SiFive U74-MC Core Complex Manual

21G3.02.00

SiFive U74-MC Core Complex Manual

Proprietary Notice

Contents

Tables

Figures

Chapter 1

Introduction

SiFive's U74-MC Core Complex is a full-Linux-capable, cache-coherent 64-bit RISC-V processor available as an IP block. The SiFive U74-MC Core Complex is guaranteed to be compatible with all applicable RISC-V standards, and this document should be read together with the official RISC-V user-level, privileged, and external debug architecture specifications.

A summary of features in the U74-MC Core Complex can be found in Table 1.

U74-MC Core Complex Feature Set	
Feature	**Description**
Number of Harts	5 Harts.
S7 Core	1 × S7 RISC-V core.
U7 Core	4 × U7 RISC-V cores.
PLIC Interrupts	127 Interrupt signals, which can be connected to off-core-complex devices.
PLIC Priority Levels	The PLIC supports 7 priority levels.
Level 2 Cache	2 MiB 16-way L2 Cache.
Hardware Breakpoints	2 hardware breakpoints.
Physical Memory Protection Unit	PMP with 8 regions and a minimum granularity of 4096 bytes.

Table 1: U74-MC Core Complex Feature Set

The U74-MC Core Complex also has a number of on-core-complex configurability options, allowing one to tune the design to a specific application. The configurable options are described in Appendix A.

1.1 About this Document

This document describes the functionality of the U74-MC Core Complex 21G3.02.00. To learn more about the production deliverables of the U74-MC Core Complex, consult the U74-MC Core Complex User Guide.

1.2 About this Release

This is a general release of the U74-MC Core Complex 21G3.02.00, with a supported life cycle of two years from the release date. Contact support@sifive.com if you have any questions.

1.3 U74-MC Core Complex Overview

The U74-MC Core Complex includes 1 × S7 and 4 × U7 64-bit RISC-V cores, along with the necessary functional units required to support the cores. These units include a Core-Local Interruptor (CLINT) to support local interrupts, a Platform-Level Interrupt Controller (PLIC) to support platform interrupts, physical memory protection, a Debug unit to support a JTAG-based debugger host connection, and a local crossbar that integrates the various components together.

The U74-MC Core Complex memory system consists of a Data Cache, Data Tightly-Integrated Memory (DTIM), Instruction Cache, and Instruction Tightly-Integrated Memory (ITIM), with coherent L1 caches, shared L2 Cache, and a directory based coherence manager. The U74-MC Core Complex also includes a Front Port, which allows external masters to be coherent with the L1 memory system and access to the TIMs, thereby removing the need to maintain coherence in software for any external agents.

All memories, including caches and TIMs, support Single Error Correction, Double Error Detection (SECDED) ECC to provide improved reliability and address safety critical applications.

An overview of the SiFive U7-MC Series is shown in Figure 1. Refer to the `docs/core_complex_configuration.txt` file for a comprehensive summary of the U74-MC Core Complex configuration.

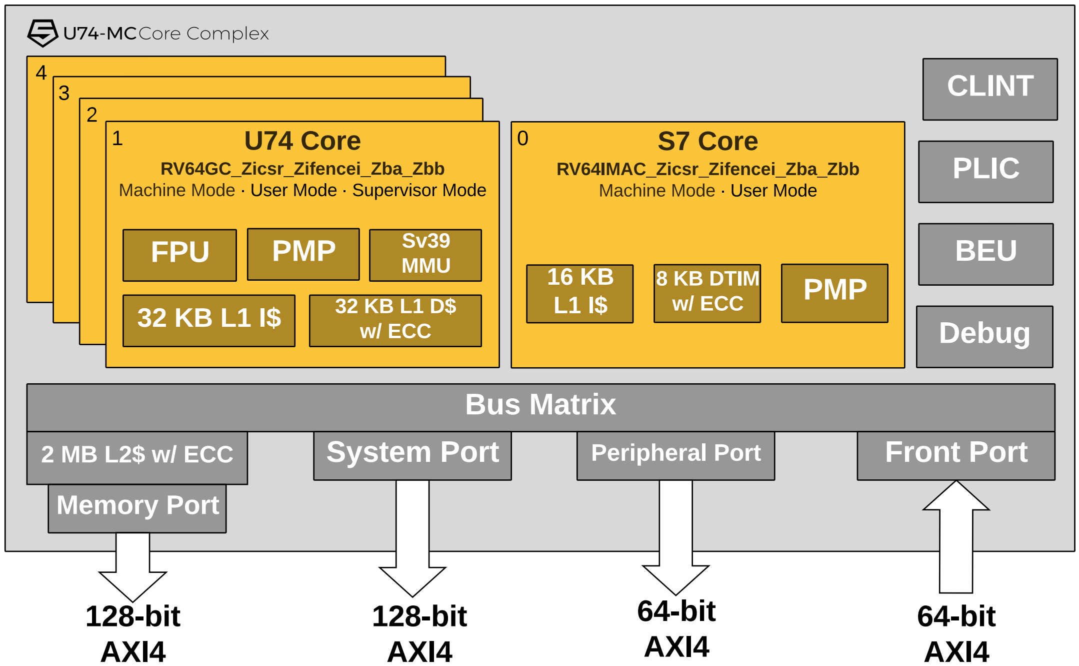

Figure 1: U7-MC Series Block Diagram

The U74-MC Core Complex memory map is detailed in Section 5.2, and the interfaces are described in full in the U74-MC Core Complex User Guide.

1.4 S7 RISC-V Monitor Core

The U74-MC Core Complex includes a 64-bit S7 RISC-V core, which has a dual-issue, in-order execution pipeline, with a peak execution rate of two instructions per clock cycle. The SiFive S7 core is guaranteed to be compatible with all applicable RISC-V standards.

The S7 core is configured to support the RV64I base ISA, as well as the Multiply (M), Atomic (A), Compressed (C), CSR Instructions (Zicsr), Instruction-Fetch Fence (Zifencei), Address Calculation (Zba), Basic Bit Manipulation (Zbb), and Count Overflow and Mode-Based Filtering (Sscofpmf) RISC-V extensions. This is captured by the RISC-V extension string: *RV64IMAC_Zicsr_Zifencei_Zba_Zbb_Sscofpmf*. The base ISA and instruction extensions are described in Chapter 6.

The S7 also supports machine and user privilege modes, in conjunction with Physical Memory Protection (PMP), thereby allowing System-on-Chip (SoC) implementations to make the right area, power, and feature trade-offs.

The S7 core is designed to be feature rich, providing a very flexible memory system that includes an L1 cache, Tightly-Integrated Memory (TIM), standards-based configurable bus interfaces, and memory maps that provide a lot of flexibility for SoC integration.

The microarchitecture also incorporates a branch prediction unit that is composed of a 16-entry Branch Target Buffer (BTB), a 3.6 KiB-entry Branch History Table (BHT), a 6-entry Return Address Stack (RAS), 8-entry Indirect Jump Target Predictor (IJTP), and a 16-entry Return Instruction Predictor.

The S7 monitor core is described in more detail in Chapter 3.

1.5 U7 RISC-V Application Cores

The U74-MC Core Complex includes four 64-bit U7 RISC-V cores, which each have a dual-issue, in-order execution pipeline, with a peak execution rate of two instructions per clock cycle. The SiFive U7 core is guaranteed to be compatible with all applicable RISC-V standards.

Each U7 core is configured to support the RV64I base ISA, as well as the Multiply (M), Atomic (A), Single-Precision Floating Point (F), Double-Precision Floating Point (D), Compressed (C), CSR Instructions (Zicsr), Instruction-Fetch Fence (Zifencei), Address Calculation (Zba), Basic Bit Manipulation (Zbb), and Count Overflow and Mode-Based Filtering (Sscofpmf) RISC-V extensions. This is captured by the RISC-V extension string: *RV64GC_Zba_Zbb_Sscofpmf*. The base ISA and instruction extensions are described in Chapter 6.

The U7 also supports machine, supervisor, and user privilege modes, in conjunction with Physical Memory Protection (PMP), thereby allowing System-on-Chip (SoC) implementations to make the right area, power, and feature trade-offs.

The U7 core is designed to be feature rich, providing a very flexible memory system that includes L1 caches, Tightly-Integrated Memory (TIM), standards-based configurable bus interfaces, and memory maps that provide a lot of flexibility for SoC integration.

The microarchitecture also incorporates a branch prediction unit that is composed of a 16-entry Branch Target Buffer (BTB), a 3.6 KiB-entry Branch History Table (BHT), a 6-entry Return Address Stack (RAS), 8-entry Indirect Jump Target Predictor (IJTP), and a 16-entry Return Instruction Predictor.

The U7 includes an IEEE 754-2008 compliant Floating-Point Unit.

The U7 application cores are described in more detail in Chapter 4.

1.6 Interrupts

The U74-MC Core Complex provides the standard RISC-V M-mode timer and software interrupts via the Core-Local Interruptor (CLINT).

The U74-MC Core Complex also includes a RISC-V standard Platform-Level Interrupt Controller (PLIC), which supports 136 global interrupts with 7 priority levels pre-integrated with the on-core-complex peripherals.

Interrupts are described in Chapter 8. The CLINT is described in Chapter 9. The PLIC is described in Chapter 10.

1.7 Debug Support

The U74-MC Core Complex provides external debugger support over an industry-standard JTAG port, including 2 hardware-programmable breakpoints per hart.

Debug support is described in detail in Chapter 16, and the debug interface is described in the U74-MC Core Complex User Guide.

1.8 Compliance

The U74-MC Core Complex is compliant to the following versions of the various RISC-V specifications:

ISA	Version	Status
RV64I Base Integer Instruction Set	2.1	Ratified
Extensions	**Version**	**Status**
M Standard Extension for Integer Multiplication and Division	2.0	Ratified
A Standard Extension for Atomic Instruction	2.1	Ratified
F Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D Standard Extension for Double-Precision Floating-Point	2.2	Ratified
C Standard Extension for Compressed Instruction	2.0	Ratified
Zicsr Standard Extension for Control and Status Register (CSR) Instructions	2.0	Ratified
Zifencei Standard Extension for Instruction-Fetch Fence	2.0	Ratified
Zba Standard Extension for Address Calculation	1.0	Ratified
Zbb Standard Extension for Basic Bit Manipulation	1.0	Ratified
Sscofpmf Standard Extension for Count Overflow and Mode-Based Filtering	0.1	Ratified
Privilege Mode	**Version**	**Status**
Machine-Level ISA	1.11	Ratified
User-Level ISA	1.11	Ratified
Supervisor-Level ISA	1.11	Ratified
Devices	**Version**	**Status**
The RISC-V Debug Specification	1.0	Frozen
RISC-V Platform-Level Interrupt Controller (PLIC) Specification	—	—

Table 2: RISC-V Specification Compliance

Chapter 2

List of Abbreviations and Terms

Term	Definition
BHT	Branch History Table
BTB	Branch Target Buffer
CLIC	Core-Local Interrupt Controller. Configures priorities and levels for core-local interrupts.
CLINT	Core-Local Interruptor. Generates per hart software and timer interrupts.
DTIM	Data Tightly-Integrated Memory
Hart	HARdware Thread
IJTP	Indirect-Jump Target Predictor
ITIM	Instruction Tightly-Integrated Memory
JTAG	Joint Test Action Group
LIM	Loosely-Integrated Memory. Used to describe memory space delivered in a SiFive Core Complex that is not tightly integrated to a CPU core.
PLIC	Platform-Level Interrupt Controller. The global interrupt controller in a RISC-V system.
PMC	Power Management Controller
PMP	Physical Memory Protection
RAS	Return-Address Stack
RO	Used to describe a Read-Only register field
RS	Read/Set field. A register field that cannot be cleared by software, only reset will clear.
RW	Used to describe a Read/Write register field
RW1C	Used to describe a Read/Write-1-to-Clear register field
TileLink	A free and open interconnect standard originally developed at UC Berkeley
W1C	Used to describe a Write-1-to-Clear register field
WARL	Write-Any, Read-Legal field. A register field that can be written with any value, but returns only supported values when read.
WIRI	Writes-Ignored, Reads-Ignore field. A read-only register field reserved for future use. Writes to the field are ignored and reads should ignore the value returned.
WLRL	Write-Legal, Read-Legal field. A register field that should only be written with legal values and that only returns legal value if last written with a legal value.
WO	Used to describe a Write-Only register field
WPRI	Writes-Preserve, Reads-Ignore field. A register field that might contain unknown information. Reads should ignore the value returned, but writes to the whole register should preserve the original value.

Table 3: Abbreviations and Terms

Chapter 3

S7 RISC-V Core

This chapter describes the 64-bit S7 RISC-V processor core, instruction fetch and execution unit, L1 and L2 memory systems, Physical Memory Protection unit, Hardware Performance Monitor, and external interfaces.

The S7 feature set is summarized in Table 4.

