (OS), 2 is hypervisor, and 3 is for a virtual machine monitor.

Typically there are many different calls that a user-level program may want to do to system level software. As there is only one interrupt per protection level, the actual call that is needed must be signaled to system software in some manner. By software convention, a system call number is deposited into a known General Purpose Register (GPR) and then the swint is signaled. The system call number allows the system to determine which service a user-level program requires from system software. Parameters can also be passed through other processor registers and through memory. After the completion of the system call, the system software returns control to the user-level program via the `iret` instruction. The system software may elect to return a value or set of values to the user process through processor registers or through memory.

1. "swint" is an abbreviation of software interrupt.

8.3 Interrupt Overview

Exceptional occurrences happen in any computer system. The Tile Processor Architecture unifies all exceptional occurrences in a class of events called interrupts.

Four protection levels are provided by hardware to isolate protection concerns. These protection levels effect how interrupts occur on the Tile Processor Architecture. The protection levels are numbered 0 through 3 with 0 being the least privileged and 3 being the most privileged. This document focuses on programs executed at protection level 0 (user-level).

Table 8-11 presents the list of interrupts that are available on the Tile Processor Architecture. The *System Architecture Manual* provides more detail of interrupt processing and the various attributes of interrupts. If multiple interrupts are signaled at the same time, the interrupt with the lowest interrupt number will be signaled first.

8.3.1 Interrupt List

Table 8-11 lists all interrupts that can be seen by the user.

Table 8-11. Master Interrupt Table

Interrupt Number	Name	Description
0	ITLB_MISS	ITLB Miss.
1	MEM_ERROR	Memory Error
2	ILL	Illegal Instruction
3	GPV	General Protection Violation
4	SN_ACCESS	Static Networks Access
5	IDN_ACCESS	IO Dynamic Network (IDN) Access
6	UDN_ACCESS	User Dynamic Network (UDN) Access
7	IDN_REFILL	IDN Refill
8	UDN_REFILL	UDN Refill
9	IDN_COMPLETE	IDN Complete
10	UDN_COMPLETE	UDN Complete
11	SWINT_3	Software Interrupt 3
12	SWINT_2	Software Interrupt 2
13	SWINT_1	Software Interrupt 1
14	SWINT_0	Software Interrupt 0
15	UNALIGN_DATA	Unaligned Data

Table 8-11. Master Interrupt Table (continued)

Interrupt Number	Name	Description
16	DTLB_MISS	Data Translation Lookaside Buffer (DTLB) Miss
17	DTLB_ACCESS	DTLB Access Error
18	DMATLB_MISS	Direct Memory Access (DMA) Translation Lookaside Buffer Miss
19	DMATLB_ACCESS	DMA Translation Lookaside Buffer Access Error
20	Reserved	Reserved
21	Reserved	Reserved
22	SN_FIREWALL	SN Firewall Violation
23	IDN_FIREWALL	IDN Firewall Violation
24	UDN_FIREWALL	UDN Firewall Violation
25	TILE_TIMER	Tile Timer
26	IDN_TIMER	IDN Timer
27	UDN_TIMER	UDN Timer
28	DMA_NOTIFY	DMA Notification
29	IDN_CA	IDN Catch-All Available
30	UDN_CA	UDN Catch-All Available
31	IDN_AVAIL	IDN Available
32	UDN_AVAIL	UDN Available
33	PERF_COUNT	Performance Counters
34	INTCTRL_3	Interrupt Control 3
35	INTCTRL_2	Interrupt Control 2
36	INTCTRL_1	Interrupt Control 1
37	INTCTRL_0	Interrupt Control 0
38	BOOT_ACCESS	Boot Access
39	WORLD_ACCESS	World Access
40	I_ASID	Instruction Address Space Identifier (ASID)
41	D_ASID	Data ASID

Table 8-11. Master Interrupt Table (continued)

Interrupt Number	Name	Description
42	DMA_ASID	DMA ASID
43	RESERVED	RESERVED
44	DMA_CPL	DMA Current Protection Level
45	RESERVED	RESERVED
46	DOUBLE_FAULT	Double Fault

The Tile Processor Architecture uses a vectored approach to interrupts; there are four sets of interrupt vectors, one for each protection level. On an interrupt, the architecture changes the program counter to a value derived from the interrupt number and the protection level at which the interrupt executes. The offset is INTERRUPT_BASE_ADDRESS (0xFC000000), plus the protection level multiplied by 16 MB (0x01000000), plus the interrupt number multiplied by 256. This allows 32 VLIW instructions to fit in each interrupt vector, and allows all of a protection level's interrupt vectors and up to 16 MB of accompanying code to be mapped into virtual address space using a single large-page ITLB entry. If more than 32 instructions are needed to handle an interrupt, the interrupt vector code can jump to the rest of the interrupt handler located in that same large page, or anywhere else in the address space.

When an interrupt occurs, the program counter of the processor is vectored to a fixed interrupt location. The fixed interrupt location is the virtual address:

- `(interrupt_number<<(INTERRUPT_VECTOR_NUMBER_OF_INSTRUCTIONS_LOG_2 + INSTRUCTION_SIZE_LOG_2) + destination_protection_level<<(INTERRUPT_VECTOR_PL_OFFSET_LOG_2 + INTERRUPT_BASE_ADDRESS.`
- INTERRUPT_VECTOR_NUMBER_OF_INSTRUCTIONS_LOG_2 is 5.
- INSTRUCTION_SIZE_LOG_2 is 3
- INTERRRUPT_VECTOR_PL_OFFSET_LOG_2 is 24.
- INTERRUPT_BASE_ADDRESS is 0xFC000000.

When an interrupt occurs, in order for a subsequent interrupt to not interrupt a pending interrupt, the INTERRUPT_CRITICAL_SECTION bit is atomically set. This SPR effects the masking of further interrupts and is discussed along with interrupt masking in more detail in the *System Architecture Manual*.

The program counter that was interrupted and the associated protection level on interrupt is placed in the SPRs EX_CONTEXT_X_0 and EX_CONTEXT_X_1. There are four sets of these SPRs, one for each protection level that can be interrupted into. The "X" in EX_CONTEXT_X_0 and EX_CONTEXT_X_1 denotes a value between 0 and 3 for each protection level. EX_CONTEXT_X_0 contains the exceptional program counter and EX_CONTEXT_X_1 contains the protection level that was interrupted along with the interrupted state of the INTERRUPT_CRITICAL_SECTION SPR.

8.4 User-Level Interrupts

Unlike most computer architectures, the Tile Processor Architecture supports user-level interrupts. User-level interrupts interrupt from protection level 0 destined for protection level 0. An example of an interrupt that would interrupt from protection level 0 to protection level 0 is the UDN Available interrupt. The UDN is a user-level network and the availability of a network message can trigger an interrupt to occur. In order for the interrupt to be delivered, place an appropriate interrupt handler in the correct interrupt vector location and unmask the interrupt.

User-level interrupt routines consist of 32 bundles of instructions that are laid out starting at address 0xFC000000. Thus to install a protection level 0 interrupt handler for the UDN Available interrupt (number 32), the interrupt handler would be installed at address 0xFC002000. If more than 32 instruction bundles is required, the last instruction in the interrupt handler should be used to jump to the appropriate code. If other interrupts need to be enabled inside of an interrupt handler, the `INTERRUPT_CRITICAL_SECTION` SPR may be cleared. The *System Architecture Manual* details the masking of interrupts in more detail.

When an asynchronous interrupt is signaled, all of the Tile Processor Architecture's general purpose registers can potentially contain state that cannot be modified in order to return transparently to the interrupted process. This leaves the interrupt handler with a dilemma, it needs to save off state in the general purpose register file to memory in order to use the general purpose registers, but in order to execute a store instruction, at least one general purpose register is needed to hold an address. To address this problem, the Tile Processor Architecture provides system save registers. Four 32-bit system save registers are provided for each protection level. The system save registers can be read and written by the corresponding protection level and higher privileged protection levels. If a lower protection level attempts to access the system save registers of a higher protection level, a General Protection Violation occurs. The system save registers are mapped into the SPR space. The corresponding SPRs for a given protection level are `SYSTEM_SAVE_`*`X`*`_0`, `SYSTEM_SAVE_`*`X`*`_1`, `SYSTEM_SAVE_`*`X`*`_2`, and `SYSTEM_SAVE_`*`X`*`_3`, where *X* denotes a protection level 0 through 3.

After all of the interrupt processing is complete, the `iret` instruction should be executed. The `iret` instruction transitions the program counter to that stored in the `EX_CONTEXT_0_0`. Likewise it updates the `INTERRUPT_CRITICAL_SECTION` SPR and the protection level from `EX_CONTEXT_0_1`. These updates are done atomically.

8.5 Interaction with I/O Devices

User-level code typically interacts with I/O devices by utilizing the features provided by system software. Typically there is a driver for an I/O device which resides in the system software. The prototypical I/O device on Tile Processor Architecture is connected to the IDN and to the memory system via the iMesh. In order for a user-level program to access I/O, a system call is made to system software which then may message an I/O device for the user-level software. In response to the IDN message, the I/O device may respond back with data over the IDN, or may deposit data into memory. The I/O device will typically have sophisticated DMA engines which orchestrate the movement of bulk data from the I/O device to memory or from memory to the I/O device. More details of I/O device specifics are described in the *System Architecture Manual* and in the *Tile Processor I/O Device Guide* (UG104) for a particular implementation.

8.6 Cycle Count

Each tile contains a 64-bit cycle counter. The 64-bit cycle counter is a monotonically increasing counter that can be read by reads to the `CYCLE_LOW` and `CYCLE_HIGH` SPRs. The cycle counter increases for each major cycle of a specific implementation. The relationship between cycle count and instructions executed is implementation specific. A suggested implementation increments the

cycle count for each cycle that a bundle could issue. CYCLE_LOW returns the lower 32 bits of the cycle counter while CYCLE_HIGH returns the upper 32 bits of the cycle count. The cycle counter is reset to 0 when the machine is reset. The CYCLE_LOW and CYCLE_HIGH registers are read only registers. System software can modify the cycle counter for virtualization purposes via the CYCLE_LOW_MODIFY and CYCLE_HIGH_MODIFY special purpose registers.

APPENDIX A SPECIAL PURPOSE REGISTERS

A.1 Introduction

In addition to having the processor state be accessible by the standard Instruction Set Architecture (ISA), every modern processor contains some state that software needs to access, but only infrequently. Consider a DMA operation, for example. A program initiates a DMA transfer by specifying the size of the data block to be transferred, along with the source and destination addresses. The program also polls a status bit to determine when the transfer has completed.

The Tile Architecture™ provides access to all software-readable and software-writable state through a 15-bit addressed, word-oriented register file. This register file is called the Special Purpose Register File (SPRF), and each register in this register set is called an SPR. Not every bit within every SPR is physically implemented to hold state information/data. Some bits merely provide an interface to another state within the tile. Further, the SPRF is sparsely populated—not every address within the SPRF refers to an actual SPR.

Two instructions in the Tile Architecture provide access to the special purpose registers: the *Move To Special Purpose Register Word* (mtspr) and *Move From Special Purpose Register Word* (mfspr). User programs can access SPRs to control and monitor the Static Network, the User Dynamic Network, the Tile Timer, and the DMA Engine.

Table A-12 provides the list of SPRs organized by function. Information for TILE64 users is shown in yellow shading and information for TILE*Pro* users is shown in red shading. Note that the SPRs listed below and described in the sections that follow represent a portion of the complete Special Purpose Register listing. For more information, refer to Chapter 8: Special Purpose Registers in the *System Architecture Manual* (UG103).

Table A-12. Special Purpose Registers

Register/Details	Address	Access MPL
Static Network Registers		
"Static Network Control Register (SNCTL)" on page 396	0x805	SN_ACCESS
"Static Network Fifo Data (SNFIFO_DATA)" on page 397	0x806	
"Static Network FIFO Select Register (SNFIFO_SEL)" on page 398	0x807	
"Static Network Input State Register (SNISTATE)" on page 399	0x809	
"Static Network Output State Register (SNOSTATE) " on page 400	0x80a	
"Static Network Static Route (SNSTATIC)" on page 401	0x80c	

Table A-12. Special Purpose Registers (continued)

Register/Details	Address	Access MPL
Static Network Registers (continued)		
"Static Network Data Available (SN_DATA_AVAIL)" on page 402	0x900	SN_ACCESS
Static Network Static Registers — Used for TILE*Pro* Processors ONLY		
"Static Network Control (SN_STATIC_CTL)" on page 403		SN_STATIC _ACCESS
"Static Network FIFO Data (SN_STATIC_FIFO_DATA)" on page 404		
"Static Network FIFO Select (SN_STATIC_FIFO_SEL)" on page 405		
"Static Network Input State (SN_STATIC_ISTATE)" on page 406		
"Static Network Output State (SN_STATIC_OSTATE)" on page 407		
"Static Network Static Route (SN_STATIC_STATIC)" on page 408		
"Static Network Data Available (SN_STATIC_DATA_AVAIL)" on page 409		
User Dynamic Network Registers		
"User Dynamic Network Catch-all Demultiplexor Count Register (UDN_DEMUX_CA_COUNT) " on page 410	0xc05	UDN_ACCESS
"User Dynamic Network Demultiplexor Count 0 Register (UDN_DEMUX_COUNT_0) " on page 411	0xc06	
"User Dynamic Network Demultiplexor Count 1 Register (UDN_DEMUX_COUNT_1) " on page 412	0xc07	
"User Dynamic Network Demultiplexor Count 2 Register (UDN_DEMUX_COUNT_2) " on page 413	0xc08	
"User Dynamic Network Demultiplexor Count 3 Register (UDN_DEMUX_COUNT_3) " on page 414	0xc09	
"UDN Demux Control Register (UDN_DEMUX_CTL) " on page 415	0xc0a	
"User Dynamic Network Demux Current Tag (UDN_DEMUX_CURR_TAG)" on page 415	0xc0b	
"UDN Demux Queue Select Register (UDN_DEMUX_QUEUE_SEL) " on page 415	0xc0c	
"User Dynamic Network Demux State (UDN_DEMUX_STATUS)" on page 416	0xc0d	
"User Dynamic Network Demux FIFO (UDN_DEMUX_WRITE_FIFO)" on page 416	0xc0e	
"User Dynamic Network Demux Write Queue (UDN_DEMUX_WRITE_QUEUE)" on page 417	0xc0f	

Table A-12. Special Purpose Registers (continued)

Register/Details	Address	Access MPL
User Dynamic Network Registers (continued)		
"User Dynamic Network Words Pending (UDN_PENDING)" on page 417	0xc10	UDN_ACCESS
"User Dynamic Network FIFO Data (UDN_SP_FIFO_DATA)" on page 418	0xc11	
"User Dynamic Network FIFO Data (UDN_SP_FIFO_DATA)" on page 418	0xc12	
"User Dynamic Network Freeze (UDN_SP_FREEZE)" on page 419	0xc13	
"User Dynamic Network Port State (UDN_SP_STATE)" on page 420	0xc14	
"User Dynamic Network Tag 0 (UDN_TAG_0)" on page 421	0xc15	
"User Dynamic Network Tag 1 (UDN_TAG_1)" on page 421	0xc16	
"User Dynamic Network Tag 2 (UDN_TAG_2)" on page 421	0xc17	
"User Dynamic Network Tag 3 (UDN_TAG_3)" on page 422	0xc18	
"User Dynamic Network Tag Valid (UDN_TAG_VALID)" on page 422	0xc19	
"User Dynamic Network Tile Coordinates (UDN_TILE_COORD)" on page 423	0xc1a	
"User Dynamic Network Catch-All Data (UDN_CA_DATA)" on page 424	0xd00	
"User Dynamic Network Catch-all Remaining Words (UDN_CA_REM)" on page 424	0xd01	
"User Dynamic Network Catch-All Data (UDN_CA_DATA)" on page 424	0xd02	
"User Dynamic Network Data Available (UDN_DATA_AVAIL)" on page 425	0xd03	
"User Dynamic Network Refill Available Enable (UDN_REFILL_EN)" on page 426	0x1005	UDN_REFILL
"User Dynamic Network Remaining (UDN_REMAINING)" on page 427	0x1405	UDN_COMPLETE
"User Dynamic Network Available Enables (UDN_AVAIL_EN)" on page 428	0x4005	UDN_AVAIL
User Dynamic Network Registers (continued		
"User Dynamic Network Deadlock Counter (UDN_DEADLOCK_COUNT)" on page 429	0x3605	UDN_TIMER
"User Dynamic Network Deadlock Timeout (UDN_DEADLOCK_TIMEOUT)" on page 430	0x3606	

Table A-12. Special Purpose Registers (continued)

Register/Details	Address	Access MPL
World-Accessible Registers		
"Cycle Counter High (CYCLE_HIGH)" on page 431	0x4e06	WORLD_ACCESS
"Cycle Counter Low (CYCLE_LOW)" on page 431	0x4e07	
"Done Magic Register (DONE)" on page 432	0x4e08	
"Fail Magic Register (FAIL)" on page 432	0x4e09	
"Interrupt Critical Section (INTERRUPT_CRITICAL_SECTION)" on page 433	0x4e0a	
"Pass Magic Register (PASS)" on page 433	0x4e0b	
Interrupt Control 0 Registers		
"Exceptional Context Protection Level 0 Entry 0 (EX_CONTEXT_0_0)" on page 434	0x4a05	INTCTRL_0
"Exceptional Context Protection Level 0 Entry 1 (EX_CONTEXT_0_1)" on page 435	0x4a06	
"Interrupt Control 0 Status (INTCTRL_N_STATUS)" on page 436	0x4a07	
"Interrupt Mask Protection Level 0 Entry 0 (INTERRUPT_MASK_0_0)" on page 437	0x4a08	
"Interrupt Mask Protection Level 0 Entry 1 (INTERRUPT_MASK_0_1)" on page 439	0x4a09	
"Interrupt Mask Protection Level 0 Entry 0 (INTERRUPT_MASK_RESET_0)" on page 440	0x4a0a	
"Interrupt Mask Protection Level 0 Entry 1 (INTERRUPT_MASK_RESET_0_1)" on page 442	0x4a0b	
"Interrupt Mask Protection Level 0 Entry 0 (INTERRUPT_MASK_SET_0_0)" on page 443	0x4a0c	
"Interrupt Mask Protection Level 0 Entry 1 (INTERRUPT_MASK_SET_0_1)" on page 445	0x4a0d	
"System Save Register Level 0 Entry 0 (SYSTEM_SAVE_0_0)" on page 446	0x4b00	
"System Save Register Level 0 Entry 1 (SYSTEM_SAVE_0_1)" on page 446	0x4b01	
"System Save Register Level 0 Entry 2 (SYSTEM_SAVE_0_2)" on page 446	0x4b02	
"System Save Register Level 0 Entry 3 (SYSTEM_SAVE_0_3)" on page 446	0x4b03	
Tile Timer Register		
"Minimum Protection Level for Tile Timer (MPL_TILE_TIMER)" on page 447	0x3205	TILE_TIMER

Table A-12. Special Purpose Registers (continued)

Register/Details	Address	Access MPL
DMA Registers		
"DMA Byte (DMA_BYTE) Register" on page 448	0x3900	DMA_NOTIFY
"DMA Chunk Size (DMA_CHUNK_SIZE) Register" on page 449	0x3901	
"DMA Control (DMA_CTR) Register" on page 450	0x3902	
"DMA Destination Address (DMA_DST_ADDR) Register" on page 451	0x3903	
"DMA Destination Chunk Address (DMA_DST_CHUNK_ADDR) Register" on page 452	0x3904	
"DMA Source Address (DMA_SRC_ADDR) Register" on page 453	0x3905	
"DMA Source Chunk Address (DMA_SRC_CHUNK_ADDR) Register" on page 454	0x3906	
"DMA Source And Destination Strides (DMA_STRIDE) Register" on page 455	0x3907	
"DMA User Status (DMA_USER_STATUS) Register" on page 456	0x3908	

A.2 SPR Register Descriptions

Registers are described in ascending address order.

Static Network Control Register (SNCTL)

This register controls execution of the static network processor and fabric.

Speed

Slow

Minimum Protection Level

SN_ACCESS

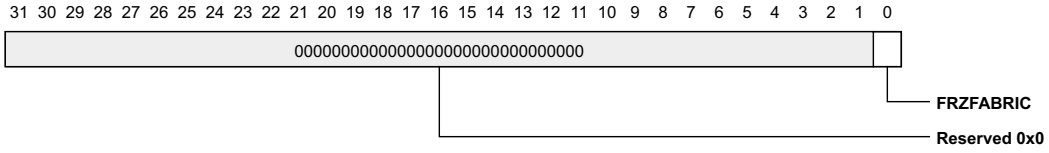

Figure 7. SNCTL Register Diagram Register Diagram

Table A-13. SNCTL Register Bit Descriptions

Bits	Name	Reset	Description
31:2	Reserved		
1	FRZPROC	1	For TILE 64, this bit freezes the static network processor. For TILEPro, this is reserved.
0	FRZFABRIC	1	Freeze the static network fabric.

Static Network Fifo Data (SNFIFO_DATA)

Accesses the data FIFO specified by SNFIFO_SEL. When read, returns the top entry on the specified FIFO and removes it from the FIFO. When written, it writes the specified data into the FIFO.

Speed

Slow

Minimum Protection Level

SN_ACCESS

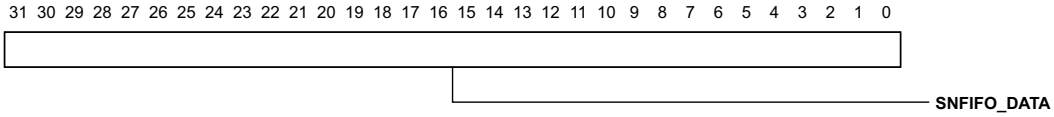

Figure 8. SNFIFO_DATA Register Diagram

Static Network FIFO Select Register (SNFIFO_SEL)

This register specifies which FIFO will be read and written by the SNFIFO_DATA register.

Speed

Slow

Minimum Protection Level

SN_ACCESS

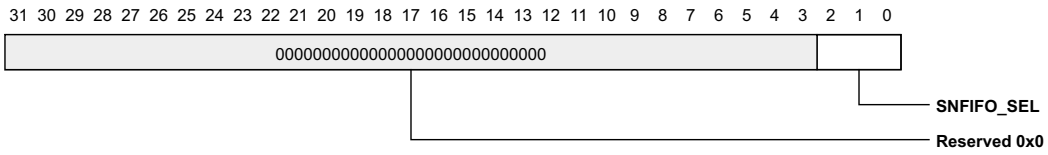

Figure 9. SNFIFO_SEL Register Diagram

Table A-14. SNFIFO_SEL Register Bit Descriptions

Bits	Name	Reset	Description	
31:3	Reserved			
2:0	SNFIFO_SEL	0	This bitfield specifies which FIFO will be read and written by the SNFIFO_DATA register. FIFOs are as follows:	
			0	North Input FIFO
			1	East Input FIFO
			2	South Input FIFO
			3	West Input FIFO
			4	Processor Input FIFO
			5	Processor Output FIFO
			6 and 7	Undefined

Static Network Input State Register (SNISTATE)

This register specifies the number of entries in the static network's input FIFOs.

Speed

Slow

Minimum Protection Level

SN_ACCESS

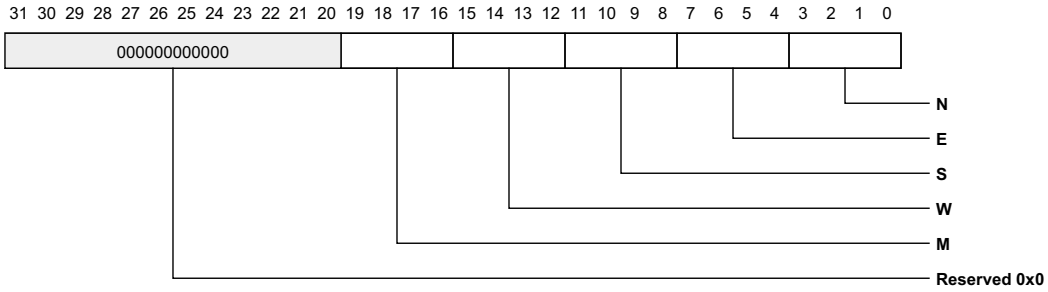

Figure 10. SNISTATE Register Diagram

Table A-15. SNISTATE Register Bit Descriptions

Bits	Name	Reset	Description
31:20	Reserved		Reserved
19:16	M	0	Main Processor Input FIFO entry count. TILE64 implements the bitfield 18:16; writes to bit 19 are ignored, and these bits are read as 0.
15:12	W	0	West Input FIFO entry count. TILE64 implements the bitfield 13:12; writes to bits 15:14 are ignored, and these bits are read as 0.
11:8	S	0	South Input FIFO entry count. TILE64 implements the bitfield 9:8; writes to bits 11:10 are ignored, and these bits are read as 0.
7:4	E	0	East Input FIFO entry count. TILE64 implements the bitfield 5:4; writes to bits 7:6 are ignored, and these bits are read as 0.
3:0	N	0	North Input FIFO entry count. TILE64 implements the bitfield 1:0; writes to bits 3:2 are ignored, and these bits are read as 0.

Static Network Output State Register (SNOSTATE)

This register specifies the number of credits available to the static network's output FIFOs on the compass points as well as the number of entries present in the output FIFO going from the static network to the processor.

Speed

Slow

Minimum Protection Level

SN_ACCESS

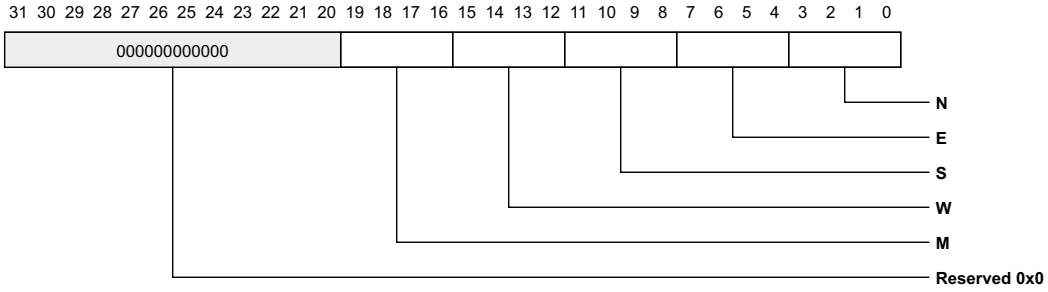

Figure 11. SNOSTATE Register Diagram

Table A-16. SNOSTATE Register Bit Descriptions

Bits	Name	Reset	Description
31:20	Reserved		Reserved
19:16	M	0	Main Processor Output FIFO entry count. TILE64 implements the bitfield 18:16; writes to bit 19 are ignored, and these bits are read as 0.
15:12	W	0	West Output FIFO credit count. TILE64 implements the bitfield 13:12; writes to bits 15:14 are ignored, and these bits are read as 0.
11:8	S	0	South Output FIFO credit count. TILE64 implements the bitfield 9:8; writes to bits 11:10 are ignored, and these bits are read as 0.
7:4	E	0	East Output FIFO credit count. TILE64 implements the bitfield 5:4; writes to bits 7:6 are ignored, and these bits are read as 0.
3:0	N	0	North Output FIFO credit count. TILE64 implements the bitfield 1:0; writes to bits 3:2 are ignored, and these bits are read as 0.

Static Network Static Route (SNSTATIC)

This register specifies the static input route to a given output port.

Speed

Slow

Minimum Protection Level

SN_ACCESS

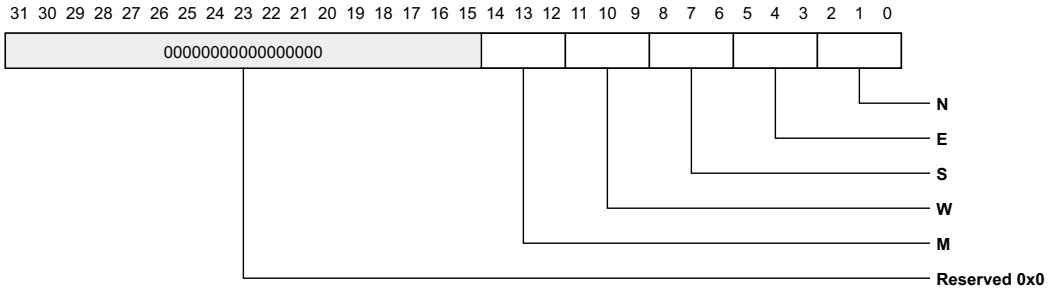

Figure A-1: SNSTATIC Register Diagram

Table A-17. SNSTATIC Register Bit Descriptions

Bits	Name	Default	Description
31:15	Reserved		
14:12	M	0	Main Processor static input route.
11:9	W	0	West static input route.
8:6	S	0	South static input route.
5:3	E	0	East static input route.
2:0	N	0	North static input route.

As shown in Table A-18, each field contains a number that specifies which input port will route to that output port:

Table A-18. Port Designations

Numbers	Input Port
0	None
1	North
2	East
3	South

Table A-18. Port Designations (continued)

Numbers	Input Port
4	West
5	Main Processor

Static Network Data Available (SN_DATA_AVAIL)

This register contains a bit field that indicates that data is available on the static network.