Feature	**Description**
ISA	*RV64IMAC_Zicsr_Zifencei_Zba_Zbb_Sscofpmf*
SiFive Custom Instruction Extension (SCIE)	Not Present
Privilege Modes	M, U
L1 Instruction Cache	16 KiB 2-way instruction cache
Instruction Tightly-Integrated Memory (ITIM)	16 KiB ITIM
Data Tightly-Integrated Memory (DTIM)	8 KiB DTIM
L2 Cache	2 MiB 16-way L2 cache with 2 banks
ECC Support	Single error correction, double error detection on the ITIM, DTIM, and L2 cache
Fast I/O	Present
Physical Memory Protection	8 regions with a granularity of 64 bytes

Table 4: S7 Feature Set

3.1 Supported Privilege Modes

The S7 supports the RISC-V user mode, providing two levels of privilege: machine (M) and user (U). U-mode provides a mechanism to isolate application processes from each other and from trusted code running in M-mode.

See *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* for more information on the privilege modes.

3.2 Instruction Memory System

This section describes the instruction memory system of the S7 Monitor core.

3.2.1 Execution Memory Space

The regions of executable memory consist of all directly addressable memory in the system. The memory includes any volatile or non-volatile memory located off the Core Complex ports, and includes the on-core-complex DTIM, ITIM, L2 LIM, and L2 Zero Device.

Table 5 shows the executable regions of the U74-MC Core Complex.

Base	Top	Description
`0x0100_0000`	`0x0100_1FFF`	S7 Hart 0 DTIM (8 KiB)
`0x0180_0000`	`0x0180_3FFF`	S7 Hart 0 ITIM
`0x0800_0000`	`0x081F_FFFF`	L2 LIM
`0x0A00_0000`	`0x0A1F_FFFF`	L2 Zero Device
`0x2000_0000`	`0x3FFF_FFFF`	Peripheral Port (512 MiB)
`0x4000_0000`	`0x5FFF_FFFF`	System Port (512 MiB)
`0x8000_0000`	`0x10_7FFF_FFFF`	Memory Port (64 GiB)

Table 5: Executable Memory Regions for the U74-MC Core Complex

All executable regions, except the ITIM, are treated as instruction cacheable. There is no method to disable this behavior.

Trying to execute an instruction from a non-executable address results in an instruction access trap.

3.2.2 L1 Instruction Cache

The L1 instruction cache is a 16 KiB 2-way set-associative cache. It has a line size of 64 bytes and is read/write-allocate with a random replacement policy. A cache line fill triggers a burst access outside of the Core Complex, starting with the first address of the cache line. There are no write-backs to memory from the instruction cache and it is not kept coherent with rest of the platform memory system. In multi-core systems, the instruction caches are not kept coherent with each other.

Out of reset, all blocks of the instruction cache are invalidated. The access latency of the cache is one clock cycle. There is no way to disable the instruction cache and cache allocations begin immediately out of reset.

The L1 instruction cache has parity error protection support.

3.2.3 Cache Maintenance

The instruction cache supports the `FENCE.I` instruction, which invalidates the entire instruction cache, as described in Section 6.14. Writes to instruction memory from the core or another master must be synchronized with the instruction fetch stream by executing `FENCE.I`.

3.2.4 Coherence with Higher Level Caches

The L1 instruction cache is partially inclusive with the L2 Cache, described in Chapter 14.

When a block of instruction memory is allocated to the L1 cache, it is also allocated to the L2 cache if the access was from the Memory Port. Instruction accesses to all other ports will not allocate to the L2 cache, only the L1 cache.

When a block is evicted from L1, it might still reside in L2, which will reduce access time the next time the block is fetched.

If a hart modifies instruction memory (i.e., self-modifying code), then a `FENCE.I` instruction is required to synchronize the instruction and data streams. Even though `FENCE.I` targets the L1 instruction cache, no cache operation is required on the L2 cache to maintain instruction coherency.

3.2.5 Instruction Tightly-Integrated Memory (ITIM)

The S7 includes a 16 KiB ITIM in addition to the L1 instruction cache. ITIM accesses have the same performance as instruction cache hits, but can never suffer a miss. This makes the ITIM useful for storing code that benefits from deterministic execution, such as interrupt handlers.

The ITIM supports ECC protection, as described in Chapter 17.

3.2.6 Instruction Fetch Unit

The S7 instruction fetch unit is responsible for keeping the pipeline fed with instructions from memory. The instruction fetch unit delivers up to 8 bytes of instructions per clock cycle to support superscalar instruction execution. Fetches are always word-aligned and there is a one-cycle penalty for branching to a 32-bit instruction that is not word-aligned.

The S7 implements the standard Compressed (C) extension to the RISC-V architecture, which allows for 16-bit RISC-V instructions. As four 16-bit instructions can be fetched per cycle, the instruction fetch unit can be idle when executing programs comprised mostly of compressed 16-bit instructions. This reduces memory accesses and power consumption.

All branches must be aligned to half-word addresses. Otherwise, the fetch generates an instruction address misaligned trap. Trying to fetch from a non-executable or unimplemented address results in an instruction access trap.

3.2.7 Branch Prediction

The S7 instruction fetch unit contains sophisticated predictive hardware to mitigate the performance impact of control hazards within the instruction stream. The instruction fetch unit is decoupled from the execution unit, so that correctly predicted control-flow events usually do not result in execution stalls.

- A 16-entry branch target buffer (BTB), which predicts the target of taken branches and direct jumps;
- A 3.6 KiB branch history table (BHT), which predicts the direction of conditional branches;
- An 8-entry indirect-jump target predictor (IJTP);
- A 6-entry return-address stack (RAS), which predicts the target of procedure returns.

The BHT is a correlating predictor that supports long branch histories. The BTB has one-cycle latency, so that correctly predicted branches and direct jumps result in no penalty, provided the target is 8-byte aligned.

Direct jumps that miss in the BTB result in a one-cycle fetch bubble. This event might not result in any execution stalls if the fetch queue is sufficiently full.

The BHT, IJTP, and RAS take precedence over the BTB. If these structures' predictions disagree with the BTB's prediction, a one-cycle fetch bubble results. Similar to direct jumps that miss in the BTB, the fetch bubble might not result in an execution stall.

Mispredicted branches usually incur a four-cycle penalty, but sometimes the branch resolves later in the execution pipeline and incurs a six-cycle penalty instead. Mispredicted indirect jumps incur a six-cycle penalty.

Branch prediction is enabled out of reset and cannot be disabled. However, instruction speculation, fetching before a prediction is confirmed, must be enabled in the Feature Disable CSR, described in Chapter 7.

As instruction speculation can occur at any point after it has been enabled, data cacheable regions of memory (i.e., DDR) must be able to respond to instruction fetches immediately after instruction speculation is enabled. If DDR initialization is not completed before instruction speculation is enabled, the memory system must return a decode error (`DECERR`) for accesses made to DDR. The fetch unit will ignore errors associated with speculative accesses and continue to operate normally.

The Branch Prediction Mode CSR, also described in Chapter 7, provides a means to customize the branch predictor behavior to trade average performance for more predictable execution time.

3.3 Execution Pipeline

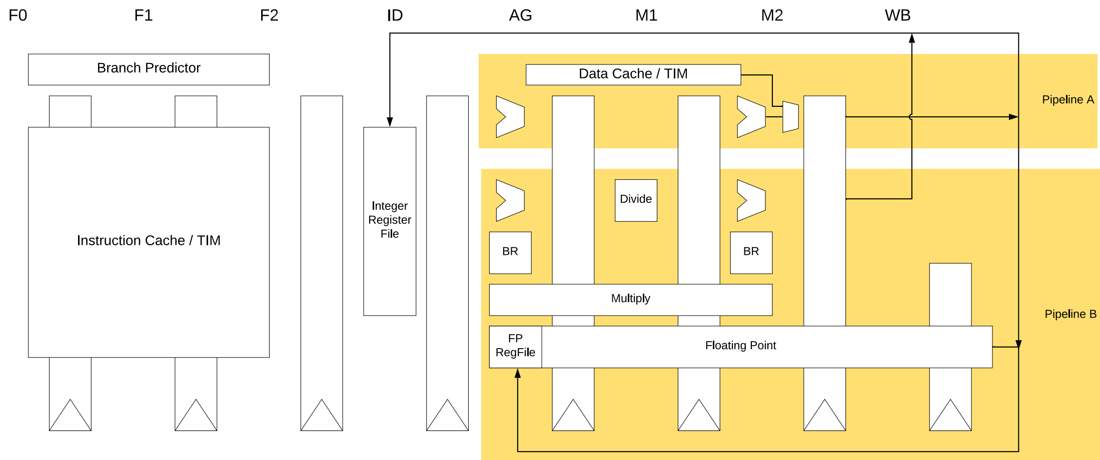

Figure 2: S7 Execution Pipeline

The S7 execution unit is a dual-issue, in-order pipeline. The pipeline comprises eight stages: three stages of instruction fetch (F0-F2), instruction decode (ID), address generation (AG), two stages of data memory access (M1 and M2), and register write-back (WB). The pipeline has a peak execution rate of two instructions per clock cycle, and is fully bypassed so that most instructions have a one-cycle result latency:

- Integer arithmetic and branch instructions can execute in either the AG or M2 pipeline stage. If such an instruction's operands are available when the instruction enters the AG stage, then it executes in AG; otherwise, it executes in M2.
- Loads produce their result in the M2 stage. There is no load-use delay for most integer instructions. However, effective addresses for memory accesses are always computed in the AG stage. Hence, loads, stores, and indirect jumps require their address operands to be ready when the instruction enters AG. If an address-generation operation depends upon a load from memory, then the load-use delay is two cycles.
- Integer multiplication instructions consume their operands in the AG stage and produce their results in the M2 stage. The integer multiplier is fully pipelined.
- Integer division instructions consume their operands in the AG stage. These instructions have between a six-cycle and 68-cycle result latency, depending on the operand values.
- CSR accesses execute in the M2 stage. CSR read data can be bypassed to most integer instructions with no delay. Most CSR writes flush the pipeline, which is a seven-cycle penalty.

Instruction	Latency
LW	Three-cycle latency, assuming cache hit[1]
LH, LHU, LB, LBU	Three-cycle latency, assuming cache hit[1]
CSR Reads	One-cycle latency[2]
MUL, MULH, MULHU, MULHSU	Three-cycle latency
DIV, DIVU, REM, REMU	Between six-cycle to 68-cycle latency, depending on operand values[3]
[1]Effective address not ready in AG stage. Load-to-use latency = load-to-use delay + 1 [2] cycle latency = cycle delay + 1 [3]The latency of DIV, DIVU, REM, and REMU instructions can be determined by calculating: `Latency = 2 cycles + log2(dividend) - log2(divisor) + 1 cycle` if the input is negative + 1 cycle if the output is negative	

Table 6: S7 Instruction Latency

The pipeline has some register dependencies, where it interlocks on read-after-write and write-after-write hazards, so instructions may be scheduled to avoid stalls. Otherwise, the processor can have multiple outstanding memory-mapped I/O accesses, even to the same address. The pipeline implements a flexible dual-instruction-issue scheme. Provided there are no data hazards between a pair of instructions, the two instructions may issue in the same cycle, provided the following constraints are met:

- At most one instruction accesses data memory.
- At most one instruction is a branch or jump.
- At most one instruction is an integer multiplication or division operation.
- Neither instruction explicitly accesses a CSR.

3.4 Data Memory System

The data memory system consists of on-core-complex data and the ports in the U74-MC Core Complex memory map, shown in Section 5.2. The on-core-complex data memory consists of an 8 KiB Data Tightly-Integrated Memory (DTIM) and 2 MiB L2 cache. A design cannot have both data cache and DTIM.

As no data cache is present, all data accesses are non-cacheable. Data accesses that are not targeted at the DTIM are also called memory-mapped I/O accesses, or MMIOs.

The S7 pipeline allows for multiple outstanding memory accesses. The memory system includes the Fast I/O feature, described in Section 3.5, which improves the throughput of MMIOs. The number of outstanding MMIOs are implementation dependent. Misaligned accesses are not allowed to any memory region and result in a trap to allow for software emulation.

3.4.1 Data Tightly-Integrated Memory (DTIM)

The DTIM provides deterministic access time, which is important for applications with hard real-time requirements. The access latency is two clock cycles for words and double-words, and three clock cycles for smaller quantities.

Stores are pipelined and commit on cycles where the data memory system is otherwise idle. Loads to addresses currently in the store pipeline result in a five-cycle penalty.

The DTIM region can be used to store instructions, but it has no lasting performance advantage over other memory regions. Fetching from the DTIM first results in an instruction cache line fill and execution occurs from the instruction cache.

The DTIM supports the RISC-V standard Atomic (A) extension, with the exception of the LR/SC instructions that are only supported in data cacheable regions. See Chapter 6 for more information on the instructions added by this extension.

The DTIM supports ECC protection, as described in Chapter 17.

3.5 Fast I/O

The Fast I/O feature improves the performance of the memory-mapped I/O (MMIO) subsystem. This is achieved by predicting whether an access is I/O or not by examining the base address of a read or write.

Fast I/O enables a sustained rate of one MMIO operation per clock cycle. By contrast, when this feature is excluded, MMIO loads can only sustain half that rate. Fast I/O also decouples the MMIO load response from the cache-hit path. This way, MMIO requests and responses can happen on the same cycle, doubling the peak load throughput.

> **Note**
>
> Fast I/O is NOT an I/O port.

3.6 Atomic Memory Operations

The S7 core supports the RISC-V standard Atomic (A) extension on the Peripheral Port and internal memory regions.

The load-reserved (LR) and store-conditional (SC) instructions are special atomic instructions that are only supported in data cacheable regions. As the S7 core does not have a data cache, the LR and SC instructions will always generate a precise access exception.

Atomic memory operations are not supported on the System Port. Atomic operations that target the System Port will generate a precise access exception.

See Section 6.4 for more information on the instructions added by this extension.

3.7 Physical Memory Protection (PMP)

Machine mode is the highest privilege level and by default has read, write, and execute permissions across the entire memory map of the device. However, privilege levels below machine mode do not have read, write, or execute permissions to any region of the device memory map unless it is specifically allowed by the PMP. For the lower privilege levels, the PMP may grant permissions to specific regions of the device's memory map, but it can also revoke permissions when in machine mode.

When programmed accordingly, the PMP will check every access when the hart is operating in user mode. For machine mode, PMP checks do not occur unless the lock bit (`L`) is set in the `pmpcfgY` CSR for a particular region.

PMP checks also occur on loads and stores when the machine previous privilege level is user (`mstatus.MPP=0x0`), and the Modify Privilege bit is set (`mstatus.MPRV=1`).

The S7 PMP supports 8 regions with a minimum region size of 64 bytes.

This section describes how PMP concepts in the RISC-V architecture apply to the S7. For additional information on the PMP refer to *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

3.7.1 PMP Functional Description

The S7 PMP unit has 8 regions and a minimum granularity of 64 bytes. Access to each region is controlled by an 8-bit `pmpXcfg` field and a corresponding `pmpaddrX` register. Overlapping regions are permitted, where the lower numbered `pmpXcfg` and `pmpaddrX` registers take priority over higher numbered regions. The S7 PMP unit implements the architecturally defined `pmpcfgY` CSR `pmpcfg0`, supporting 8 regions. `pmpcfg2` is implemented, but hardwired to zero. Access to `pmpcfg1` or `pmpcfg3` results in an illegal instruction exception.

The PMP registers may only be programmed in M-mode. Ordinarily, the PMP unit enforces permissions on U-mode accesses. However, locked regions (see Section 3.7.2) additionally enforce their permissions on M-mode.

3.7.2 PMP Region Locking

The PMP allows for region locking whereby, once a region is locked, further writes to the configuration and address registers are ignored. Locked PMP entries may only be unlocked with a system reset. A region may be locked by setting the `L` bit in the `pmpXcfg` register.

In addition to locking the PMP entry, the `L` bit indicates whether the R/W/X permissions are enforced on machine mode accesses. When the `L` bit is clear, the R/W/X permissions apply only to U-mode.

3.7.3 PMP Registers

Each PMP region is described by an 8-bit `pmpXcfg` field, used in association with a 64-bit `pmpaddrX` register that holds the base address of the protected region. The range of each region depends on the Addressing (`A`) mode described in the next section. The `pmpXcfg` fields reside within 64-bit `pmpcfgY` CSRs.

Each 8-bit `pmpXcfg` field includes a read, write, and execute bit, plus a two bit address-matching field `A`, and a Lock bit, `L`. Overlapping regions are permitted, where the lowest numbered PMP entry wins for that region.

PMP Configuration Registers

For RV64 architectures, `pmpcfg1` and `pmpcfg3` are not implemented. This reduces the footprint since `pmpcfg2` already contains configuration fields `pmp8cfg` through `pmp11cfg` for both RV32 and RV64.

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
pmp7cfg	pmp6cfg	pmp5cfg	pmp4cfg	pmp3cfg	pmp2cfg	pmp1cfg	pmp0cfg

Figure 3: RV64 `pmpcfg0` Register

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
pmp15cfg	pmp14cfg	pmp13cfg	pmp12cfg	pmp11cfg	pmp10cfg	pmp9cfg	pmp8cfg

Figure 4: RV64 `pmpcfg2` Register

The `pmpcfgY` and `pmpaddrX` registers are only accessible via CSR specific instructions such as `csrr` for reads, and `csrw` for writes.

7	6 5	4 3	2	1	0
L (**WARL**)	0 (**WARL**)	A (**WARL**)	X (**WARL**)	W (**WARL**)	R (**WARL**)

Figure 5: RV64 `pmpXcfg` bitfield

<table>
<tr><th>Bits</th><th>Description</th></tr>
<tr><td>0</td><td>R: Read Permissions
0x0 - No read permissions for this region
0x1 - Read permission granted for this region</td></tr>
<tr><td>1</td><td>W: Write Permissions
0x0 - No write permissions for this region
0x1 - Write permission granted for this region</td></tr>
<tr><td>2</td><td>X: Execute permissions
0x0 - No execute permissions for this region
0x1 - Execute permission granted for this region</td></tr>
<tr><td>[4:3]</td><td>A: Address matching mode
0x0 - PMP Entry disabled. No PMP protection applied for any privilege level.
0x1 - Top of range (TOR) region defined by two adjacent <code>pmpaddr</code> registers. The upper limit of region X is defined by <code>pmpaddrX</code>, and the base of the region is defined by <code>pmpaddr(X-1)</code>. Address 'a' matches the region if [<code>pmpaddr(X-1)</code> ≤ a < <code>pmpaddrX</code>]. If <code>pmp0cfg</code> defines a TOR region, then the base address of that region is 0x0, and <code>pmpaddr0</code> defines the upper limit. Supports only a four byte granularity.
0x2 - Naturally aligned four-byte region (NA4). Supports only a four-byte region with four byte granularity.
0x3 - Naturally aligned power-of-two region (NAPOT), ≥ 8 bytes. When this setting is programmed, the low bits of the <code>pmpaddrX</code> register encode the size, while the upper bits encode the base address right shifted by two. There is a zero bit in between, we will refer to as the least significant zero bit (LSZB).</td></tr>
<tr><td>7</td><td>L: Lock Bit
0x0 - PMP Entry Unlocked, no permission restrictions applied to machine mode. PMP entry only applies to S and U modes.
0x1 - PMP Entry Locked, permissions enforced for all privilege levels including machine mode. Writes to <code>pmpXcfg</code> and <code>pmpcfgY</code> are ignored and can only be cleared with system reset.</td></tr>
<tr><td colspan="2">Note: The combination of <code>R</code>=0 and <code>W</code>=1 is not currently implemented.</td></tr>
</table>

Table 7: `pmpXcfg` Bitfield Description

Out of reset, the PMP register fields `A` and `L` are set to 0. All other hart state is unspecified by *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

Some examples follow using NAPOT address mode.

Base Address	Region Size*	LSZB Position	`pmpaddrX` Value
`0x4000_0000`	8 B	0	`(0x1000_0000 | 1’b0)`
`0x4000_0000`	32 B	2	`(0x1000_0000 | 3’b011)`
`0x4000_0000`	4 KB	9	`(0x1000_0000 | 10’b01_1111_1111)`
`0x4000_0000`	64 KB	13	`(0x1000_0000 | 14’b01_1111_1111_1111)`
`0x4000_0000`	1 MB	17	`(0x1000_0000 | 18’b01_1111_1111_1111_1111)`
*Region size is $2^{(LSZB+3)}$.			

Table 8: `pmpaddrX` Encoding Examples for A=NAPOT

PMP Address Registers

The PMP has 8 address registers. Each address register `pmpaddrX` correlates to the respective `pmpXcfg` field. Each address register contains the base address of the protected region right shifted by two, for a minimum 4-byte alignment.

The maximum encoded address bits per *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* are [55:2].

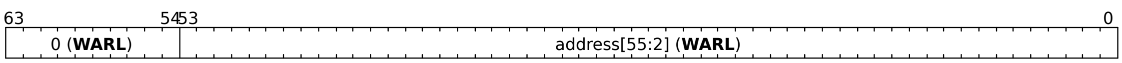

Figure 6: RV64 `pmpaddrX` Register

3.7.4 PMP and PMA

The PMP values are used in conjunction with the Physical Memory Attributes (PMAs) described in Section 5.1. Since the PMAs are static and not configurable, the PMP can only revoke read, write, or execute permissions to the PMA regions if those permissions already apply statically.

3.7.5 PMP Programming Overview

The PMP registers can only be programmed in machine mode. The `pmpaddrX` register should be first programmed with the base address of the protected region, right shifted by two. Then, the `pmpcfgY` register should be programmed with the properly configured 64-bit value containing each properly aligned 8-bit `pmpXcfg` field. Fields that are not used can be simply written to 0, marking them unused.

PMP Programming Example

The following example shows a machine mode only configuration where PMP permissions are applied to three regions of interest, and a fourth region covers the remaining memory map. Recall that lower numbered `pmpXcfg` and `pmpaddrX` registers take priority over higher numbered regions. This rule allows higher numbered PMP registers to have blanket coverage over the entire memory map while allowing lower numbered regions to apply permissions to specific regions of interest. The following example shows a 64 KB Flash region at base address `0x0`, a

32 KB RAM region at base address 0x2000_0000, and finally a 4 KB peripheral region at base address base 0x3000_0000. The rest of the memory map is reserved space.

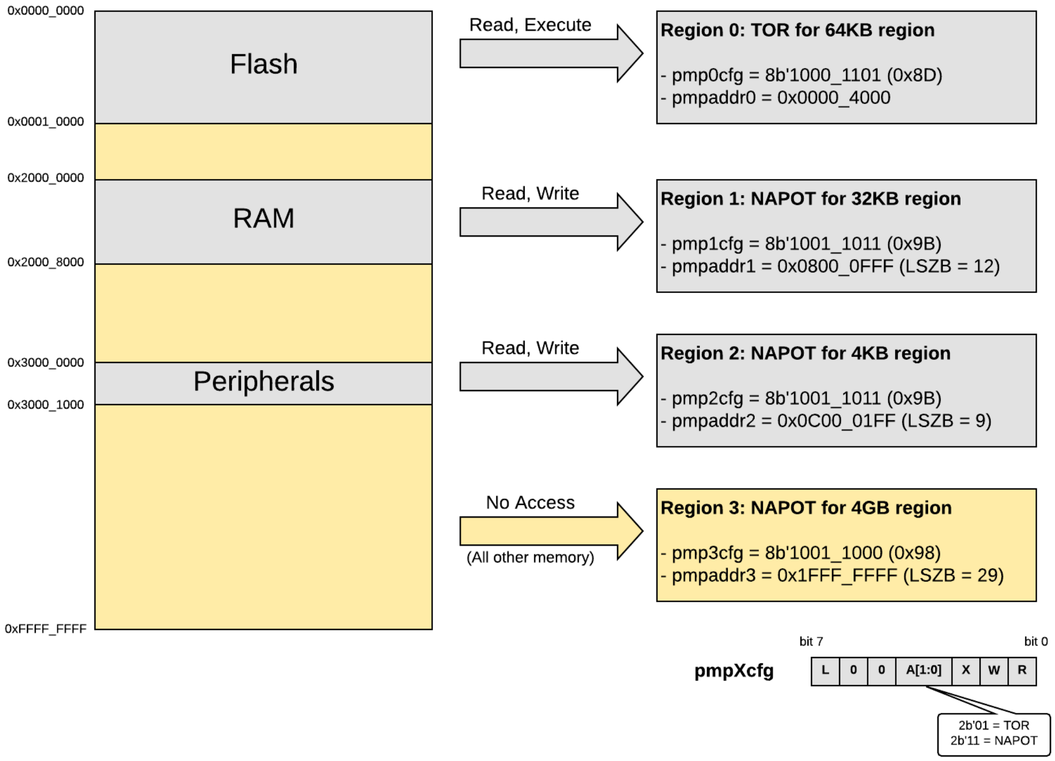

Figure 7: PMP Example Block Diagram

PMP Access Scenarios

The L, R, W, and X bits only determine if an access succeeds if all bytes of that access are covered by that PMP entry. For example, if a PMP entry is configured to match the four-byte range 0xC–0xF, then an 8-byte access to the range 0x8–0xF will fail, assuming that PMP entry is the highest-priority entry that matches those addresses.

While operating in machine mode when the lock bit is clear (L=0), if a PMP entry matches all bytes of an access, the access succeeds. If the lock bit is set (L=1) while in machine mode, then the access depends on the permissions set for that region. Similarly, while in Supervisor mode, the access depends on permissions set for that region.