Speed

Fast

Minimum Protection Level

SN_ACCESS

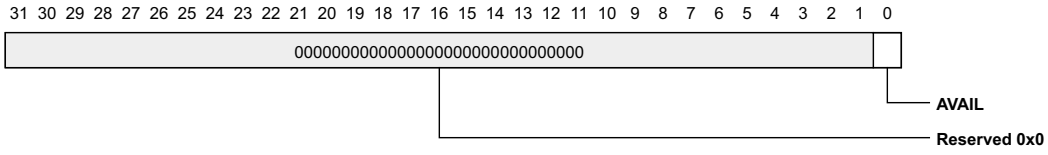

Figure A-2: SN_DATA_AVAIL Register Diagram

Table A-19. SN_DATA_AVAIL Register Bit Descriptions

Bits	Name	Default	Description
31:1	Reserved		
0	AVAIL	0	Data is available to be read on the static network.

Static Network Control (SN_STATIC_CTL)

This register controls execution of the static network processor and fabric.

NOTE: This SPR is reserved for TILE64 and is not reserved for TILEPro.

Speed

Slow

Minimum Protection Level

SN_STATIC_ACCESS

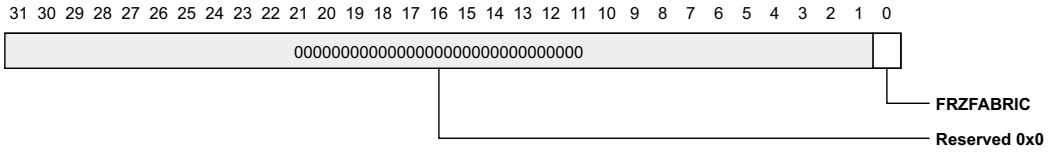

Figure 2. SN_STATIC_CTL Register Diagram

Table A-20. SN_STATIC_CTL Register Bit Descriptions

Bits	Name	Reset	Description
31:1	Reserved		Reserved
0	FRZFABRIC	1	Added in TILEPro: Freeze the static network fabric.

Static Network FIFO Data (SN_STATIC_FIFO_DATA)

Accesses the data FIFO specified by SNFIFO_SEL. When read, returns the top entry on the specified FIFO and removes it from the FIFO. When written, it writes the specified data into the FIFO.

NOTE: This SPR is reserved for TILE64 and is not reserved for TILEPro.

Speed

Slow

Minimum Protection Level

SN_STATIC_ACCESS

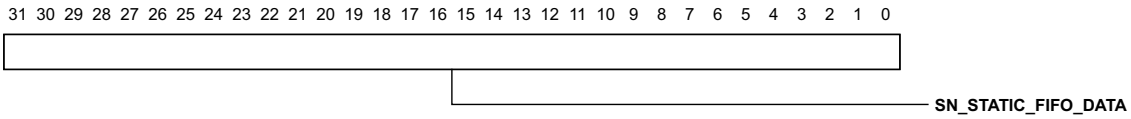

Figure 3. SN_STATIC_FIFO_DATA Register Diagram

Table A-21. SN_STATIC_FIFO_DATA Register Bit Descriptions

Bits	Name	Reset	Description
31:0	SN_STATIC _FIFO_DATA	0	Added in TILEPro: Accesses the data fifo specified by SNFIFO_SEL. When read, returns the top entry on the specified fifo and removes it from the FIFO. When written, it writes the specified data into the FIFO.

Static Network FIFO Select (SN_STATIC_FIFO_SEL)

This SPR specifies which FIFO will be read and written by the `SNFIFO_DATA` register.

NOTE: This SPR is reserved for TILE64 and is not reserved for TILEPro.

Speed

Slow

Minimum Protection Level

`SN_STATIC_ACCESS`

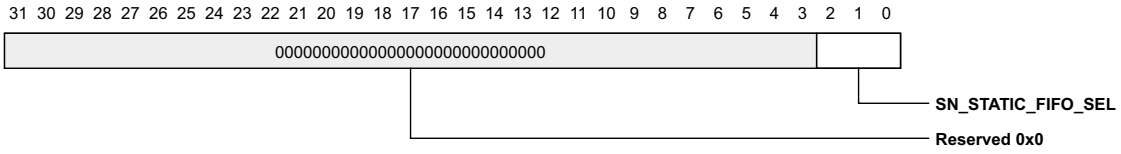

Figure 4. SN_STATIC_FIFO_SEL Register Diagram

Table A-22. SN_STATIC_FIFO_SEL Register Bit Descriptions

Bits	Name	Reset	Description
31:3	Reserved		Reserved
2:0	SN_STATIC _FIFO_SEL	0	Added in TILEPro: This bitfield specifies which FIFO will be read and written by the SNFIFO_DATA register.

Static Network Input State (SN_STATIC_ISTATE)

This register specifies the number of entries in the static network's Input FIFOs.

NOTE: This SPR is reserved for TILE64 and is not reserved for TILEPro.

Speed

Slow

Minimum Protection Level

SN_STATIC_ACCESS

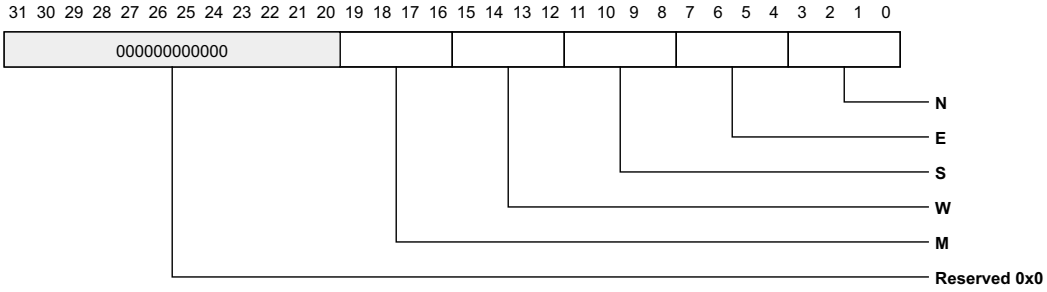

Figure 5. SN_STATIC_ISTATE Register Diagram

Table A-23. SN_STATIC_ISTATE Register Bit Descriptions

Bits	Name	Reset	Description
31:20	Reserved		Reserved
19:16	M	0	Added in TILEPro: Main Processor Input FIFO entry count.
15:12	W	0	Added in TILEPro: West Input FIFO entry count.
11:8	S	0	Added in TILEPro: South Input FIFO entry count.
7:4	E	0	Added in TILEPro: East Input FIFO entry count.
3:0	N	0	Added in TILEPro: North Input FIFO entry count.

Static Network Output State (SN_STATIC_OSTATE)

This register specifies the number of credits available to the static network's output FIFOs on the compass points as well as the number of entries present in the output FIFO going from the static network to the processor.

NOTE: This SPR is reserved for TILE64 and is not reserved for TILEPro.

Speed

Slow

Minimum Protection Level

SN_STATIC_ACCESS

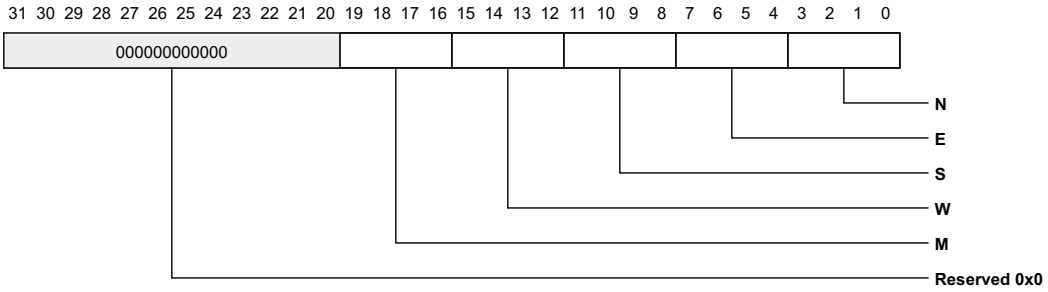

Figure 6. SN_STATIC_OSTATE Register Diagram

Table A-24. SN_STATIC_OSTATE Register Bit Descriptions

Bits	Name	Reset	Description
31:20	Reserved		Reserved
19:16	M	0	Added in TILEPro: Main Processor Output FIFO credit count.
15:12	W	0	Added in TILEPro: West Output FIFO credit count.
11:8	S	0	Added in TILEPro: South Output FIFO credit count.
7:4	E	0	Added in TILEPro: East Output FIFO credit count.
3:0	N	0	Added in TILEPro: North Output FIFO entry count.

Static Network Static Route (SN_STATIC_STATIC)

This register specifies the static input route to a given output port.

NOTE: This SPR is reserved for TILE64 and is not reserved for TILEPro.

Speed

Slow

Minimum Protection Level

SN_STATIC_ACCESS

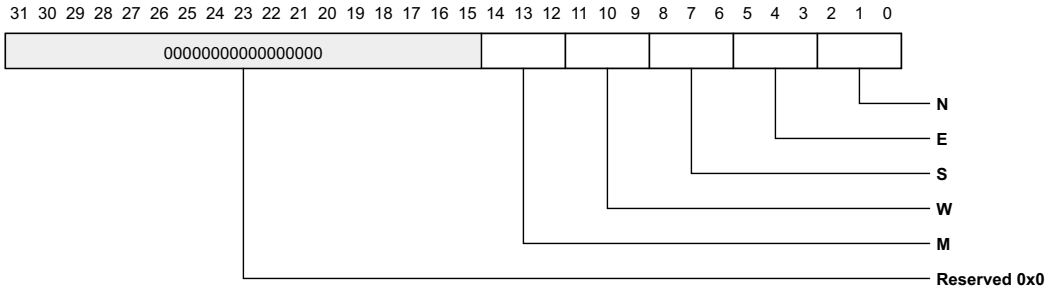

Figure 7. SN_STATIC_STATIC Register Diagram

Table A-25. SN_STATIC_STATIC Register Bit Descriptions

Bits	Name	Reset	Description
31:15	Reserved		Reserved
14:12	M	0	Added in TILEPro: Main Processor static input route.
11:9	W	0	Added in TILEPro: West static input route.
8:6	S	0	Added in TILEPro: South static input route.
5:3	E	0	Added in TILEPro: East static input route.
2:0	N	0	Added in TILEPro: North static input route.

Static Network Data Available (SN_STATIC_DATA_AVAIL)

This register contains a bit field that indicates that data is available on the static network.

NOTE: This SPR is reserved for TILE64 and is not reserved for TILEPro.

Speed

Fast

Minimum Protection Level

SN_STATIC_ACCESS

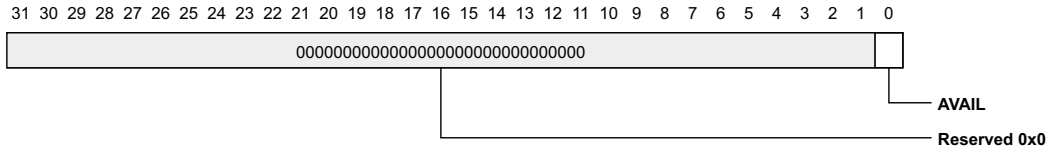

Figure 8. SN_STATIC_DATA_AVAIL Register Diagram

Table A-26. SN_STATIC_DATA_AVAIL Register Bit Descriptions

Bits	Name	Reset	Description
31:1	Reserved		Reserved
0	AVAIL	0	Added in TILEPro: This bit indicates added in TILEPro: Data is available to be read on the static network.

User Dynamic Network Catch-all Demultiplexor Count Register (UDN_DEMUX_CA_COUNT)

This register contains the number of words that have been received for Catch-all Queue of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

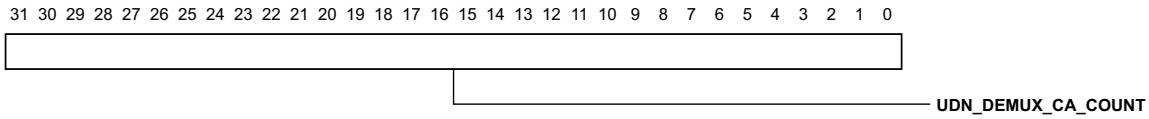

Figure 9. UDN_DEMUX_CA_COUNT Register Diagram

Table A-27. UDN_DEMUX_CA_COUNT Register Bit Descriptions

Bits	Name	Reset	Description
31:0	UDN_DEMUX_CA _COUNT	0	Number two-word slices the UDN is allowed to consume in the demux buffer. If the sum of IDN and UDN thresholds exceeds 56, the IDN and UDN networks can compete for buffer entries and the refill/context swap flows must account for concurrent activity on the other network. TILE64 implements the bitfield 6:0; writes to bits 31:7 are ignored, and these bits are read as 0.

User Dynamic Network Demultiplexor Count 0 Register (UDN_DEMUX_COUNT_0)

This register contains the number of words that have been received for channel 0 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

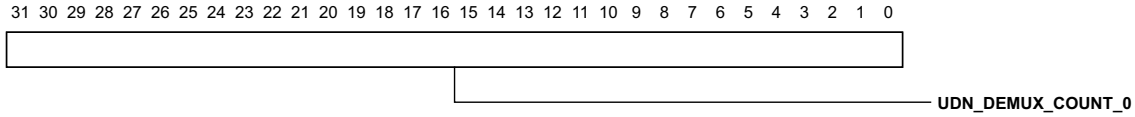

Figure 10. UDN_DEMUX_COUNT_0 Register Diagram

Table A-28. UDN_DEMUX_COUNT_0 Register Bit Descriptions

Bits	Name	Reset	Description
31:0	UDN_DEMUX _COUNT_0	0	Count. Implements the bitfield 6:0; writes to bits 31:7 are ignored, and these bits are read as 0.

User Dynamic Network Demultiplexor Count 1 Register (UDN_DEMUX_COUNT_1)

This register contains the number of words that have been received for channel 1 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

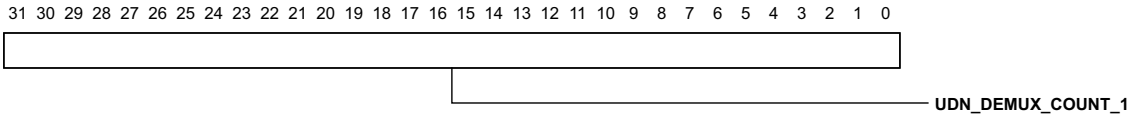

Figure 11. UDN_DEMUX_COUNT_1 Register Diagram

Table A-29. UDN_DEMUX_COUNT_1 Register Bit Descriptions

Bits	Name	Reset	Description
31:0	UDN_DEMUX _COUNT_1	0	Count. Implements the bitfield 6:0; writes to bits 31:7 are ignored, and these bits are read as 0.