Failed read or write accesses generate a load or store access exception, and an instruction access fault would occur on a failed instruction fetch. When an exception occurs while attempting to execute from a region without execute permissions, the fault occurs on the fetch and not the branch, so the mepc CSR will reflect the value of the targeted protected region, and not the address of the branch.

It is possible for a single instruction to generate multiple accesses, which may not be mutually atomic. If at least one access generated by an instruction fails, then an exception will occur. It might be possible that other accesses from a single instruction will succeed, with visible side effects. For example, references to virtual memory may be decomposed into multiple accesses.

On some implementations, misaligned loads, stores, and instruction fetches may also be decomposed into multiple accesses, some of which may succeed before an access exception occurs. In particular, a portion of a misaligned store that passes the PMP check may become visible, even if another portion fails the PMP check. The same behavior may manifest for floating-point stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

3.7.6 PMP and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*. When paging is enabled, instructions that access virtual memory may result in multiple physical-memory accesses, including implicit references to the page tables. The PMP checks apply to all of these accesses. The effective privilege mode for implicit page-table accesses is supervisor mode.

Implementations with virtual memory are permitted to perform address translations speculatively and earlier than required by an explicit virtual-memory access. The PMP settings for the resulting physical address may be checked at any point between the address translation and the explicit virtual-memory access. A mis-predicted branch to a non-executable address range does not generate a trap. Hence, when the PMP settings are modified in a manner that affects either the physical memory that holds the page tables or the physical memory to which the page tables point, M-mode software must synchronize the PMP settings with the virtual memory system. This is accomplished by executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0`, after the PMP CSRs are written.

If page-based virtual memory is not implemented, or when it is disabled, memory accesses check the PMP settings synchronously, so no fence is needed.

3.7.7 PMP Limitations

In a system containing multiple harts, each hart has its own PMP device. The PMP permissions on a hart cannot be applied to accesses from other harts in a multi-hart system. In addition, SiFive designs may contain a Front Port to allow external bus masters access to the full memory map of the system. The PMP cannot prevent access from external bus masters on the Front Port.

3.7.8 Behavior for Regions without PMP Protection

If a non-reserved region of the memory map does not have PMP permissions applied, then by default, supervisor or user mode accesses will fail, while machine mode access will be allowed.

Access to reserved regions within a device's memory map (an interrupt controller for example) will return `0x0` on reads, and writes will be ignored. Access to reserved regions outside of a device's memory map without PMP protection will result in a bus error. The bus error can generate an interrupt to the hart using the Bus-Error Unit (BEU). See Chapter 12 for more information.

3.7.9 Cache Flush Behavior on PMP Protected Region

When a line is brought into cache and the PMP is set up with the lock (`L`) bit asserted to protect a part of that line, a data cache flush instruction will generate a store access fault exception if the flush includes any part of the line that is protected. The cache flush instruction does an invalidate and write-back, so it is essentially trying to write back to the memory location that is protected. If a cache flush occurs on a part of the line that was not protected, the flush will succeed and not generate an exception. If a data cache flush is required without a write-back, use the cache discard instruction instead, as this will invalidate but not write back the line.

3.8 Hardware Performance Monitor

The S7 processor core supports a basic hardware performance monitoring (HPM) facility. The performance monitoring facility is divided into two classes of counters: fixed-function and event-programmable counters. These classes consist of a set of fixed counters and their counter-enable registers, as well as a set of event-programmable counters and their event selector registers. The registers are available to control the behavior of the counters. Performance monitoring can be useful for multiple purposes, from optimization to debug.

3.8.1 Performance Monitoring Counters Reset Behavior

The `instret` and `cycle` counters are initialized to zero on system reset. The hardware performance monitor event counters are not initialized on system reset, and thus have an arbitrary value. Users can write desired values to the counter control and status registers (CSRs) to start counting at a given, known value.

3.8.2 Fixed-Function Performance Monitoring Counters

A fixed-function performance monitor counter is hardware wired to only count one specific event type. That is, they cannot be reconfigured with respect to the event type(s) they count. The only modification to the fixed-function performance monitoring counters that can be done is to enable or disable counting, and write the counter value itself.

The S7 processor core contains two fixed-function performance monitoring counters.

Fixed-Function Cycle Counter (`mcycle`)

The fixed-function performance monitoring counter `mcycle` holds a count of the number of clock cycles the hart has executed since some arbitrary time in the past. The `mcycle` counter is read-write and 64 bits wide. Reads of `mcycle` return all 64 bits of the `mcycle` CSR.

Fixed-Function Instructions-Retired Counter (`minstret`)

The fixed-function performance monitoring counter `minstret` holds a count of the number of instructions the hart has retired since some arbitrary time in the past. The `minstret` counter is read-write and 64 bits wide. Reads of `minstret` return all 64 bits of the `minstret` CSR.

3.8.3 Event-Programmable Performance Monitoring Counters

Complementing the fixed-function counters are a set of programmable event counters. The S7 HPM includes two additional event counters, `mhpmcounter3` and `mhpmcounter4`. These programmable event counters are read-write and 64 bits wide. The hardware counters themselves are implemented as 40-bit counters on the S7 core series. These hardware counters can be written to in order to initialize the counter value.

3.8.4 Event Selector Registers

To control the event type to count, event selector CSRs `mhpmevent3` and `mhpmevent4` are used to program the corresponding event counters. These event selector CSRs are 64-bit **WARL** registers.

The event selectors are partitioned into three fields; the lower 8 bits select an event class, a middle set of bits that form a mask of events in that class, with the upper 8-bits used for counter overflow and event filtering.

Figure 8: Event Selector Fields

The counter increments if the event corresponding to any set mask bit occurs. For example, if `mhpmevent3` is set to `0x4200`, then `mhpmcounter3` will increment when either a load instruction or a conditional branch instruction retires. An event selector of 0 means "count nothing".

Counter Overflow and Event Filters

The upper 8-bits of the `mhmpevent` register are used for controlling a counter overflow interrupt, as well as mode-based event filtering. The bit layout is shown below:

Machine Hardware Performance Monitor Event Register			
Bits	**Name**	**Attr**	**Description**
[7:0]	`Class`	WARL	Selects the Event Class to make available for counting
[55:8]	`EventSel`	WARL	Bit-mask of Event(s) to count
[57:56]	Reserved	—	
58	`VUINH`	WARL	Reserved
59	`VSINH`	WARL	Reserved
60	`UINH`	WARL	If set, counting of events in U-mode is inhibited
61	`SINH`	WARL	Reserved
62	`MINH`	RW	If set, then counting of events in M-mode is inhibited
63	`OF`	RW	Overflow status and interrupt disable bit. Set by hardware when counter overflows.

Table 9: `mhpmeventX` Register Bit Layout for Overflow and Filtering

Performance Counter Overflow Operation

Each of the five `xINH` bits inhibits counting of event when the core is in privilege mode `x`. All zeroes in these fields results in counting of events in all modes.

The `OF` bit is set by hardware when the corresponding `hpmcounterX` overflows and remains set until written by software. Since hpmcounter values are unsigned, overflow is defined as incrementing from all-ones to all-zeroes. Note that there is no loss of information after an overflow since the counter wraps around and keeps counting while the sticky `OF` bit remains set.

If an hpmcounter overflows while the associated `OF` bit is zero, then a *count overflow interrupt request* is generated. If an hpmcounter overflows while the associated `OF` bit is one, then no interrupt is generated. Consequently, the `OF` bit also functions as a count overflow interrupt disable for the associated hpmcounter.

Count overflow never results from writes to the `mhpmcounterX` or `mhpmeventX` registers. Overflow occurs only as a result of an event causing a counter to increment.

The counter overflow interrupt is a standard local interrupt that corresponds to bit 13 in the `mip` and `mie` registers. The `mip` `LCOFIP` bit is the interrupt-pending bit for this interrupt and the `mie` `LCOFIE` bit is the interrupt-enable bit for this interrupt. `LCOFI` represents Local Count Overflow Interrupt. If S-mode is present, `sip` and `sie` include the corresponding bits for supervisor interrupt control and status.

Generation of a count overflow interrupt request by an hpmcounter sets the `LCOFIP` bit in the `mip` register and sets the associated `OF` bit. If S-mode is present, the `mideleg` register controls the delegation of this interrupt to S-mode, which sets the `LCOFIP` bit in the `sip` register. The `LCOFIP` bit is cleared by software after servicing the count overflow interrupt.

Multiple simultaneous interrupts destined for the same privilege mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

Note that there are not separate overflow status and overflow interrupt enable bits. In practice, enabling overflow interrupt generation by clearing the `OF` bit is done in conjunction with initializing the counter to a starting value. Once a counter has overflowed, it and the `OF` bit must be reinitialized before another overflow interrupt can be generated.

Software can distinguish newly-overflowed counters which have yet to be serviced by an overflow interrupt handler from overflowed counters that have already been serviced (or that are configured not to generate an interrupt on overflow by maintaining a bit mask reflecting which counters are active and due to overflow eventually.

Disabling Counters in Debug Mode

When set, the `dcsr.stopcount` bit stops counters while in debug mode. This is especially important for `mcycle` and `minstret` counters, since in debug mode, the core is executing ROM instructions in a tight loop. The Freedom Studio Performance Monitor View automatically sets the `dcsr.stopcount` bit.

3.8.5 Event Selector Encodings

Table 10 describes the event selector encodings available. Events are categorized into classes based on the Event Class field encoded in `mhpmeventX[7:0]`. One or more events can be programmed by setting the respective Event Mask bit for a given event class. An event selector encoding of 0 means "count nothing". Multiple events will cause the counter to increment any time any of the selected events occur.

Machine Hardware Performance Monitor Event Register	
Instruction Commit Events, `mhpmeventX[7:0]=0x0`	
Bits	**Description**
8	Exception taken
9	Integer load instruction retired
10	Integer store instruction retired
11	Atomic memory operation retired
12	System instruction retired
13	Integer arithmetic instruction retired
14	Conditional branch retired
15	JAL instruction retired
16	JALR instruction retired
17	Integer multiplication instruction retired
18	Integer division instruction retired
Microarchitectural Events, `mhpmeventX[7:0]=0x1`	
Bits	**Description**
8	Address-generation interlock
9	Long-latency interlock
10	CSR read interlock
11	Instruction cache/ITIM busy
12	Data cache/DTIM busy
13	Branch direction misprediction
14	Branch/jump target misprediction
15	Pipeline flush from CSR write
16	Pipeline flush from other event
17	Integer multiplication interlock
Memory System Events, `mhpmeventX[7:0]=0x2`	
Bits	**Description**
8	Instruction cache miss
9	Memory-mapped I/O access

Table 10: `mhpmevent` Register

Event mask bits that are writable for any event class are writable for all classes. Setting an event mask bit that does not correspond to an event defined in Table 10 has no effect for current implementations. However, future implementations may define new events in that encoding space, so it is not recommended to program unsupported values into the `mhpmevent` registers.

Combining Events

It is common usage to directly count each respective event. Additionally, it is possible to use combinations of these events to count new, unique events. For example, to determine the average cycles per load from a data memory subsystem, program one counter to count "Data cache/DTIM busy" and another counter to count "Integer load instruction retired". Then, simply divide the "Data cache/DTIM busy" cycle count by the "Integer load instruction retired" instruction count and the result is the average cycle time for loads in cycles per instruction.

It is important to be cognizant of the event types being combined; specifically, event types counting occurrences and event types counting cycles.

3.8.6 Counter-Enable Registers

The 32-bit counter-enable register `mcounteren` controls the availability of the hardware performance-monitoring counters to the next-lowest privileged mode.

The settings in these registers only control accessibility. The act of reading or writing these enable registers does not affect the underlying counters, which continue to increment when not accessible.

When any bit in the `mcounteren` register is clear, attempts to read the cycle, time, instruction retire, or `hpmcounterX` register while executing in U-mode will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode, U-mode.

`mcounteren` is a **WARL** register. Any of the bits may contain a hardwired value of zero, indicating reads to the corresponding counter will cause an illegal instruction exception when executing in a less-privileged mode.

3.9 Ports

This section describes the Port interfaces to the S7 core.

3.9.1 Front Port

The Front Port can be used by external masters to read from and write into the memory system utilizing any Memory Port in the Core Complex. The ITIM and DTIM can also be accessed through the Front Port. Transactions cannot be routed from the Front Port to any System or Peripheral Ports.

Accesses from external masters through the Front Port have the same latency to either ITIM or DTIM, as Front Port instruction fetches are considered reads. In other words, code execution or data accesses to either ITIM or DTIM will have the same cycle count. Internal arbitration prioritizes access from the local hart over Front Port accesses, in the event that they happen at the same time.

If a Front Port access targets the Memory Port, a coherency manager is responsible for maintaining coherency with the L1 and L2 caches. A read access can be returned directly from the caches without generating an external bus access. If a write from the Front Port targets a location in the L1 data cache, it results in the line being evicted and invalidated. The write will then allocate to the L2 cache.

Any Front Port access that targets the Memory Port and results in an L1 and L2 cache miss will allocate to the L2 cache.

The U74-MC Core Complex User Guide describes the implementation details of the Front Port.

3.9.2 Memory Port

The Memory Port is used to interface with memory that offers the highest performance for the U74-MC Core Complex, such as DDR. It supports cacheable accesses for data and instructions.

Consult Section 5.1 for further information about the Memory Port and its Physical Memory Attributes.

See the U74-MC Core Complex User Guide for a description of the Memory Port implementation in the U74-MC Core Complex.

3.9.3 Peripheral Port

The Peripheral Port is used to interface with lower speed peripherals and also supports code execution. When a device is attached to the Peripheral Port, it is expected that there are no other masters connected to that device.

Consult Section 5.1 for further information about the Peripheral Port and its Physical Memory Attributes.

See the U74-MC Core Complex User Guide for a description of the Peripheral Port implementation in the U74-MC Core Complex.

3.9.4 System Port

The System Port is used to interface with lower performance memory, like SRAM, memory-mapped I/O (MMIO), and higher speed peripherals. The System Port also supports code execution.

Consult Section 5.1 for further information about the System Port and its Physical Memory Attributes.

See the U74-MC Core Complex User Guide for a description of the System Port implementation in the U74-MC Core Complex.

Chapter 4

U7 RISC-V Core

This chapter describes the 64-bit U7 RISC-V processor core, instruction fetch and execution unit, L1 and L2 memory systems, Physical Memory Protection unit, Hardware Performance Monitor, and external interfaces.

The U7 feature set is summarized in Table 11.

Feature	**Description**
ISA	*RV64GC_Zba_Zbb_Sscofpmf*
SiFive Custom Instruction Extension (SCIE)	Not Present
Privilege Modes	M, U, S
L1 Instruction Cache	32 KiB 2-way instruction cache
L1 Data Cache	32 KiB 4-way data cache
L2 Cache	2 MiB 16-way L2 cache with 2 banks
ECC Support	Single error correction, double error detection on the L1 data cache and L2 cache
Physical Memory Protection	8 regions with a granularity of 4096 bytes
Memory Management Unit	Sv39 virtual memory support with fully-associative 40-entry L1 data and instruction TLBs, and a direct-mapped 512-entry unified TLB

Table 11: U7 Feature Set

4.1 Supported Privilege Modes

The U7 supports the RISC-V supervisor and user modes, providing three levels of privilege: machine (M), user (U), and supervisor (S). U-mode provides a mechanism to isolate application processes from each other and from trusted code running in M-mode. S-mode adds a number of additional CSRs and capabilities.

See *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* for more information on the privilege modes.

4.2 Instruction Memory System

This section describes the instruction memory system of the U7 Application core.

4.2.1 Execution Memory Space

The regions of executable memory consist of all directly addressable memory in the system. The memory includes any volatile or non-volatile memory located off the Core Complex ports, and includes the on-core-complex L2 LIM and L2 Zero Device.

Table 12 shows the executable regions of the U74-MC Core Complex.

Base	Top	Description
`0x0100_0000`	`0x0100_1FFF`	S7 Hart 0 DTIM (8 KiB)
`0x0180_0000`	`0x0180_3FFF`	S7 Hart 0 ITIM
`0x0800_0000`	`0x081F_FFFF`	L2 LIM
`0x0A00_0000`	`0x0A1F_FFFF`	L2 Zero Device
`0x2000_0000`	`0x3FFF_FFFF`	Peripheral Port (512 MiB)
`0x4000_0000`	`0x5FFF_FFFF`	System Port (512 MiB)
`0x8000_0000`	`0x10_7FFF_FFFF`	Memory Port (64 GiB)

Table 12: Executable Memory Regions for the U74-MC Core Complex

All executable regions are treated as instruction cacheable. There is no method to disable this behavior.

Trying to execute an instruction from a non-executable address results in an instruction access trap.

4.2.2 L1 Instruction Cache

The L1 instruction cache is a 32 KiB 2-way set-associative cache. It is virtually-indexed, physically-tagged with a line size of 64 bytes and is read/write-allocate with a random replacement policy. A cache line fill triggers a burst access outside of the Core Complex, starting with the first address of the cache line. There are no write-backs to memory from the instruction cache and it is not kept coherent with rest of the platform memory system. In multi-core systems, the instruction caches are not kept coherent with each other.

Out of reset, all blocks of the instruction cache are invalidated. The access latency of the cache is one clock cycle. There is no way to disable the instruction cache and cache allocations begin immediately out of reset.

The L1 instruction cache has parity error protection support.

4.2.3 Cache Maintenance

The instruction cache supports the `FENCE.I` instruction, which invalidates the entire instruction cache, as described in Section 6.14. Writes to instruction memory from the core or another master must be synchronized with the instruction fetch stream by executing `FENCE.I`.

4.2.4 Coherence with Higher Level Caches

The L1 instruction cache is partially inclusive with the L2 Cache, described in Chapter 14.

When a block of instruction memory is allocated to the L1 cache, it is also allocated to the L2 cache if the access was from the Memory Port. Instruction accesses to all other ports will not allocate to the L2 cache, only the L1 cache.

When a block is evicted from L1, it might still reside in L2, which will reduce access time the next time the block is fetched.

If a hart modifies instruction memory (i.e., self-modifying code), then a `FENCE.I` instruction is required to synchronize the instruction and data streams. Even though `FENCE.I` targets the L1 instruction cache, no cache operation is required on the L2 cache to maintain instruction coherency.

4.2.5 Instruction Fetch Unit

The U7 instruction fetch unit is responsible for keeping the pipeline fed with instructions from memory. The instruction fetch unit delivers up to 8 bytes of instructions per clock cycle to support superscalar instruction execution. Fetches are always word-aligned and there is a one-cycle penalty for branching to a 32-bit instruction that is not word-aligned.

The U7 implements the standard Compressed (C) extension to the RISC-V architecture, which allows for 16-bit RISC-V instructions. As four 16-bit instructions can be fetched per cycle, the instruction fetch unit can be idle when executing programs comprised mostly of compressed 16-bit instructions. This reduces memory accesses and power consumption.

All branches must be aligned to half-word addresses. Otherwise, the fetch generates an instruction address misaligned trap. Trying to fetch from a non-executable or unimplemented address results in an instruction access trap.

4.2.6 Branch Prediction

The U7 instruction fetch unit contains sophisticated predictive hardware to mitigate the performance impact of control hazards within the instruction stream. The instruction fetch unit is decoupled from the execution unit, so that correctly predicted control-flow events usually do not result in execution stalls.

- A 16-entry branch target buffer (BTB), which predicts the target of taken branches and direct jumps;

- A 3.6 KiB branch history table (BHT), which predicts the direction of conditional branches;
- An 8-entry indirect-jump target predictor (IJTP);
- A 6-entry return-address stack (RAS), which predicts the target of procedure returns.

The BHT is a correlating predictor that supports long branch histories. The BTB has one-cycle latency, so that correctly predicted branches and direct jumps result in no penalty, provided the target is 8-byte aligned.

Direct jumps that miss in the BTB result in a one-cycle fetch bubble. This event might not result in any execution stalls if the fetch queue is sufficiently full.

The BHT, IJTP, and RAS take precedence over the BTB. If these structures' predictions disagree with the BTB’s prediction, a one-cycle fetch bubble results. Similar to direct jumps that miss in the BTB, the fetch bubble might not result in an execution stall.

Mispredicted branches usually incur a four-cycle penalty, but sometimes the branch resolves later in the execution pipeline and incurs a six-cycle penalty instead. Mispredicted indirect jumps incur a six-cycle penalty.

Branch prediction is enabled out of reset and cannot be disabled. However, instruction speculation, fetching before a prediction is confirmed, must be enabled in the Feature Disable CSR, described in Chapter 7.

As instruction speculation can occur at any point after it has been enabled, data cacheable regions of memory (i.e., DDR) must be able to respond to instruction fetches immediately after instruction speculation is enabled. If DDR initialization is not completed before instruction speculation is enabled, the memory system must return a decode error (`DECERR`) for accesses made to DDR. The fetch unit will ignore errors associated with speculative accesses and continue to operate normally.

The Branch Prediction Mode CSR, also described in Chapter 7, provides a means to customize the branch predictor behavior to trade average performance for more predictable execution time.

4.3 Execution Pipeline

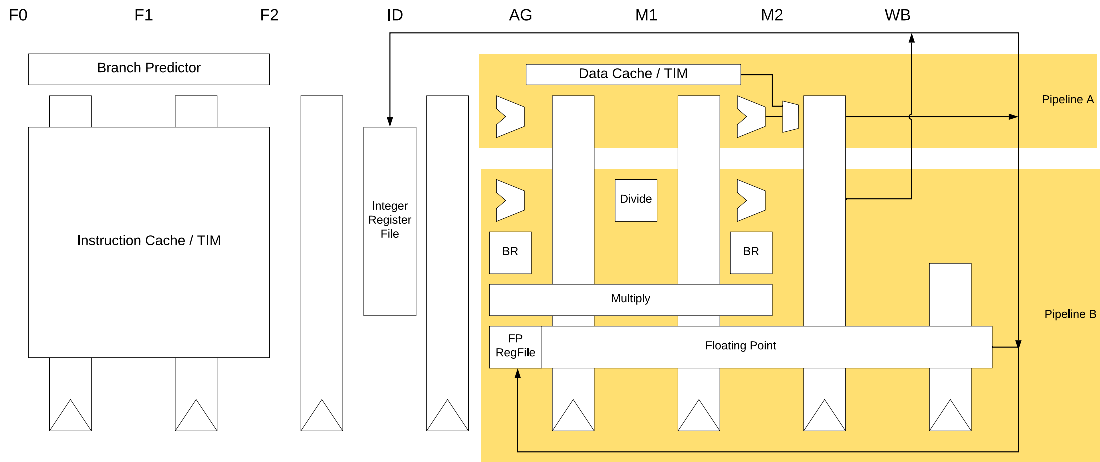

Figure 9: U7 Execution Pipeline

The U7 execution unit is a dual-issue, in-order pipeline. The pipeline comprises eight stages: three stages of instruction fetch (F0-F2), instruction decode (ID), address generation (AG), two stages of data memory access (M1 and M2), and register write-back (WB). The pipeline has a peak execution rate of two instructions per clock cycle, and is fully bypassed so that most instructions have a one-cycle result latency:

- Integer arithmetic and branch instructions can execute in either the AG or M2 pipeline stage. If such an instruction's operands are available when the instruction enters the AG stage, then it executes in AG; otherwise, it executes in M2.
- Loads produce their result in the M2 stage. There is no load-use delay for most integer instructions. However, effective addresses for memory accesses are always computed in the AG stage. Hence, loads, stores, and indirect jumps require their address operands to be ready when the instruction enters AG. If an address-generation operation depends upon a load from memory, then the load-use delay is two cycles.
- Integer multiplication instructions consume their operands in the AG stage and produce their results in the M2 stage. The integer multiplier is fully pipelined.
- Integer division instructions consume their operands in the AG stage. These instructions have between a six-cycle and 68-cycle result latency, depending on the operand values.
- CSR accesses execute in the M2 stage. CSR read data can be bypassed to most integer instructions with no delay. Most CSR writes flush the pipeline, which is a seven-cycle penalty.

Instruction	**Latency**
LW	Three-cycle latency, assuming cache hit[1]
LH, LHU, LB, LBU	Three-cycle latency, assuming cache hit[1]
CSR Reads	One-cycle latency[2]
MUL, MULH, MULHU, MULHSU	Three-cycle latency
DIV, DIVU, REM, REMU	Between six-cycle to 68-cycle latency, depending on operand values[3]
[1]Effective address not ready in AG stage. Load-to-use latency = load-to-use delay + 1 [2] cycle latency = cycle delay + 1 [3]The latency of DIV, DIVU, REM, and REMU instructions can be determined by calculating: `Latency = 2 cycles + log`$_2$`(dividend) - log`$_2$`(divisor) + 1 cycle` if the input is negative + 1 cycle if the output is negative	

Table 13: U7 Instruction Latency

The pipeline has some register dependencies, where it interlocks on read-after-write and write-after-write hazards, so instructions may be scheduled to avoid stalls. Otherwise, the processor can have multiple outstanding memory-mapped I/O accesses, even to the same address. The pipeline implements a flexible dual-instruction-issue scheme. Provided there are no data hazards between a pair of instructions, the two instructions may issue in the same cycle, provided the following constraints are met:

- At most one instruction accesses data memory.
- At most one instruction is a branch or jump.
- At most one instruction is a floating-point arithmetic operation.
- At most one instruction is an integer multiplication or division operation.
- Neither instruction explicitly accesses a CSR.