User Dynamic Network Demultiplexor Count 2 Register (UDN_DEMUX_COUNT_2)

This register contains the number of words that have been received for channel 2 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

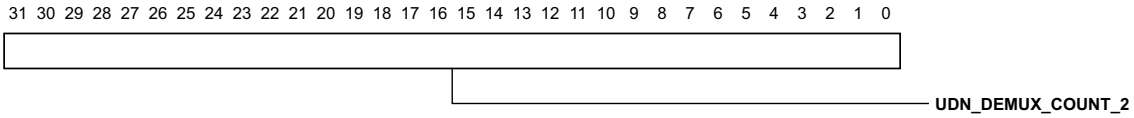

Figure 12. UDN_DEMUX_COUNT_2 Register Diagram

Table A-30. UDN_DEMUX_COUNT_2 Register Bit Descriptions

Bits	Name	Reset	Description
31:0	UDN_DEMUX _COUNT_2	0	Count. Implements the bitfield 6:0; writes to bits 31:7 are ignored, and these bits are read as 0.

User Dynamic Network Demultiplexor Count 3 Register (UDN_DEMUX_COUNT_3)

This register contains the number of words that have been received for channel 3 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

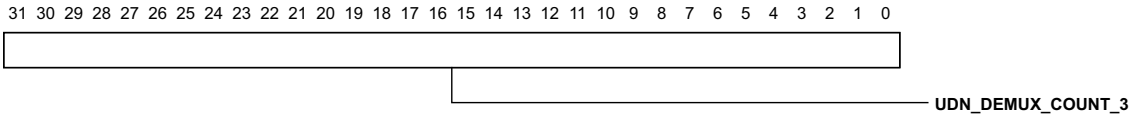

Figure 13. UDN_DEMUX_COUNT_3 Register Diagram

Table A-31. UDN_DEMUX_COUNT_3 Register Bit Descriptions

Bits	Name	Reset	Description
31:0	UDN_DEMUX _COUNT_3	0	Count. Implements the bitfield 6:0; writes to bits 31:7 are ignored, and these bits are read as 0.

UDN Demux Control Register (UDN_DEMUX_CTL)

When written, demux state is cleared. Used after state extraction and during state restore.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Demux Current Tag (UDN_DEMUX_CURR_TAG)

This register contains the tag of current packet being dequeued. This register is valid only when the CURR_REM field is not 0 in the UDN_DEMUX_STATUS register.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

UDN Demux Queue Select Register (UDN_DEMUX_QUEUE_SEL)

Selects demux queue to be written on UDN_DEMUX_WRITE_QUEUE.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

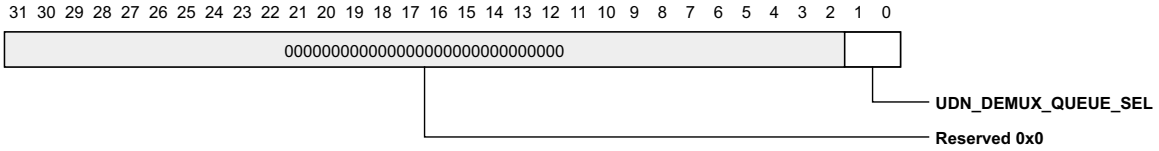

Figure 14. UDN_DEMUX_QUEUE_SEL Register Diagram

Table A-32. UDN_DEMUX_QUEUE_SEL Register Bit Descriptions

Bits	Name	Reset	Description
31:2	Reserved		Reserved
1:0	UDN_DEMUX _QUEUE_SEL	0	Selects demux queue to be written on UDN_DEMUX_WRITE_QUEUE.

User Dynamic Network Demux FIFO (UDN_DEMUX_WRITE_FIFO)

When this register is written to, one word of data is pushed into demux FIFO. When this register is read, one word is read from FIFO.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Demux State (UDN_DEMUX_STATUS)

This register enables access to the demux logic state for context swapping, deadlock recovery information, and tag changes.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

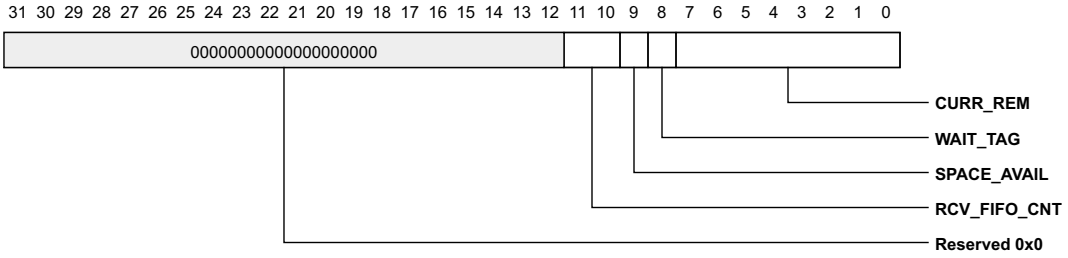

Figure A-1: UDN_DEMUX_STATUS Register Diagram

Table A-33. UDN_DEMUX_STATUS Register Bit Descriptions

Bits	Name	Default	Description
31:12	Reserved		
11:10	RCV_FIFO_CNT		Number of entries in the receive FIFO.
9	SPACE_AVAIL		Space is available in the demux framing logic for at least one word.
8	WAIT_TAG		Currently waiting for tag word. State save/restore should ignore current tag and not refill.
7:0	CURR_REM		Number of words remaining in packet currently being dequeued. When 0, no packet inflight.

User Dynamic Network Demux Write Queue (UDN_DEMUX_WRITE_QUEUE)

When this register is written to, one word of data is pushed into demux queue selected by QUEUE_SEL — used to push data into queues that are in refill mode.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Words Pending (UDN_PENDING)

This register contains the number of words remaining in packet being sent into network from the main processor.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

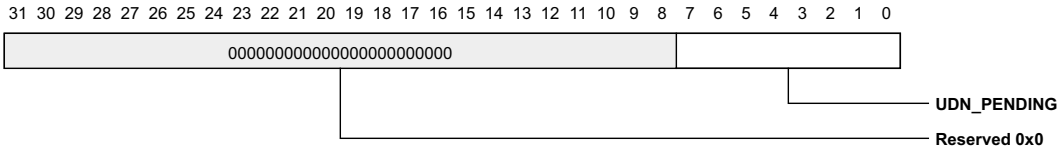

Figure A-2: UDN_PENDING Register Diagram

Table A-34. UDN_PENDING Register Bit Descriptions

Bits	Name	Default	Description
31:8	Reserved		
7:0	UDN_PENDING		The number of words remaining in packet being sent into network from the main processor.

User Dynamic Network FIFO Data (UDN_SP_FIFO_DATA)

This register provide access to the data FIFO specified by UDN_SP_FIFO_SEL. When this register is read, it returns the top entry on the specified FIFO and removes the entry from the FIFO. When this register is written to, it writes the specified data into the FIFO.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network FIFO Select (UDN_SP_FIFO_SEL)

This register specifies which port's data and state will be read and written by the UDN_SP_FIFO_DATA and UDN_SP_STATE registers. When set to 4(d), main processor FIFO may be restored by writing to the network register. Data may be lost if left set to 4(d) and switch point is frozen and too much data is written to egress FIFO.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

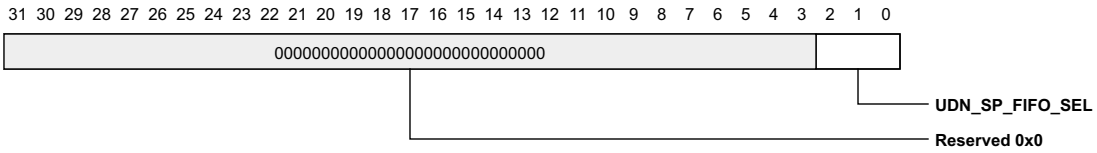

Figure A-3: UDN_SP_FIFO_SEL Register Diagram

Table A-35. UDN_SP_FIFO_SEL Register Bit Descriptions

Bits	Name	Reset	Description
31:3	Reserved		Reserved
2:0	UDN_SP_FIFO_SEL	0	Specifies which port's data and state will be read and written by the UDN SP FIFO DATA and UDN SP STATE registers. When set to 4(d), processor FIFO may be restored by writing to the network register. Data may be lost if left set to 4(d) and switch point is frozen and too much data is written to egress FIFO. The encodings are: 0 North 1 South 2 East 3 West 4 cORE

User Dynamic Network Freeze (UDN_SP_FREEZE)

This register freezes the network in preparation for context swap.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

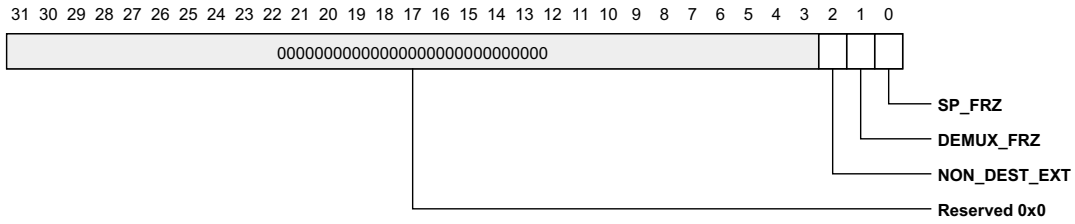

Figure A-4: UDN_SP_FREEZE Register Diagram

Table A-36. UDN_SP_FREEZE Register Bit Descriptions

Bits	Name	Default	Description
31:3	Reserved		
2	NON_DEST_EXT	0	When asserted, the tile will return credit to neighbors when data is extracted from FIFOs via SPR reads. This is used for extracting data in the protection-violation case.
1	DEMUX_FRZ	0	Freeze demux.
0	SP_FRZ	0	Freeze Switchpoint.

User Dynamic Network Port State (UDN_SP_STATE)

This register accesses the switch point state for the port specified by `UDN_SP_FIFO_SEL`. When read, returns the associated port state. When written, it writes the specified data into the FIFO.

Speed

Slow

Minimum Protection Level

`UDN_ACCESS`

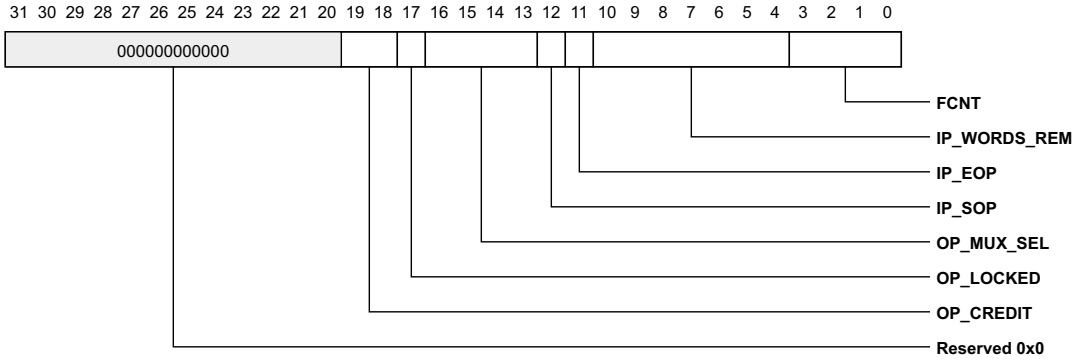

Figure A-5: UDN_SP_STATE Register Diagram

Table A-37. UDN_SP_STATE Register Bit Descriptions

Bits	Name	Default	Description
31:20	Reserved		
19:18	OP_CREDIT		Number of credits at the output port available to send packet words to neighbor.
17	OP_LOCKED		Output port is currently locked on a given input port (mid-packet).
16:13	OP_MUX_SEL		Input port being selected for output port. Bit[0] is the default route (South input port for North output port for example). the remaining bits walk around compass clockwise starting from default route, skipping output port. The core port is between South and West for this algorithm. For the North output port, bits are: 3 East 2 West 1 Core 0 South.
12	IP_SOP		The next word to be dequeued is the route header for a new packet.

Table A-37. UDN_SP_STATE Register Bit Descriptions (continued)

Bits	Name	Default	Description
11	IP_EOP		Next word to be dequeued is the last word in a packet.
10:4	IP_WORDS_REM		The number of words remaining in packet being dequeued from input port. When 0 and IP_SOP = 1, no packet is being dequeued. When 0 and IP_SOP is 0, there are 128 words remaining in the packet.
3:0	FCNT		The number of valid entries in the associated FIFO.

User Dynamic Network Tag 0 (UDN_TAG_0)

This register contains the tag for channel 0 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Tag 1 (UDN_TAG_1)

This register contains the tag for channel 1 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Tag 2 (UDN_TAG_2)

This register contains the tag for channel 2 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Tag 3 (UDN_TAG_3)

This register contains the tag for channel 3 of the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Tag Valid (UDN_TAG_VALID)

This register specifies which tags are valid for the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

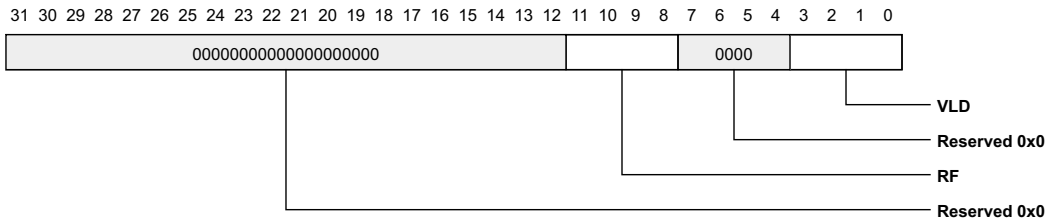

Figure A-6: UDN_TAG_VALID Register Diagram

Table A-38. UDN_TAG_VALID Register Bit Descriptions

Bits	Name	Default	Description
31:12	Reserved		
11:8	RF	0	Refill Mode
7:4	Reserved		
3:0	VLD	0	Tag Valid

User Dynamic Network Tile Coordinates (UDN_TILE_COORD)

This register contains the tile coordinates for the User Dynamic Network.

Speed

Slow

Minimum Protection Level

UDN_ACCESS

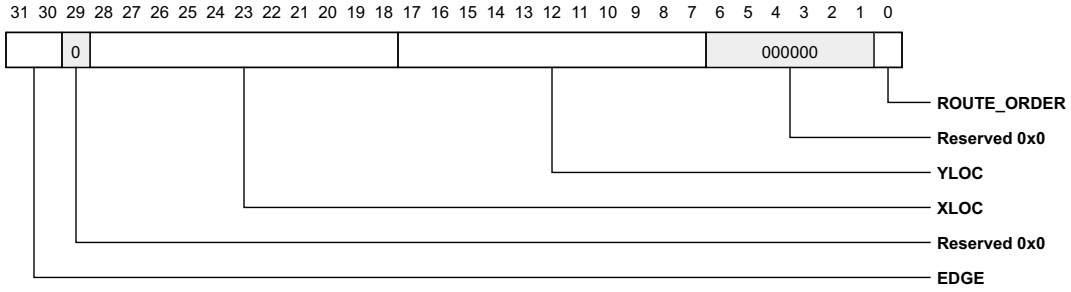

Figure A-7: UDN_TILE_COORD Register Diagram

Table A-39. UDN_TILE_COORD Register Bit Descriptions

Bits	Name	Reset	Description
31:30	EDGE	0	Edge.
29	Reserved		Reserved.
28:18	XLOC	1	X location.
17:7	YLOC	1	Y location.
6:1	Reserved		Reserved
0	ROUTE_ORDER	0	For TILEPro: 0 When 0, packets are routed in the X dimension first followed by the Y dimension. 1 When 1, the Y dimension is routed first.

User Dynamic Network Catch-All Data (UDN_CA_DATA)

This register contains the next word to be read from the message at the head of the Catch-all Queue. Reading this register dequeues from the Catch-all Queue.

Speed

Fast

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Catch-all Remaining Words (UDN_CA_REM)

This register contains the number of words remaining to be read from the message at the head of the Catch-all Queue.

Speed

Fast

Minimum Protection Level

UDN_ACCESS

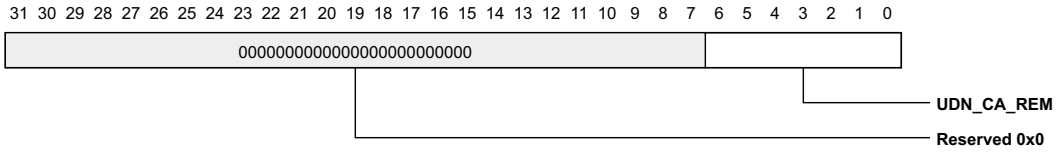

Figure A-8: UDN_CA_REM Register Diagram

Table A-40. UDN_CA_REM Register Bit Descriptions

Bits	Name	Default	Description
31:7	Reserved		
6:0	UDN_CA_REM	0	This register contains the number of words remaining to be read from the message at the head of the Catch-all Queue. When no message is in the Catch-all Queue, this field is 0.

User Dynamic Network Catch-all Tag (UDN_CA_TAG)

This register contains the tag the message at the head of the Catch-all Queue.

Speed

Fast

Minimum Protection Level

UDN_ACCESS

User Dynamic Network Data Available (UDN_DATA_AVAIL)

This register contains bit fields that indicate that data is available on particular User Dynamic Network demultiplexor ports.

Speed

Fast

Minimum Protection Level

UDN_ACCESS

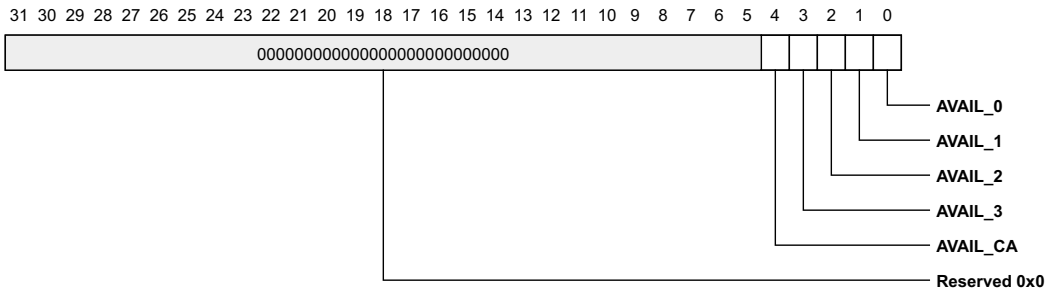

Figure A-9: UDN_DATA_AVAIL Register Diagram

Table A-41. UDN_DATA_AVAIL Register Bit Descriptions

Bits	Name	Default	Description
31:5	Reserved		
4	AVAIL_CA	0	Data is available to be read on UDN catch-all queue.
3	AVAIL_3	0	Data is available to be read on UDN demultiplexor port 3.
2	AVAIL_2	0	Data is available to be read on UDN demultiplexor port 2.
1	AVAIL_1	0	Data is available to be read on UDN demultiplexor port 1.
0	AVAIL_0	0	Data is available to be read on UDN demultiplexor port 0.

User Dynamic Network Refill Available Enable (UDN_REFILL_EN)

This register controls whether or not a particular UDN input port signals the UDN refill interrupt when data is available.

Speed

Slow

Minimum Protection Level

UDN_REFILL

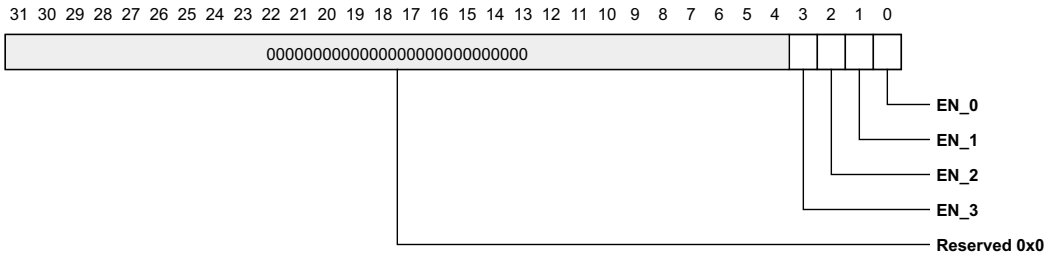

Figure A-10: UDN_REFILL_EN Register Diagram

Table A-42. UDN_REFILL_EN Register Bit Descriptions

Bits	Name	Default	Description
31:4	Reserved		
3	EN_3	0	Enable UDN 3 Refill Interrupt
2	EN_2	0	Enable UDN 2 Refill Interrupt
1	EN_1	0	Enable UDN 1 Refill Interrupt
0	EN_0	0	Enable UDN 0 Refill Interrupt

User Dynamic Network Remaining (UDN_REMAINING)

This register controls how many words remain to be written until the UDN complete interrupt is signaled.

Speed

Slow

Minimum Protection Level

UDN_COMPLETE

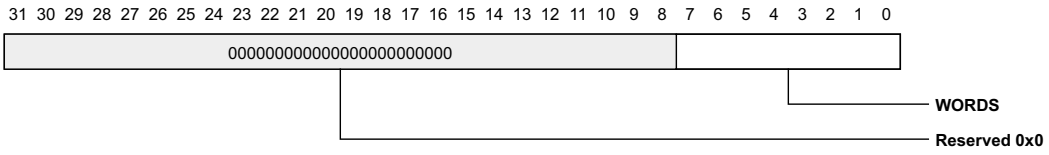

Figure A-11: UDN_REMAINING Register Diagram

Table A-43. UDN_REMAINING Register Bit Descriptions

Bits	Name	Default	Description
31:8	Reserved		
7:0	WORDS	0	Number of words left to be written

User Dynamic Network Available Enables (UDN_AVAIL_EN)

This register controls whether or not a particular UDN input port signals the UDN available interrupt when data is available.

Speed

Slow

Minimum Protection Level

UDN_AVAIL

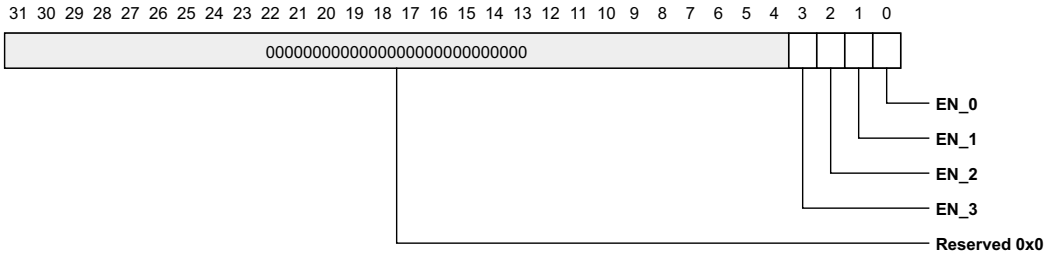

Figure A-12: UDN_AVAIL_EN Register Diagram

Table A-44. UDN_AVAIL_EN Register Bit Descriptions

Bits	Name	Default	Description
31:4	Reserved		
3	EN_3		Enable UDN 3 Available Interrupt
2	EN_2		Enable UDN 2 Available Interrupt
1	EN_1		Enable UDN 1 Available Interrupt
0	EN_0		Enable UDN 0 Available Interrupt

User Dynamic Network Deadlock Counter (UDN_DEADLOCK_COUNT)

This register is used to save/restore current state of deadlock down-counter.

Speed

Slow

Minimum Protection Level

UDN_TIMER

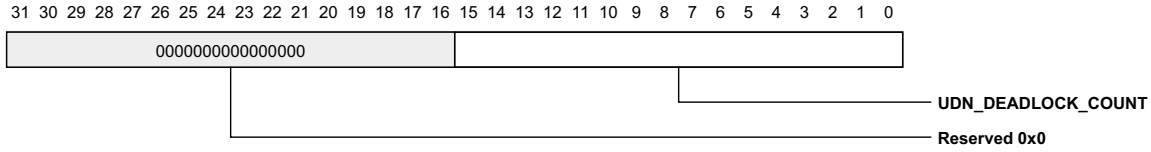

Figure A-13: UDN_DEADLOCK_COUNT Register Diagram

Table A-45. UDN_DEADLOCK_COUNT Register Bit Descriptions

Bits	Name	Default	Description
31:16	Reserved		
15:0	UDN_DEADLOCK_COUNT	0	UDN deadlock count

User Dynamic Network Deadlock Timeout (UDN_DEADLOCK_TIMEOUT)

This register provides the number of 16-cycle intervals to wait before asserting the deadlock interrupt when data is stalled in the demux logic's dequeueing buffer.

Speed

Slow

Minimum Protection Level

UDN_TIMER

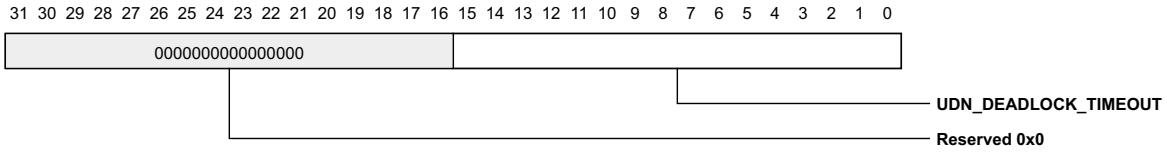

Figure A-14: UDN_DEADLOCK_TIMEOUT Register Diagram

Table A-46. UDN_DEADLOCK_TIMEOUT Register Bit Descriptions

Bits	Name	Default	Description
31:16	Reserved		
15:0	UDN_DEADLOCK _TIMEOUT	0	UDN Deadlock Timeout

Cycle Counter High (CYCLE_HIGH)

This register contains the top 32 bits of the 64 bit cycle counter. The cycle counter is incremented every machine cycle.

Speed

Slow

MPL

WORLD_ACCESS

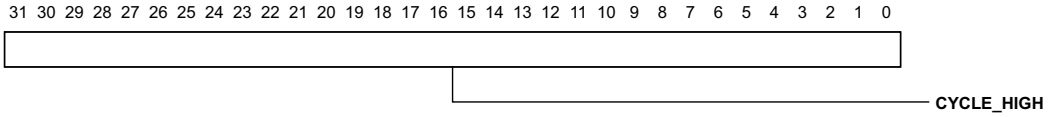

Figure A-15: CYCLE_HIGH Register Diagram

Cycle Counter Low (CYCLE_LOW)

This register contains the bottom 32 bits of the 64 bit cycle counter. The cycle counter is incremented every machine cycle.

Speed

Slow

MPL

WORLD_ACCESS

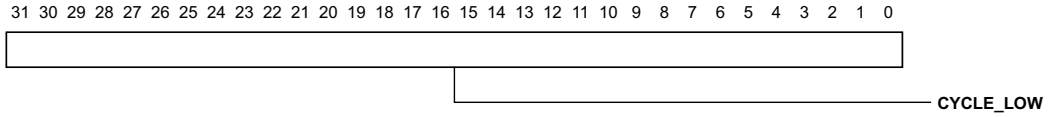

Figure A-16: CYCLE_LOW Register Diagram

Done Magic Register (DONE)

A magic register that is used to signal completion information to the test system. PASS/FAIL/DONE share a 32-bit storage element.

Speed

Slow

MPL

WORLD_ACCESS

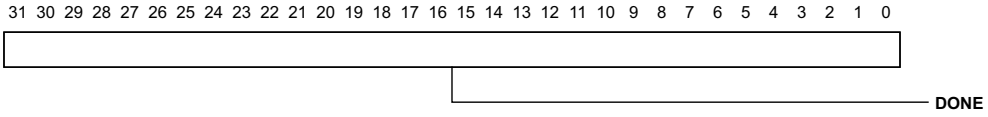

Figure A-17: DONE Register Diagram

Fail Magic Register (FAIL)

A magic register that is used to signal failure information to the test system. PASS/FAIL/DONE share a 32-bit storage element.

Speed

Slow

MPL

WORLD_ACCESS

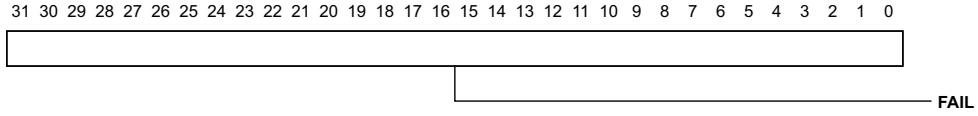

Figure A-18: FAIL Register Diagram

Interrupt Critical Section (INTERRUPT_CRITICAL_SECTION)

This register specifies whether or not the main processor is in an interrupt critical section. This register is used by interrupts and iret instructions.

Speed

Slow

MPL

WORLD_ACCESS

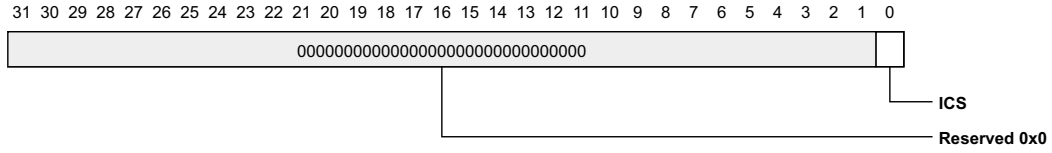

Figure A-19: INTERRUPT_CRITICAL_SECTION Register Diagram

Pass Magic Register (PASS)

A magic register that is used to pass information to the test system. PASS/FAIL/DONE share a 32-bit storage element.