See Appendix D for a complete list of floating-point unit instruction timings.

4.4 Data Memory System

The data memory system consists of on-core-complex data and the ports in the U74-MC Core Complex memory map, shown in Section 5.2. The on-core-complex data memory consists of a 32 KiB L1 data cache and 2 MiB L2 cache. A design cannot have both data cache and DTIM.

Data accesses are classified as cacheable, for those targeting the Memory Port; or non-cacheable, for those targeting any other port in the Core Complex. Non-cacheable data accesses are collectively called memory-mapped I/O accesses, or MMIOs.

The U7 pipeline allows for multiple outstanding memory accesses, but only allows one outstanding cache line fill. The number of outstanding MMIOs are implementation dependent. Misaligned accesses are not allowed to any memory region and result in a trap to allow for software emulation.

4.4.1 L1 Data Cache

The L1 data cache is a 32 KiB 4-way set-associative cache. It is virtually-indexed, physically-tagged with a line size of 64 bytes and is read/write-allocate with a PLRU replacement policy. The cache operates in write-back mode; this means that if a cache line is dirty, it is written back to memory when evicted. Out of reset, all lines of the cache are invalidated.

The L1 data cache supports ECC protection, as described in Chapter 17.

The Memory Port address range is the only cacheable region of memory. A cache line fill triggers a burst access starting with the first address of the cache line. On a cache hit, the access latency is two clock cycles for words and double-words, and three clock cycles for smaller quantities. Stores are pipelined and commit on cycles where the data memory system is otherwise idle. Pending stores are stored in a buffer, which drains whenever there is an idle cycle or another store. Loads to addresses currently in the store pipeline result in a five-cycle penalty.

The data cache supports only one outstanding line fill. Once a cacheable access is made that misses, another cannot be issued until the line fill completes. However, other MMIOs can be issued before or after the line fill as long as there are no address or register hazards.

The data cache cannot be disabled and the properties of the Memory Port cannot be modified to prevent cacheable accesses.

4.4.2 Cache Maintenance Operations

The data cache supports `CFLUSH.D.L1` and `CDISCARD.D.L1`. The instruction `CFLUSH.D.L1` cleans and invalidates the specified line or all cache lines. The instruction `CDISCARD.D.L1` invalidates the specified line or all cache lines.

These custom instructions are further described in Chapter 7.

4.4.3 L1 Data Cache Coherency

All of the L1 data caches in the Core Complex are kept coherent with an integrated coherency manager. This is an automatic feature and cannot be disabled. The `CFLUSH.D.L1` and `CDISCARD.D.L1` instructions only affect the core that executed the instruction. They are not broadcast to all cores in the Complex.

4.4.4 Coherence with Higher Level Caches

The L1 data cache is inclusive with the L2 Cache, described in Chapter 14.

When a block of data is allocated to the L1 cache, it is also allocated to the L2 cache. When a block is evicted from the L1, the corresponding line in L2 is then updated and marked dirty.

The custom instructions `CFLUSH.D.L1` and `CDISCARD.D.L1` only target the L1 data cache, and do not impact the L2 cache. Flush capability of the L2 Cache Controller is described in Section 14.4.9.

4.5 Atomic Memory Operations

The U7 core supports the RISC-V standard Atomic (A) extension on the Memory Port, Peripheral Port, and internal memory regions.

Atomic instructions that target the Memory Port are implemented in the data cache and are not observable on the external data bus. The load-reserved (LR) and store-conditional (SC) instructions are special atomic instructions that are only supported in data cacheable regions. They will generate a precise access exception if targeted at uncacheable data regions.

Atomic memory operations are not supported on the System Port. Atomic operations that target the System Port will generate a precise access exception.

See Section 6.4 for more information on the instructions added by this extension.

4.6 Floating-Point Unit (FPU)

The U7 FPU provides full hardware support for the IEEE 754-2008 floating-point standard for 32-bit single-precision and 64-bit double-precision arithmetic. The FPU includes a fully pipelined fused-multiply-add unit and an iterative divide and square-root unit, magnitude comparators, and float-to-integer conversion units, all with full hardware support for subnormals and all IEEE default values.

Section 6.5 describes the 32-bit single-precision instructions. Section 6.6 describes the 64-bit double-precision instructions.

The FPU comes up disabled on reset. First initialize `fcsr` and `mstatus.FS` prior to executing any floating-point instructions. In the `freedom-metal` startup code, write `mstatus.FS[1:0]` to `0x1`.

4.7 Virtual Memory Support

The U7 supports virtual memory through the use of a Memory Management Unit (MMU). The MMU supports the Sv39 address mode as described in *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*. SiFive's Sv39 implementation provides a 39-bit virtual address space using 38-bits of physical address space. Supported page sizes include 4 KiB pages, 2 MiB megapages, and 1 GiB gigapages. The default Linux page size (PAGESIZE) is 4 KiB.

The translation lookaside buffers (TLBs) are address translation caches within the MMU. Translation is accomplished through page table entries (PTE) that reside in the TLB region. A hardware page-table walker refills the TLBs upon a miss. The PTE entries are fetched from a region defined by the root page table base address in the Supervisor Address Translation and Protection (`satp`) CSR. Each PTE contains the information necessary to translate the virtual memory address to a physical address on the design.

There are both level 1 and level 2 TLB entries. Level 1 entries contain separate instruction buffers (ITLB) and data buffers (DTLB) since they are accessed in different pipeline stages. The ITLB and DTLB each contain 40 entries, which are fully associative. Level 2 TLB entries are unified, and contain 512 direct-mapped TLB entries. Level 2 TLB are all 4 KiB pages. A block diagram of the instruction and data memory access from the L2 into the MMU TLB is shown below.

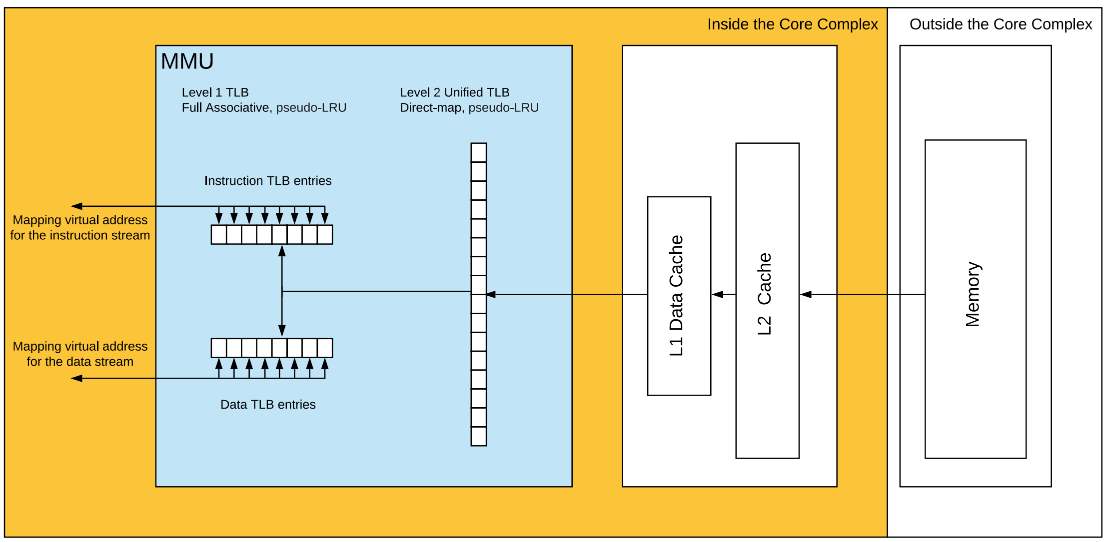

Figure 10: TLB Update Flow

Behaviors of the hardware are described below.

- When there is a TLB miss in the L1 ITLB or DTLB, the L2 unified TLB will populate the L1 TLB with the correct PTE, if it exists
- When there is a miss in both L1 and L2 TLB, a hardware page table walk will occur by the MMU to fill the TLB page table entry from the memory. The memory location where the hart will start fetching TLB page table entry from is determined by the physical page number (`PPN`) field in the Supervisor Address Translation and Protection (`satp`) CSR. The refill will occur from the data cache if it exists there, otherwise it will refill from the L2 cache. If the L2 cache does not contain the data, then it will be fetched from system memory.
- Both L1 and L2 unified TLB page table entry replacement policy is pseudo-LRU
- When L1 TLB entry is evicted, this entry is not updated in the L2 unified TLB
- When the L1 TLB entry is updated from L2, the entry will reside in L2 and will not be removed

- Executing the `SFENCE.VMA` instruction will invalidate both L1 and L2 TLB entries

4.7.1 Address and Page Table Formats

An Sv39 virtual address is partitioned as shown below. Note that address bits [63:39] of every instruction fetch, load, and store operation must be equal to bit 38, or else a page-fault exception will occur.

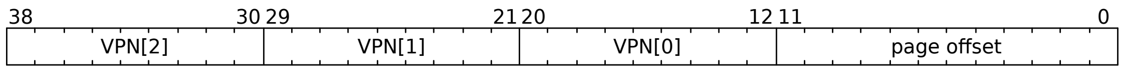

Figure 11: Sv39 Virtual Address

The 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.

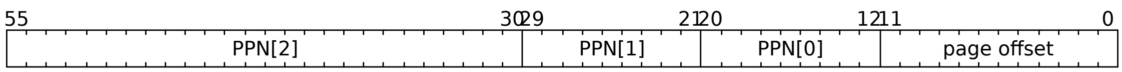

Figure 12: Sv39 Physical Address

Sv39 page tables contain 2^9 page table entries (PTEs), eight bytes each. A page table is exactly the size of a page and must always be aligned to a page boundary. As mentioned, `satp.PPN` holds the physical page number of the root page table. Any level of PTE may be a leaf PTE, and all page sizes (4 KiB, 2 MiB, and 1 GiB) must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

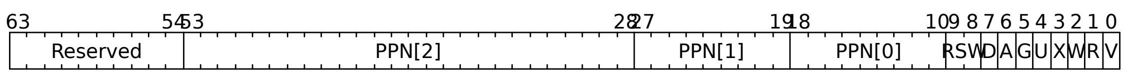

Figure 13: Sv39 PTE Format

A description of the PTE configuration bits can be found in Table 14.

Bits	Description
0	**V: Valid** • 0x0 - Page table entry not valid • 0x1 - Page table entry valid
1	**R: Readable** • 0x0 - Page table entry not readable • 0x1 - Page table entry readable
2	**W: Writable** • 0x0 - Page table entry not writable • 0x1 - Page table entry writable
3	**X: Executable** • 0x0 - Page table entry not executable • 0x1 - Page table entry executable
4	**U: User mode access** • 0x0 - No access to user mode software • 0x1 - Access granted to user mode software
5	**G: Global mapping** • 0x0 - This mapping does not exist globally • 0x1 - This mapping exists globally
6	**A: Accessed** • 0x0 - Leaf page table entry has not been read, written, or fetched since the last time `A` was cleared • 0x1 - Leaf page table entry has been read, written, or fetched since the last time `A` was cleared
7	**D: Dirty** • 0x0 - The virtual page has not been written since the last time `D` was cleared • 0x1 - The virtual page has been written since the last time `D` was cleared
[9:8]	**RSW: Supervisor software use** • X - Open for supervisor software use

Table 14: PTE Configuration Bits

Page Table Configurations

Read, write, and execute permissions for Sv39 are summarized in Table 15. The value `PTE.V=1` indicates the PTE is valid, while `PTE.V=0` means all other bits in PTE are don't cares, and software can use these freely. The value `PTE.R=1` indicates the page is readable. Likewise, `PTE.W=1` indicates the page is writable, while `PTE.X=1` means the page is executable. When `PTE.V=0`, `PTE.R=0`, and `PTE.W=0`, this indicates the PTE is a pointer to the next level page table, otherwise it is a leaf PTE. If a page is marked writable, it must also be marked readable. Combinations of `PTE.W=1` and `PTE.R=0` are not currently supported.

X	W	R	Meaning
0	0	0	Pointer to next level of page table
0	0	1	Read-only page
0	1	0	Reserved
0	1	1	Read-write page
1	0	0	Execute-only page
1	0	1	Read-execute page
1	1	0	Reserved
1	1	1	Read-write-execute page

Table 15: PTE Encoding fields

A fetch page-fault exception will occur if an instruction is fetched from a page that does not have execute permissions. A load page-fault exception will occur if a load or load-reserved instruction falls within a page without read permissions. A store page-fault exception will occur if a store, store-conditional, or AMO instruction falls within a page without write permissions.

The value `PTE.U=1` indicates the page is accessible to user mode. Supervisor mode software may also perform loads and stores to a page marked with `PTE.U=1`, but only if `sstatus.SUM=1`. The `sstatus.SUM` bit modifies the privilege of supervisor mode loads and stores to virtual memory. Supervisor mode software may not execute code on any page marked with `PTE.U=1`.