Speed

Slow

MPL

WORLD_ACCESS

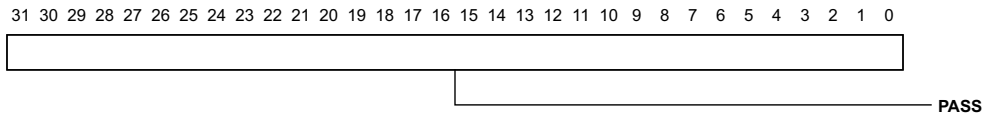

Figure A-20: PASS Register Diagram

Exceptional Context Protection Level 0 Entry 0 (EX_CONTEXT_0_0)

This register specifies the first part of the exceptional context for protection level 0. This register is used by interrupts and iret instructions.

Speed

Slow

Minimum Protection Level

INTCTRL_0

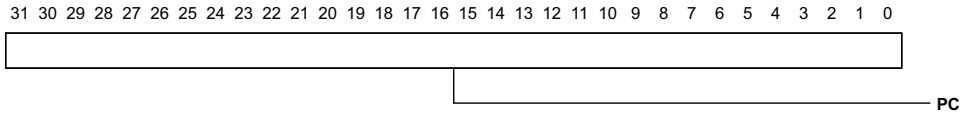

Figure A-21: EX_CONTEXT_0_0 Register Diagram

Table A-47. EX_CONTEXT_0_0 Register Bit Descriptions

Bits	Name	Default	Description
31:0	PC	0	The program counter for the context that was interrupted.

Exceptional Context Protection Level 0 Entry 1 (EX_CONTEXT_0_1)

This register specifies the second part of the exceptional context for protection level 0. This register is used by interrupts and iret instructions.

Speed

Slow

Minimum Protection Level

INTCTRL_0

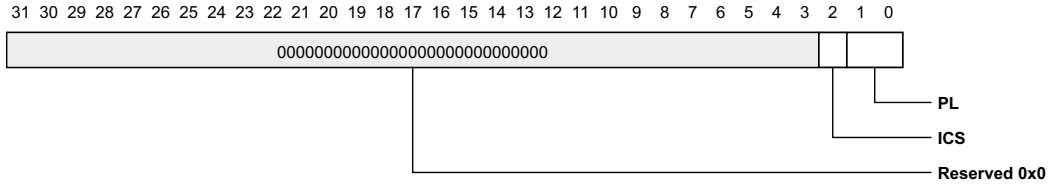

Figure A-22: EX_CONTEXT_0_1 Register Diagram

Table A-48. EX_CONTEXT_0_1 Register Bit Descriptions

Bits	Name	Default	Description
31:3	Reserved	0	Reserved
2	ICS	0	Interrupt Critical Section. This bit indicates if the interrupted context is in an interrupt critical section.
1:0	PL	0	Protection Level. This field provides the protection level for the context that was interrupted.

Interrupt Control 0 Status (INTCTRL_N_STATUS)

This register is used to specify the interrupt control 0 interrupt.

Speed

Slow

Minimum Protection Level

INTCTRL_0

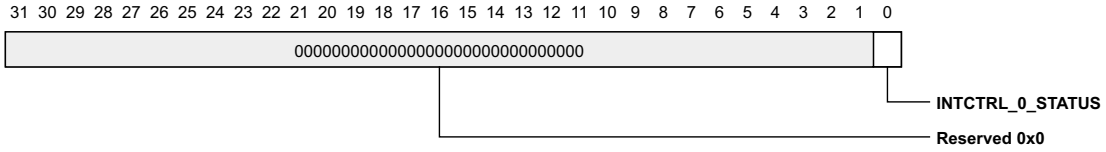

Figure A-23: INTCTRL_0_STATUS Register Diagram

Table A-49. INTCTRL_0_STATUS Register Bit Descriptions

Bits	Name	Default	Description
31:1	Reserved	0	Reserved
0	INTCTRL_N_STATUS	0	This field specifies the interrupt control N interrupt.

Interrupt Mask Protection Level 0 Entry 0 (INTERRUPT_MASK_0_0)

This register is used to mask (disable) interrupts. A value of 1 in a bit position masks the interrupt and a value of 0 enables the interrupt. This register specifies the interrupt mask for interrupts 0 through 31 (see Table 8-11 on page 386 for the mapping of interrupt numbers).

Speed

Slow

Minimum Protection Level

INTCTRL_0

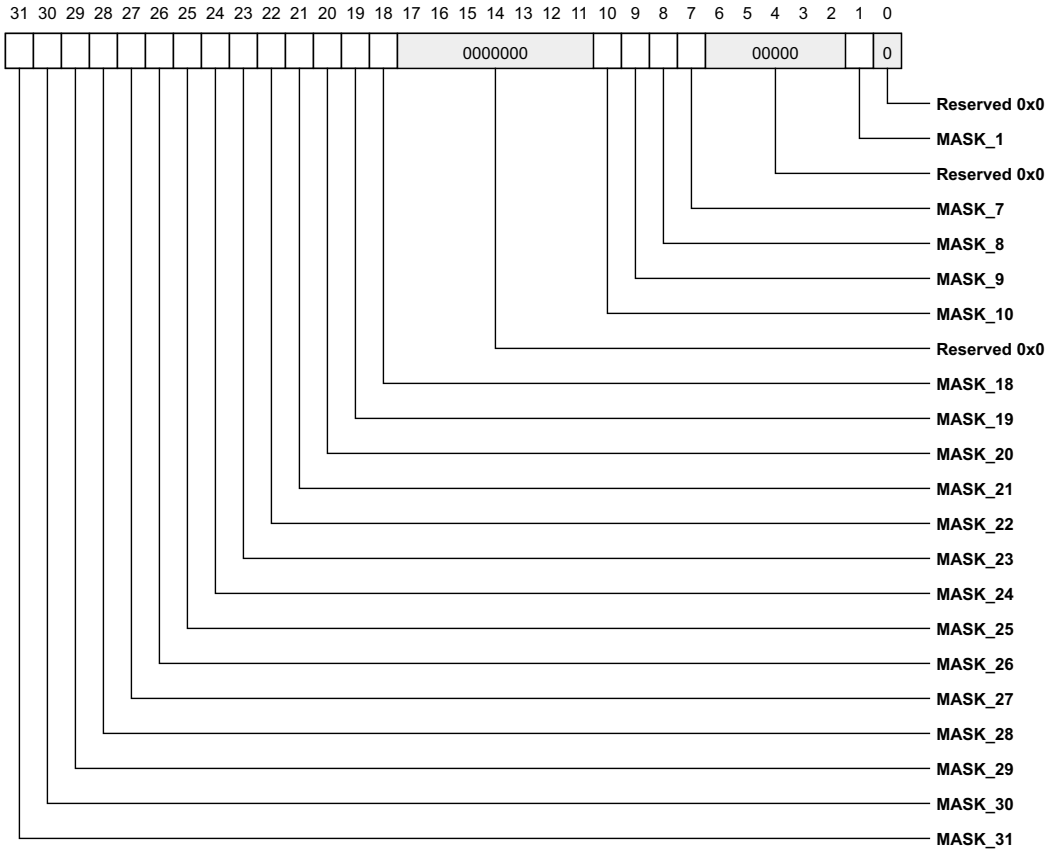

Figure A-24: INTERRUPT_MASK_0_0 Register Diagram

Table A-50. INTERRUPT_MASK_0_0 Register Bit Descriptions

Bits	Name	Default	Description
31	MASK_31	1	A value of 1 disables the IDN_AVAIL interrupt.
30	MASK_30	1	A value of 1 disables the UDN_CA interrupt.

Table A-50. INTERRUPT_MASK_0_0 Register Bit Descriptions (continued)

Bits	Name	Default	Description
29	MASK_29	1	A value of 1 disables the IDN_CA interrupt.
28	MASK_28	1	A value of 1 disables the DMA_NOTIFY interrupt.
27	MASK_27	1	A value of 1 disables the UDN_TIMER interrupt.
26	MASK_26	1	A value of 1 disables the IDN_TIMER interrupt.
25	MASK_25	1	A value of 1 disables the TILE_TIMER interrupt.
24	MASK_24	1	A value of 1 disables the UDN_FIREWALL interrupt.
23	MASK_23	1	A value of 1 disables the IDN_FIREWALL interrupt.
22	MASK_22	1	A value of 1 disables the SN_FIREWALL interrupt.
21	MASK_21	1	Reserved
20	MASK_20	1	Reserved
19	MASK_19	1	A value of 1 disables the DMATLB_ACCESS interrupt.
18	MASK_18	1	A value of 1 disables the DMATLB_MISS interrupt.
17:11	Reserved	0	Reserved
10	MASK_10	1	A value of 1 disables the UDN_COMPLETE interrupt.
9	MASK_9	1	A value of 1 disables the IDN_COMPLETE interrupt.
8	MASK_8	1	A value of 1 disables the UDN_REFILL interrupt.
7	MASK_7	1	A value of 1 disables the IDN_REFILL interrupt.
6:2	Reserved	0	Reserved
1	MASK_1	1	A value of 1 disables the MEM_ERROR interrupt.
0	Reserved	0	Reserved

Interrupt Mask Protection Level 0 Entry 1 (INTERRUPT_MASK_0_1)

This register is used to mask (disable) interrupts. A value of 1 in a bit position masks the interrupt and a value of 0 enables the interrupt. This register specifies the interrupt mask for interrupts 32 through 37 (see Table 8-11 on page 386 for the mapping of interrupt numbers).

Speed

Slow

Minimum Protection Level

INTCTRL_0

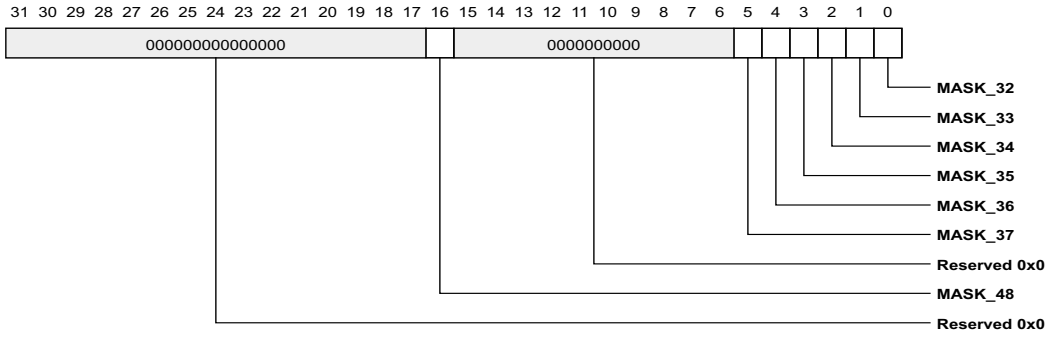

Figure A-25: INTERRUPT_MASK_0_1 Register Diagram

Table A-51. INTERRUPT_MASK_0_1 Register Bit Descriptions

Bits	Name	Reset	Description
31:6	Reserved		Reserved
31:17	Reserved	0	Reserved
16	MASK_48	1	Added in TILEPro: A value of 1 disables the AUX_PERF_COUNT interrupt.
15	MASK_47	1	Added in TILEPro: A value of 1 disables the SN_STATIC_ACCESS interrupt.
15:6	Reserved		Reserved
5	MASK_37	1	A value of 1 disables the INTCTRL_0 interrupt.
4	MASK_36	1	A value of 1 disables the INTCTRL_1 interrupt.
3	MASK_35	1	A value of 1 disables the INTCTRL_2 interrupt.
2	MASK_34	1	A value of 1 disables the INTCTRL_3 interrupt.
1	MASK_33	1	A value of 1 disables the PERF_COUNT interrupt.
0	MASK_32	1	A value of 1 disables the UDN_AVAIL interrupt.

Interrupt Mask Protection Level 0 Entry 0 (INTERRUPT_MASK_RESET_0)

This register is used to clear bits in the interrupt mask. Writing a value of 1 to a bit position resets the interrupt mask for that position. Writing a value of 0 to a bit position has no effect. This register clears the interrupt mask for interrupts 0 through 31 (see Table 8-11 on page 386 for the mapping of interrupt numbers).

Speed

Slow

Minimum Protection Level

INTCTRL_0

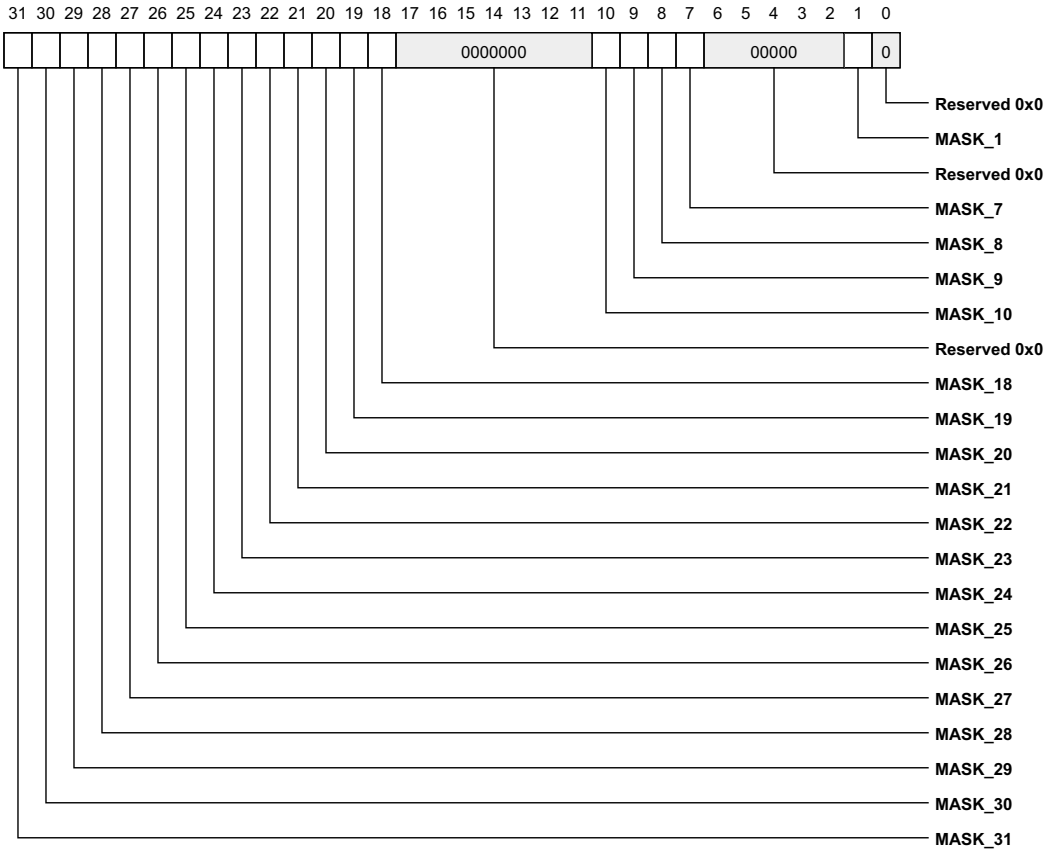

Figure A-26: INTERRUPT_MASK_RESET_0 Register Diagram

Table A-52. INTERRUPT_MASK_RESET_0 Register Bit Descriptions

Bits	Name	Default	Description
31	MASK_31	1	A value of 1 enables the IDN_AVAIL interrupt.