Two schemes to manage the `A` and `D` bits are permitted:

- When a virtual page is accessed and the `A` bit is clear, or is written and the `D` bit is clear, a page-fault exception is raised
- When a virtual page is accessed and the `A` bit is clear, or is written and the `D` bit is clear, the corresponding bit(s) are set in the PTE. The PTE update is atomic with respect to other accesses to the PTE, and memory access will not occur until the PTE update is visible globally.

For non-leaf PTEs, the `D`, `A`, and `U` bits are reserved for future use and must be cleared by software for forward compatibility.

It is important to note the U7 does not automatically set the accessed (`A`) and dirty (`D`) bits in a Sv39 Page Table Entry (PTE). Instead, the U7 MMU will raise a page fault exception for a read to a page with `PTE.A=0` or a write to a page with `PTE.D=0`.

4.7.2 Supervisor Address Translation and Protection Register (`satp`)

The `satp` register is a 64-bit read/write register used to control supervisor address translation and protection.

Supervisor Address Translation and Protection Register (`satp`)			
CSR	`0x180`		
Bits	**Field Name**	**Attr.**	**Description**
[43:0]	`PPN`	WARL	Holds the physical page number of the root page table, which is the supervisor physical address divided by 4 KiB
[59:44]	`ASID`	WARL	An address space identifier used to facilitate address-translation fences on a per-address-space basis
[63:60]	`MODE`	WARL	Determines the selected address-translation scheme. • 0x0 - Bare mode, no translation enabled • 0x8 - Page-based 38-bit virtual addressing (Sv39)

Table 16: Supervisor Address Translation and Protection Register

When `satp.MODE=0x0`, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in Section 4.8. In this case, the remaining fields in `satp` have no effect.

Note that writing `satp` does not imply any ordering constraints between page-table updates and subsequent address translations. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an `SFENCE.VMA` instruction after writing `satp`, which will:

1. Synchronize page table writes and address translation hardware for higher privilege levels
2. Guarantee previous stores are ordered before all subsequent references from the hart to the memory management data structures
3. Flush L1 and L2 unified TLB entry

Note

Content from Section 4.7.3, Section 4.7.4, Section 4.7.5, and Section 4.7.6 are directly from *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

4.7.3 Supervisor Memory-Management Fence Instruction (`SFENCE.VMA`)

The supervisor memory-management fence instruction `SFENCE.VMA` is used to synchronize updates to in-memory memory-management data structures with current execution.

31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	func3	rd	opcode
SFENCE.VMA	asid	vaddr	PRIV	0	SYSTEM

Figure 14: `SFENCE.VMA` Instruction

Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an `SFENCE.VMA` instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before all subsequent implicit references from that hart to the memory-management data structures.

The `SFENCE.VMA` is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. `SFENCE.VMA` is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

`SFENCE.VMA` orders only the local hart's implicit references to the memory-management data structures.

Consequently, other harts must be notified separately when the memory-management data structures have been modified. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local `SFENCE.VMA` in the interrupt handler of the remote thread, and finally 4) signal back to the originating thread that the operation is complete. This is, of course, the RISC-V analog to a TLB shootdown.

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page or superpage), `rs1` can specify a virtual address within that mapping to affect a translation fence for that mapping only. Furthermore, for the common case that the translation data structures have only been modified for a single address-space identifier, `rs2` can specify the address space. The behavior of `SFENCE.VMA` depends on `rs1` and `rs2` as follows:

- If `rs1=x0` and `rs2=x0`, the fence orders all reads and writes made to any level of the page tables, for all address spaces
- If `rs1=x0` and `rs2≠x0`, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register rs2. Accesses to global mappings are not ordered.
- If `rs1≠x0` and `rs2=x0`, the fence orders only reads and writes made to the leaf page table entry corresponding to the virtual address in `rs1`, for all address spaces
- If `rs1≠x0` and `rs2≠x0`, the fence orders only reads and writes made to the leaf page table entry corresponding to the virtual address in `rs1`, for the address space identified by integer register `rs2`. Accesses to global mappings are not ordered.

When `rs2≠x0`, bits `[SXLEN-1:ASIDMAX]` of the value held in `rs2` are reserved. Furthermore, if ASIDLEN<ASIDMAX, bits `[ASIDMAX-1:ASIDLEN]` of the value held in `rs2` are ignored.

4.7.4 Scenarios that Require SFENCE.VMA Instruction

The following common situations typically require executing an SFENCE.VMA instruction:

- When software recycles an ASID (i.e., reassociates it with a different page table), it should first change satp to point to the new page table using the recycled ASID, then execute SFENCE.VMA with rs1=x0 and rs2 set to the recycled ASID. Alternatively, software can execute the same SFENCE.VMA instruction while a different ASID is loaded into satp, provided the next time satp is loaded with the recycled ASID, it is simultaneously loaded with the new page table.
- If the implementation does not provide ASIDs, or software chooses to always use ASID 0, then after every satp write, software should execute SFENCE.VMA with rs1=x0. In the common case that no global translations have been modified, rs2 should be set to a register other than x0 but which contains the value zero, so that global translations are not flushed.
- If software modifies a non-leaf PTE, it should execute SFENCE.VMA with rs1=x0. If any PTE along the traversal path had its G bit set, rs2 must be x0; otherwise, rs2 should be set to the ASID for which the translation is being modified.
- If software modifies a leaf PTE, it should execute SFENCE.VMA with rs1 set to a virtual address within the page. If any PTE along the traversal path had its G bit set, rs2 must be x0; otherwise, rs2 should be set to the ASID for which the translation is being modified.
- For the special cases of increasing the permissions on a leaf PTE and changing an invalid PTE to a valid leaf, software may choose to execute the SFENCE.VMA lazily. After modifying the PTE but before executing SFENCE.VMA, either the new or old permissions will be used. In the latter case, a page fault exception might occur, at which point software should execute SFENCE.VMA in accordance with the previous bullet point.

Speculation

The U7 will perform a speculative data access as a result of speculative ITLB refill. Changes in the satp register do not necessarily flush TLB entries. It is required to execute an SFENCE.VMA instruction after modifying page table entries in order to flush the cached translations. Exceptions only occur on accesses that are generated as a result of instruction execution, not access that are done speculatively.

ASID Usage for Supervisor Software

Supervisor software that uses ASIDs should use a nonzero ASID value to refer to the same address space across all harts in the supervisor execution environment (SEE) and should not use an ASID value of 0. If supervisor software does not use ASIDs, then the ASID field in the satp CSR should be set to 0.

4.7.5 Trap Virtual Memory

The mstatus.TVM (Trap Virtual Memory) bit supports intercepting supervisor virtual-memory management operations. When TVM=1, attempts to read or write the satp CSR or execute the SFENCE.VMA instruction while executing in S-mode will raise an illegal instruction exception. When TVM=0, these operations are permitted in supervisor mode. TVM is hard-wired to 0 when supervisor mode is not supported. The TVM mechanism improves virtualization efficiency by permitting guest operating systems to execute in supervisor mode, rather than classically virtualizing them in user mode. This approach obviates the need to trap accesses to most S-mode CSRs. Trapping satp accesses and the SFENCE.VMA instruction provides the hooks necessary to lazily populate shadow page tables.

4.7.6 Virtual Address Translation Process

For Sv39, LEVELS equals 3, and PTESIZE equals 8 in the steps below. A virtual address (va) is translated into a physical address (pa) as follows:

1. Let a be satp.ppn × PAGESIZE, and let i = LEVELS − 1.
2. Let pte be the value of the PTE at address a + va.vpn[i] × PTESIZE. If accessing pte violates a PMA or PMP check, raise an access exception corresponding to the original access type.
3. If pte.v = 0, or if pte.r = 0 and pte.w = 1, stop and raise a page-fault exception corresponding to the original access type
4. Otherwise, the PTE is valid. If pte.r = 1 or pte.x = 1, go to Step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let i = i−1. If i < 0, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let a = pte.ppn × PAGESIZE and go to Step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the pte.r, pte.w, pte.x, and pte.u bits, given the current privilege mode and the value of the SUM and MXR fields of the mstatus register. If not, stop and raise a page-fault exception corresponding to the original access type.
6. If i > 0 and pte.ppn[i−1:0] ≠ 0, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type
7. If pte.a = 0, or if the memory access is a store and pte.d = 0, either raise a page-fault exception corresponding to the original access type, or:

    a. Set pte.a to 1 and, if the memory access is a store, also set pte.d to 1

    b. If this access violates a PMA or PMP check, raise an access exception corresponding to the original access type.

    c. This update and the loading of pte in Step 2 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between

8. The translation is successful. The translated physical address is given as follows:

a. `pa.pgoff` = `va.pgoff`

b. If `i` > 0, then this is a superpage translation and `pa.ppn[i−1:0]` = `va.vpn[i−1:0]`

c. `pa.ppn[LEVELS−1:i]` = `pte.ppn[LEVELS−1:i]`

4.7.7 Virtual-to-Physical Address Mapping Example

The following figure is a high-level view of how a virtual address is mapped to a physical address for a Linux application. When the Linux kernel creates a process, it will allocate multiple pages of physical memory to store the code and data. TLB MMU is used to:

- Translate the virtual addresses to physical addresses
- Provide uniform virtual memory layout for a user application
- Protect user applications unauthorized access to other address space

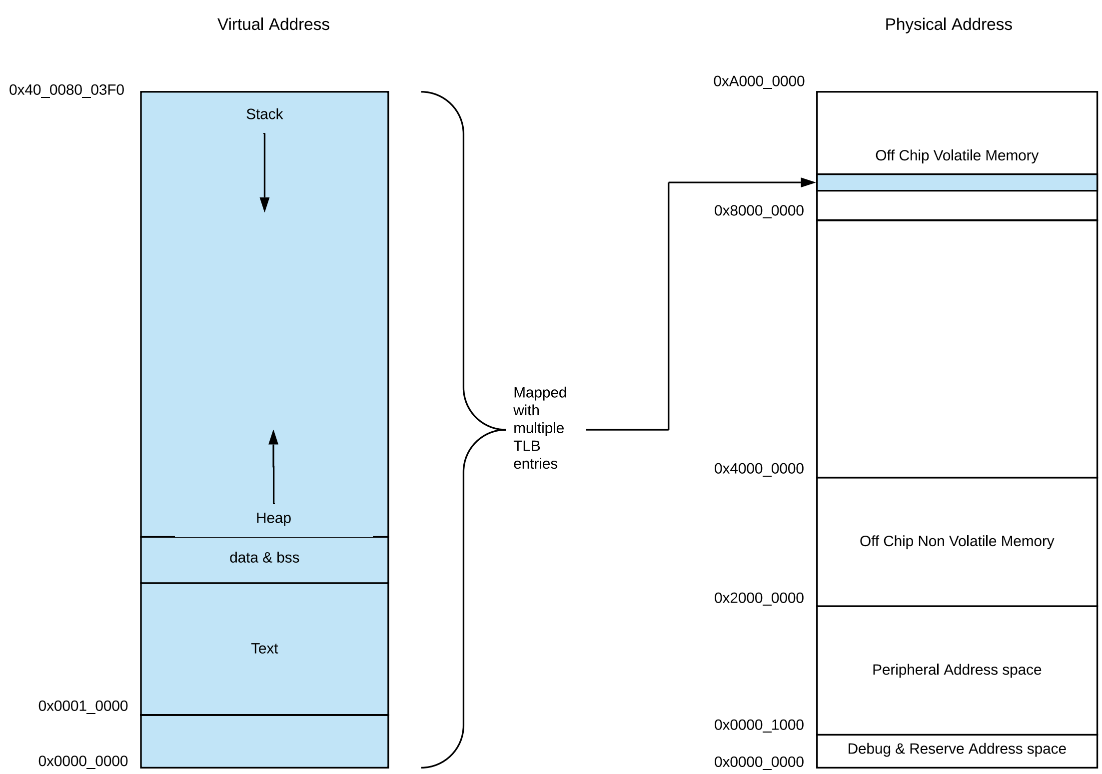

Figure 15: Linux User Application Memory Map Example

In this example, code beginning at virtual address `0x0001_0000` needs to be mapped to an address in off-chip volatile memory.

When the hart tries to execute instructions at this address, it needs to use a matched TLB page table entry to perform a virtual address to physical address translation. If it cannot be located in the L1 instruction TLB, or the L2 unified TLB, the hart will start hardware table walk from the

TLB page table base address. The page table base address is obtained by multiplying `satp.PPN` by the L2 PAGESIZE (4 KiB).