Table A-52. INTERRUPT_MASK_RESET_0 Register Bit Descriptions (continued)

Bits	Name	Default	Description
30	MASK_30	1	A value of 1 enables the UDN_CA interrupt.
29	MASK_29	1	A value of 1 enables the IDN_CA interrupt.
28	MASK_28	1	A value of 1 enables the DMA_NOTIFY interrupt.
27	MASK_27	1	A value of 1 enables the UDN_TIMER interrupt.
26	MASK_26	1	A value of 1 enables the IDN_TIMER interrupt.
25	MASK_25	1	A value of 1 enables the TILE_TIMER interrupt.
24	MASK_24	1	A value of 1 enables the UDN_FIREWALL interrupt.
23	MASK_23	1	A value of 1 enables the IDN_FIREWALL interrupt.
22	MASK_22	1	A value of 1 enables the SN_FIREWALL interrupt.
21	MASK_21	1	Reserved
20	MASK_20	1	Reserved
19	MASK_19	1	A value of 1 enables the DMATLB_ACCESS interrupt.
18	MASK_18	1	A value of 1 enables the DMATLB_MISS interrupt.
17:11	Reserved	0	Reserved
10	MASK_10	1	A value of 1 enables the UDN_COMPLETE interrupt.
9	MASK_9	1	A value of 1 enables the IDN_COMPLETE interrupt.
8	MASK_8	1	A value of 1 enables the UDN_REFILL interrupt.
7	MASK_7	1	A value of 1 enables the IDN_REFILL interrupt.
6:2	Reserved	0	Reserved
1	MASK_1	1	A value of 1 enables the MEM_ERROR interrupt.
0	Reserved	0	Reserved

Interrupt Mask Protection Level 0 Entry 1 (INTERRUPT_MASK_RESET_0_1)

This register is used to clear bits in the interrupt mask. Writing a value of 1 to a bit position resets the interrupt mask for that position. Writing a value of 0 to a bit position has no effect. This register clears the interrupt mask for interrupts 32 through 37 (see Table 8-11 on page 386 for the mapping of interrupt numbers).

Speed

Slow

Minimum Protection Level

INTCTRL_0

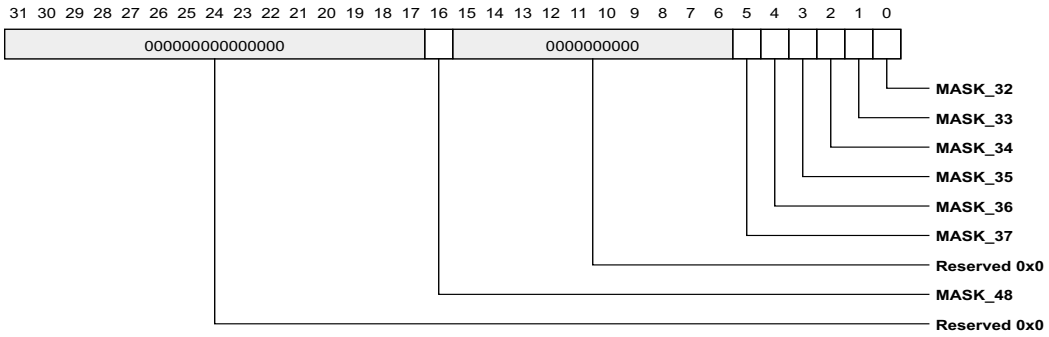

Figure A-27: INTERRUPT_MASK_RESET_0_1 Register Diagram

Table A-53. INTERRUPT_MASK_RESET_0_1 Register Bit Descriptions

Bits	Name	Reset	Description
31:17	Reserved		Reserved
16	MASK_48	1	Added in TILEPro: A value of 1 enables the AUX_PERF_COUNT interrupt.
15	MASK_47	1	Added in TILEPro: A value of 1 enables the SN_STATIC_ACCESS interrupt.
15:6	Reserved		Reserved
5	MASK_37	1	A value of 1 enables the INTCTRL_0 interrupt.
4	MASK_36	1	A value of 1 enables the INTCTRL_1 interrupt.
3	MASK_35	1	A value of 1 enables the INTCTRL_2 interrupt.
2	MASK_34	1	A value of 1 enables the INTCTRL_3 interrupt.
1	MASK_33	1	A value of 1 enables the PERF_COUNT interrupt.
0	MASK_32	1	A value of 1 enables the UDN_AVAIL interrupt.

Interrupt Mask Protection Level 0 Entry 0 (INTERRUPT_MASK_SET_0_0)

This register is used to set bits in the interrupt mask. Writing a value of 1 to a bit position sets the interrupt mask for that position. Writing a value of 0 to a bit position has no effect. This register sets the interrupt mask for interrupts 0 through 31 (see Table 8-11 on page 386 for the mapping of interrupt numbers).

Speed

Slow

Minimum Protection Level

INTCTRL_0

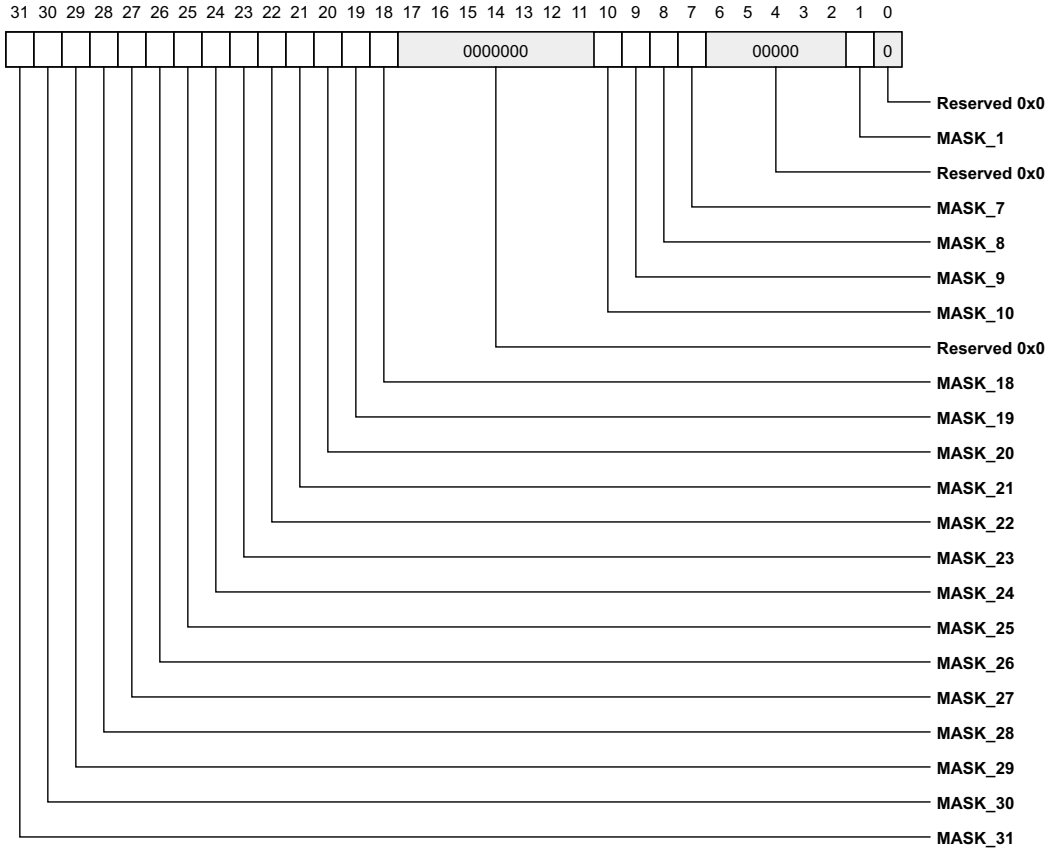

Figure A-28: INTERRUPT_MASK_SET_0_0 Register Diagram

Table A-54. INTERRUPT_MASK_SET_0_0 Register Bit Descriptions

Bits	Name	Default	Description
31	MASK_31	1	A value of 1 disables the IDN_AVAIL interrupt.

Table A-54. INTERRUPT_MASK_SET_0_0 Register Bit Descriptions (continued)

Bits	Name	Default	Description
30	MASK_30	1	A value of 1 disables the UDN_CA interrupt.
29	MASK_29	1	A value of 1 disables the IDN_CA interrupt.
28	MASK_28	1	A value of 1 disables the DMA_NOTIFY interrupt.
27	MASK_27	1	A value of 1 disables the UDN_TIMER interrupt.
26	MASK_26	1	A value of 1 disables the IDN_TIMER interrupt.
25	MASK_25	1	A value of 1 disables the TILE_TIMER interrupt.
24	MASK_24	1	A value of 1 disables the UDN_FIREWALL interrupt.
23	MASK_23	1	A value of 1 disables the IDN_FIREWALL interrupt.
22	MASK_22	1	A value of 1 disables the SN_FIREWALL interrupt.
21	MASK_21	1	Reserved
20	MASK_20	1	Reserved
19	MASK_19	1	A value of 1 disables the DMATLB_ACCESS interrupt.
18	MASK_18	1	A value of 1 disables the DMATLB_MISS interrupt.
17:11	Reserved	0	Reserved
10	MASK_10	1	A value of 1 disables the UDN_COMPLETE interrupt.
9	MASK_9	1	A value of 1 disables the IDN_COMPLETE interrupt.
8	MASK_8	1	A value of 1 disables the UDN_REFILL interrupt.
7	MASK_7	1	A value of 1 disables the IDN_REFILL interrupt.
6:2	Reserved	0	Reserved
1	MASK_1	1	A value of 1 disables the MEM_ERROR interrupt.
0	Reserved	0	Reserved

Interrupt Mask Protection Level 0 Entry 1 (INTERRUPT_MASK_SET_0_1)

This register is used to set bits in the interrupt mask. Writing a value of 1 to a bit position sets the interrupt mask for that position. Writing a value of 0 to a bit position has no effect. This register sets the interrupt mask for interrupts 32 through 37 (see Table 8-11 on page 386 for the mapping of interrupt numbers).

Speed

Slow

Minimum Protection Level

INTCTRL_0

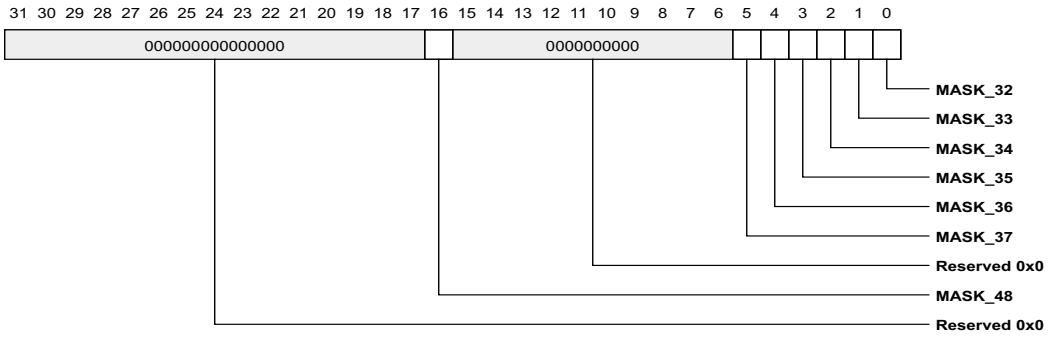

Figure A-29: INTERRUPT_MASK_SET_0_1 Register Diagram

Table A-55. INTERRUPT_MASK_SET_0_1 Register Bit Descriptions

Bits	Name	Reset	Description
31:17	Reserved		Reserved
16	MASK_48	1	Added in TILEPro: A value of 1 disables the AUX_PERF_COUNT interrupt.
15	MASK_47	1	Added in TILEPro: A value of 1 disables the SN_STATIC_ACCESS interrupt.
15:6	Reserved		Reserved
5	MASK_37	1	A value of 1 disables the INTCTRL_0 interrupt.
4	MASK_36	1	A value of 1 disables the INTCTRL_1 interrupt.
3	MASK_35	1	A value of 1 disables the INTCTRL_2 interrupt.
2	MASK_34	1	A value of 1 disables the INTCTRL_3 interrupt.
1	MASK_33	1	A value of 1 disables the UDN_AVAIL interrupt.
0	MASK_32	1	A value of 1 disables the UDN_AVAIL interrupt.

System Save Register Level 0 Entry 0 (SYSTEM_SAVE_0_0)

This register is used to save system state during interrupt critical sections.

Speed

Fast

Minimum Protection Level

INTCTRL_0

System Save Register Level 0 Entry 1 (SYSTEM_SAVE_0_1)

This register is used to save system state during interrupt critical sections.

Speed

Fast

Minimum Protection Level

INTCTRL_0

System Save Register Level 0 Entry 2 (SYSTEM_SAVE_0_2)

This register is used to save system state during interrupt critical sections.

Speed

Fast

Minimum Protection Level

INTCTRL_0

System Save Register Level 0 Entry 3 (SYSTEM_SAVE_0_3)

This register is used to save system state during interrupt critical sections.

Speed

Fast

Minimum Protection Level

INTCTRL_0

Minimum Protection Level for Tile Timer (MPL_TILE_TIMER)

This register specifies the minimum protection level needed to access the tile timer administratively. This register also serves as the protection level that handles tile timer interrupts.

Speed

Slow

Minimum Protection Level

TILE_TIMER

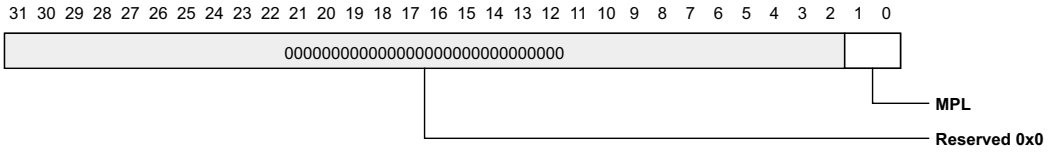

Figure 2. MPL_TILE_TIMER Register Diagram

Table A-56. MPL_TILE_TIMER Register Bit Descriptions

Bits	Name	Reset	Description
31:2	Reserved		Reserved
1:0	MPL	0	Minimum Protection Level

DMA Byte (DMA_BYTE) Register

This register specifies the number of chunks that a DMA operation will transfer, as well as the number of bytes that will be transferred in the first chunk.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

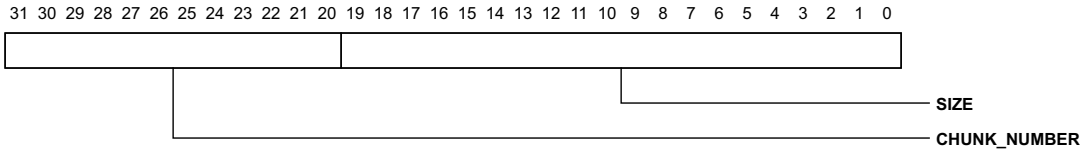

Figure A-1: DMA_BYTE Register Diagram

Table A-57. DMA_BYTE Register Bit Descriptions

Bits	Name	Default	Description
31:20	CHUNK_NUMBER	0	Number of chunks to be transferred. The first chunk will contain the number of bytes specified in the SIZE field of this register; the remaining chunks, if this field is greater than 1, will contain the number of bytes specified by the CHUNK_SIZE register.
19:0	SIZE	0	Number of bytes to be transferred in the first chunk. For multi-chunk transfers, this should be less than or equal to the value in the CHUNK_SIZE register.

DMA Chunk Size (DMA_CHUNK_SIZE) Register

This register specifies the DMA chunk size in bytes. It need not be set for an operation if only one chunk is to be transferred.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

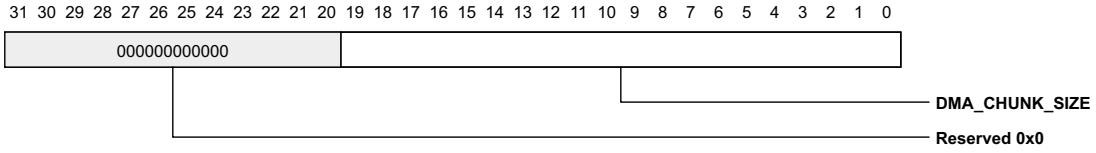

Figure A-2: DMA_CHUNK_SIZE Register Diagram

Table A-58. DMA_CHUNK_SIZE Register Bit Descriptions

Bits	Name	Default	Description
31:20	Reserved	0	
19:0	DMA_CHUNK_SIZE	0	This register specifies the DMA chunk size in bytes. It need not be set for an operation if only one chunk is to be transferred.

DMA Control (DMA_CTR) Register

This register controls the DMA engine. To perform a DMA request, the DMA transfer description registers (DMA_BYTE, DMA_CHUNK_SIZE, DMA_DST_ADDR, DMA_DST_CHUNK_ADDR, DMA_SRC_ADDR, DMA_SRC_CHUNK_ADDR, and DMA_STRIDE) are set appropriately, and then the REQUEST bit in this register is set. To context-switch the DMA engine, the SUSPEND bit in this register is set; then, once the BUSY bit in the DMA_USER_STATUS register has cleared, the transfer description registers are read and their contents saved. At a later time, those values may be re-loaded into the corresponding registers and the DMA engine restarted by writing the REQUEST bit; the transfer will then continue from when it was suspended.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

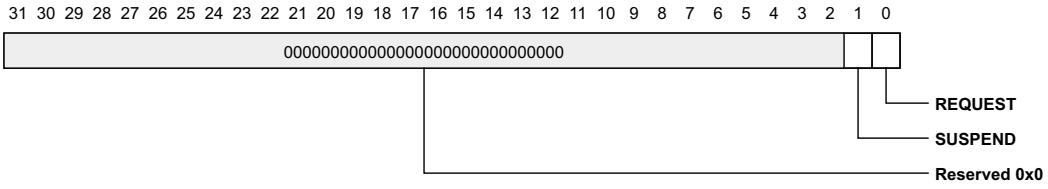

Figure A-3: DMA_CTR Register Diagram

Table A-59. DMA_CTR Register Bit Descriptions

Bits	Name	Reset	Description
31:2	Reserved	0	
1	SUSPEND	0	1 When set to 1, suspends the currently active DMA operation; this has no effect if no DMA operation is currently in progress. The DMA operation has not been suspended until the BUSY bit in the STATUS register has cleared.
0	REQUEST	0	1 When set to 1, starts a new DMA operation; this has no effect if a DMA operation is currently in progress.

DMA Destination Address (DMA_DST_ADDR) Register

This register holds the address of the first byte to be written when the next DMA operation is started; this will normally be identical to the DST_CHUNK_ADDR register unless the DMA engine is being restarted after partially transferring a chunk.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

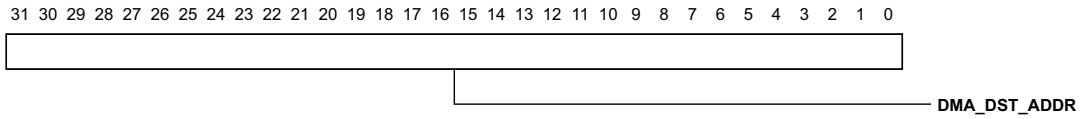

Figure A-4: DMA_DST_ADDR Register Diagram

Table A-60. DMA_DST_ADDR Register Bit Descriptions

Bits	Name	Default	Description
31:0	DMA_DST_ADDR	0	Address

DMA Destination Chunk Address (DMA_DST_CHUNK_ADDR) Register

This register holds the address of the first byte in the first destination chunk for the next DMA operation. This may not be the first byte to be written, depending on the contents of the DST_ADDR register.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

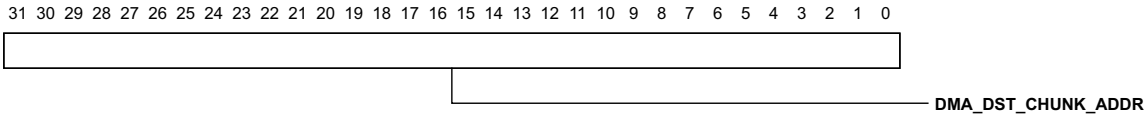

Figure A-5: DMA_DST_CHUNK_ADDR Register Diagram

Table A-61. DMA_DST_CHUNK_ADDR Register Bit Descriptions

Bits	Name	Reset	Description
31:0	DMA_DST_CHUNK_ADDR	0	Address of the first byte in the first destination chunk for the next DMA operation.

DMA Source Address (DMA_SRC_ADDR) Register

This register holds the address of the first byte to be read when the next DMA operation is started; this will normally be identical to the SRC_CHUNK_ADDR register unless the DMA engine is being restarted after partially transferring a chunk.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

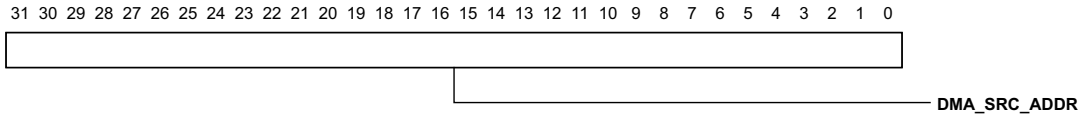

Figure A-6: DMA_SRC_ADDR Register Diagram

Table A-62. DMA_SRC_ADDR Register Bit Descriptions

Bits	Name	Reset	Description
31:0	DMA_SRC_ADDR	0	Address of the first byte to be read when the next DMA operation is started.

DMA Source Chunk Address (DMA_SRC_CHUNK_ADDR) Register

This register holds the address of the first byte in the first source chunk for the next DMA operation. This may not be the first byte to be read, depending on the contents of the SRC_ADDR register.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

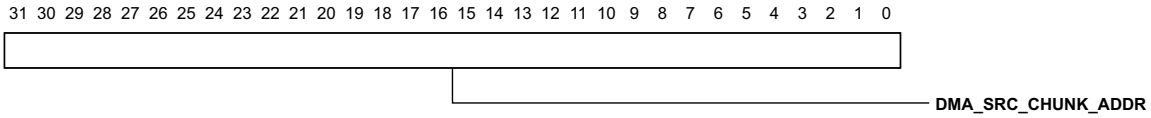

Figure A-7: DMA_SRC_CHUNK_ADDR Register Diagram

Table A-63. DMA_SRC_CHUNK_ADDR Register Bit Descriptions

Bits	Name	Reset	Description
31:0	DMA_SRC_CHUNK_ADDR	0	Address of the first byte in the first source chunk for the next DMA operation.

DMA Source And Destination Strides (DMA_STRIDE) Register

This register specifies the DMA source and destination strides. A stride is the distance between the first byte of successive chunks within one DMA operation; if only one chunk is transferred, the stride is irrelevant.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

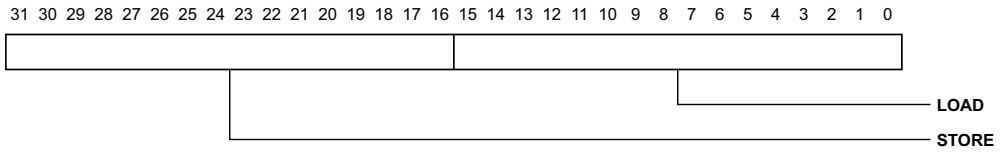

Figure A-8: DMA_STRIDE Register Diagram

Table A-64. DMA_STRIDE Register Bit Descriptions

Bits	Name	Default	Description
31:16	STORE	0	Store (destination) stride in bytes.
15:0	LOAD	0	Load (source) stride in bytes.

DMA User Status (DMA_USER_STATUS) Register

This register can be accessed by programs running at the DMA_NOTIFY PL; this is expected to be lower than the DMATLB_MISS PL.

Speed

Fast

Minimum Protection Level

DMA_NOTIFY

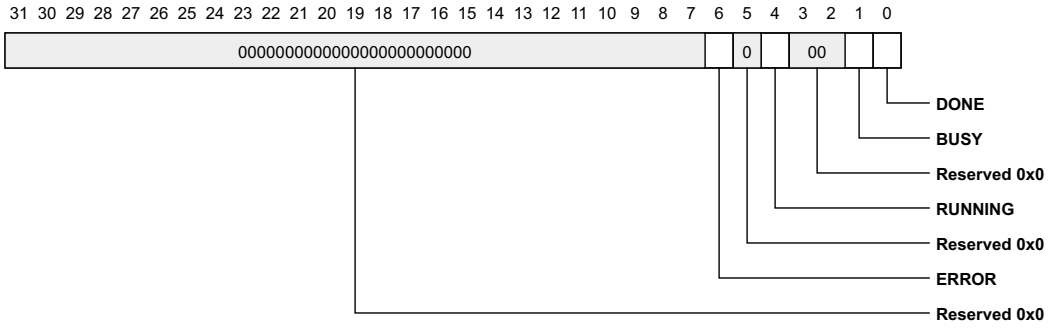

Figure A-9: DMA_USER_STATUS Register Diagram

Table A-65. DMA_USER_STATUS Register Bit Descriptions

Bits	Name	Reset	Description
31:7	Reserved	0	Reserved
6	ERROR	0	Status only. 1 This bit is set when the DMA engine encounters an internal error. This bit is cleared when a write to DMA_CTR starts a new transfer.
5	Reserved		Reserved
4	RUNNING	0	Status only. 1 If this bit is set, the last transfer started on the DMA engine has not been suspended via the SUSPEND bit in DMA_CTR. This bit is set when a write to DMA_CTR starts a new transfer; it is cleared when a write to DMA_CTR suspends an active transfer; it is not cleared in the event of a TLB miss, access violation, error, or normal DMA completion. This bit is used to determine whether the DMA engine should be restarted when exiting the DMATLB miss handler; it is suggested that the engine only be restarted if this bit is set.
3:2	Reserved	0	

Table A-65. DMA_USER_STATUS Register Bit Descriptions (continued)

Bits	Name	Reset	Description
1	BUSY	0	Busy bit. Status only. 1 If this bit is set, the DMA engine is active, and the contents of the DMA transfer description registers are undefined. If this bit is clear, and the engine has been paused due to the SUSPEND bit being set in the DM_ CTR register, or due to a TLB miss or access violation, then the DONE bit will be clear, and the DMA transfer description registers may be inspected to determine the state of the engine at the time of the suspension. 0 If this bit is 0, and the engine completed the last DMA request, the DONE bit will be set, and the content of the DMA transfer description registers are undefined.
0	DONE	0	Done bit 1 This bit is set when a DMA transfer completes. It is cleared when a write to DMA CTR starts a new transfer; it may also be cleared by writing a 1 to it whenever the BUSY bit is 0. While this bit is set, the DMA_NOTIFY interrupt is asserted.

G GLOSSARY

Term	Definition
CPLD	Complex PLD. A programmable logic device (PLD) that is made up of several simple PLDs (SPLDs) with a programmable switching matrix in between the logic blocks. CPLDs typically use EEPROM, flash memory or SRAM to hold the logic design interconnections.
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. This is in contrast to the static network, which has a statically specified route at each switch point, and every data follows an identical route.
ECC	Error-Correcting Code. A type of memory that corrects errors on the fly.
host port interfaces (HPIs)	A 16-bit-wide parallel port through which a host processor can directly access the CPU's memory space. The host device functions as a master to the interface, which increases ease of access. The host and CPU can exchange information via internal or external memory. The host also has direct access to memory-mapped peripherals. Connectivity to the CPU's memory space is provided through the DMA controller.
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).
MPI	Message Passing Interface. MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.
MPL	Minimum Protection Level. Each interrupt has a minimum protection level at which it may be processed. Interrupts which are signalled by a protection level less than the MPL are processed at the MPL protection level. Interrupts which are signalled at a protection level higher than the MPL are processed at the higher protection level. The MPL for a given interrupt is typically determined by system software.
Multicore Development Environment™ (MDE™)	Multicore programming environment.

Term	Definition
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 Processor™, SIMD instructions allow a single instruction to operate on registers containing four bytes or two halfwords.
SPI-SROM	Serial Flash with serial peripheral interface.
Static Network	A network where the routing for a given input port is specified statically. Each data on an input port will be sent to the same output port. This is in contrast to a dynamic network, where each message on an input port may be routed to a different output port.
UART	(Universal Asynchronous Receiver Transmitter). The electronic circuit that makes up the serial port. Also known as “universal serial asynchronous receiver transmitter” (USART), it converts parallel bytes from the CPU into serial bits for transmission, and vice versa. It generates and strips the start and stop bits appended to each character.
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.
WAR Dependence	Write-after-Read dependence, or anti-dependence. WAR dependencies arise when a write operation on a location follows in program order a read operation to the same location. The read operation must not receive the value from the following write operation, so the write operation must wait for all previous read operations to complete if the processor executes the operations simultaneously or out of order.
WAW Dependence	Write-after-Write dependence or output dependence. WAW dependencies arise when a write operation on a location follows in program order a write operation to the same location. The final write operation must be the value in the location after both operations are completed, so the second write operation must wait for all previous write operations to complete, or the earlier write operations must be ignored if the processor executes the operations simultaneously or out of order.
wormhole routing	A network where the routing is determined by the header of a packet, and where once the header of a packet has traversed a switch point, the routing will not be changed until the last packet word has traversed the switch.

I INDEX

N