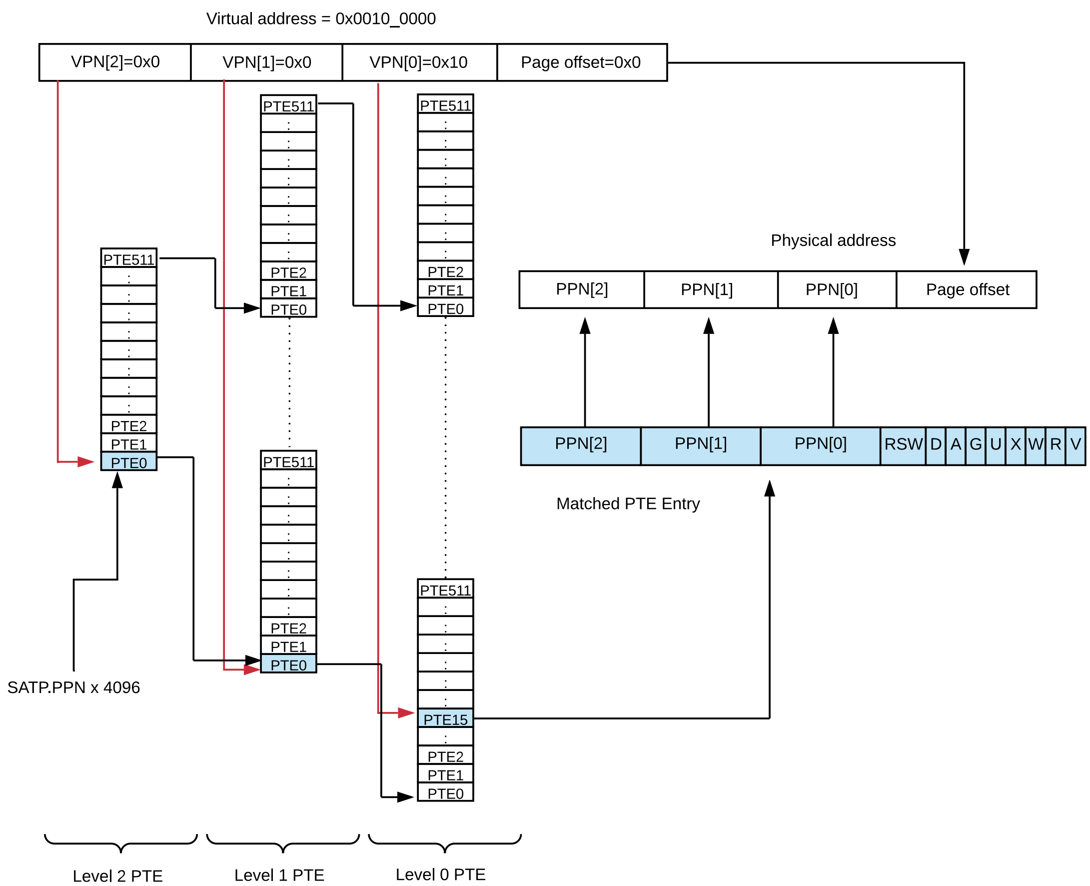

Figure 16: Hardware Table Walk Example

The TLB MMU will execute a page table walk in order to determine the correct mapping for a particular virtual address. Page table entries are pointers to the next level page table if the page is marked as not Readable (`R=0`), not Writable (`W=0`), and not Executable (`X=0`). Otherwise it is a leaf PTE.

In this example, there are 3 levels of page table entries. The hart will start the hardware table walk from the L1 page table entry. In the Sv39 scheme, there are 512 page table entries in L1 page table entry. A hart can quickly locate the entry using `VPN[2]` number, in this case, entry 0. The hart will continue the hardware table walk to the L2 page table entry when the entry doesn't match.

There are 512 clusters of page table entries in L2 and each cluster also has 512 TLB page table entries. In this example, PPNs in the L1 entry 0 is used to locate the right cluster in the L2 page entry. The hart will locate the entry using `VPN[1]` number, in this case, entry 0. The hart will continue a hardware table walk to level 3 page table entry when this entry does not match.

At the level 3 page table entry, there are 512×512 clusters of page table entries, and each cluster has 512 TLB page table entries. In this example, PPNs in the L2 entry 0 is used to locate the correct cluster in the level 3 page entry. The hart then finds the entry using the `VPN[0]` value, which in this case, corresponds to entry 15.

When there is a match in level 3 page table entry, the virtual address will map to a physical address. The physical page number is combined with the page offset to give the complete physical address.

4.7.8 MMU at Reset

The TLB MMU is disabled by default out of reset. All accessed regions have a 1:1 virtual-to-physical address mapping when the MMU is disabled. If the PMP is not yet enabled, all access permissions out of reset are determined by the static PMA values.

4.8 Physical Memory Protection (PMP)

Machine mode is the highest privilege level and by default has read, write, and execute permissions across the entire memory map of the device. However, privilege levels below machine mode do not have read, write, or execute permissions to any region of the device memory map unless it is specifically allowed by the PMP. For the lower privilege levels, the PMP may grant permissions to specific regions of the device's memory map, but it can also revoke permissions when in machine mode.

When programmed accordingly, the PMP will check every access when the hart is operating in supervisor or user modes. For machine mode, PMP checks do not occur unless the lock bit (`L`) is set in the `pmpcfgY` CSR for a particular region.

PMP checks also occur on loads and stores when the machine previous privilege level is supervisor or user (`mstatus.MPP=0x1` or `mstatus.MPP=0x0`), and the Modify Privilege bit is set (`mstatus.MPRV=1`). For virtual address translation, PMP checks are also applied to page table accesses in supervisor mode.

The U7 PMP supports 8 regions with a minimum region size of 4096 bytes.

This section describes how PMP concepts in the RISC-V architecture apply to the U7. For additional information on the PMP refer to *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

4.8.1 PMP Functional Description

The U7 PMP unit has 8 regions and a minimum granularity of 4096 bytes. Access to each region is controlled by an 8-bit `pmpXcfg` field and a corresponding `pmpaddrX` register. Overlapping regions are permitted, where the lower numbered `pmpXcfg` and `pmpaddrX` registers take priority over higher numbered regions. The U7 PMP unit implements the architecturally defined pmpcfgY CSR pmpcfg0, supporting 8 regions. pmpcfg2 is implemented, but hardwired to zero. Access to pmpcfg1 or pmpcfg3 results in an illegal instruction exception.

The PMP registers may only be programmed in M-mode. Ordinarily, the PMP unit enforces permissions on S-mode and U-mode accesses. However, locked regions (see Section 4.8.2) additionally enforce their permissions on M-mode.

4.8.2 PMP Region Locking

The PMP allows for region locking whereby, once a region is locked, further writes to the configuration and address registers are ignored. Locked PMP entries may only be unlocked with a system reset. A region may be locked by setting the L bit in the pmpXcfg register.

In addition to locking the PMP entry, the L bit indicates whether the R/W/X permissions are enforced on machine mode accesses. When the L bit is clear, the R/W/X permissions apply to S-mode and U-mode.

4.8.3 PMP Registers

Each PMP region is described by an 8-bit pmpXcfg field, used in association with a 64-bit pmpaddrX register that holds the base address of the protected region. The range of each region depends on the Addressing (A) mode described in the next section. The pmpXcfg fields reside within 64-bit pmpcfgY CSRs.

Each 8-bit pmpXcfg field includes a read, write, and execute bit, plus a two bit address-matching field A, and a Lock bit, L. Overlapping regions are permitted, where the lowest numbered PMP entry wins for that region.

PMP Configuration Registers

For RV64 architectures, pmpcfg1 and pmpcfg3 are not implemented. This reduces the footprint since pmpcfg2 already contains configuration fields pmp8cfg through pmp11cfg for both RV32 and RV64.

63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0
pmp7cfg	pmp6cfg	pmp5cfg	pmp4cfg	pmp3cfg	pmp2cfg	pmp1cfg	pmp0cfg

Figure 17: RV64 pmpcfg0 Register

63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0
pmp15cfg	pmp14cfg	pmp13cfg	pmp12cfg	pmp11cfg	pmp10cfg	pmp9cfg	pmp8cfg

Figure 18: RV64 pmpcfg2 Register

The pmpcfgY and pmpaddrX registers are only accessible via CSR specific instructions such as csrr for reads, and csrw for writes.

7	6	5	4	3	2	1	0
L (**WARL**)	0 (**WARL**)		A (**WARL**)		X (**WARL**)	W (**WARL**)	R (**WARL**)

Figure 19: RV64 `pmpXcfg` bitfield

Bits	Description
0	**R: Read Permissions** • 0x0 - No read permissions for this region • 0x1 - Read permission granted for this region
1	**W: Write Permissions** • 0x0 - No write permissions for this region • 0x1 - Write permission granted for this region
2	**X: Execute permissions** • 0x0 - No execute permissions for this region • 0x1 - Execute permission granted for this region
[4:3]	**A: Address matching mode** • 0x0 - PMP Entry disabled. No PMP protection applied for any privilege level. • 0x1 - Top of range (TOR) region defined by two adjacent `pmpaddr` registers. The upper limit of region X is defined by `pmpaddrX`, and the base of the region is defined by `pmpaddr(X-1)`. Address 'a' matches the region if [`pmpaddr(X-1)` ≤ a < `pmpaddrX`]. If `pmp0cfg` defines a TOR region, then the base address of that region is 0x0, and `pmpaddr0` defines the upper limit. Supports only a four byte granularity. • 0x2 - Naturally aligned four-byte region (NA4). Supports only a four-byte region with four byte granularity. Not supported on SiFive U7 series cores since minimum granularity is 4 KiB. • 0x3 - Naturally aligned power-of-two region (NAPOT), ≥ 8 bytes. When this setting is programmed, the low bits of the `pmpaddrX` register encode the size, while the upper bits encode the base address right shifted by two. There is a zero bit in between, we will refer to as the least significant zero bit (LSZB).
7	**L: Lock Bit** • 0x0 - PMP Entry Unlocked, no permission restrictions applied to machine mode. PMP entry only applies to S and U modes. • 0x1 - PMP Entry Locked, permissions enforced for all privilege levels including machine mode. Writes to `pmpXcfg` and `pmpcfgY` are ignored and can only be cleared with system reset.
Note: The combination of `R`=0 and `W`=1 is not currently implemented.	

Table 17: `pmpXcfg` Bitfield Description

Out of reset, the PMP register fields `A` and `L` are set to 0. All other hart state is unspecified by *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

Some examples follow using NAPOT address mode.

Base Address	Region Size*	LSZB Position	`pmpaddrX` Value
`0x4000_0000`	8 B	0	`(0x1000_0000 | 1’b0)`
`0x4000_0000`	32 B	2	`(0x1000_0000 | 3’b011)`
`0x4000_0000`	4 KB	9	`(0x1000_0000 | 10’b01_1111_1111)`
`0x4000_0000`	64 KB	13	`(0x1000_0000 | 14’b01_1111_1111_1111)`
`0x4000_0000`	1 MB	17	`(0x1000_0000 | 18’b01_1111_1111_1111_1111)`
*Region size is $2^{(LSZB+3)}$.			

Table 18: `pmpaddrX` Encoding Examples for A=NAPOT

PMP Address Registers

The PMP has 8 address registers. Each address register `pmpaddrX` correlates to the respective `pmpXcfg` field. Each address register contains the base address of the protected region right shifted by two, for a minimum 4-byte alignment.

The maximum encoded address bits per *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* are [55:2].

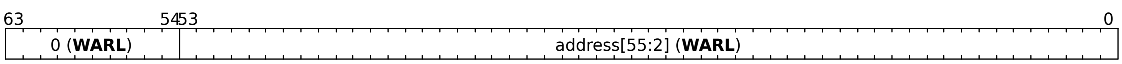

Figure 20: RV64 `pmpaddrX` Register

4.8.4 PMP and PMA

The PMP values are used in conjunction with the Physical Memory Attributes (PMAs) described in Section 5.1. Since the PMAs are static and not configurable, the PMP can only revoke read, write, or execute permissions to the PMA regions if those permissions already apply statically.

4.8.5 PMP Programming Overview

The PMP registers can only be programmed in machine mode. The `pmpaddrX` register should be first programmed with the base address of the protected region, right shifted by two. Then, the `pmpcfgY` register should be programmed with the properly configured 64-bit value containing each properly aligned 8-bit `pmpXcfg` field. Fields that are not used can be simply written to 0, marking them unused.

PMP Programming Example

The following example shows a machine mode only configuration where PMP permissions are applied to three regions of interest, and a fourth region covers the remaining memory map. Recall that lower numbered `pmpXcfg` and `pmpaddrX` registers take priority over higher numbered regions. This rule allows higher numbered PMP registers to have blanket coverage over the entire memory map while allowing lower numbered regions to apply permissions to specific regions of interest. The following example shows a 64 KB Flash region at base address `0x0`, a 32 KB RAM region at base address `0x2000_0000`, and finally a 4 KB peripheral region at base address base `0x3000_0000`. The rest of the memory map is reserved space.

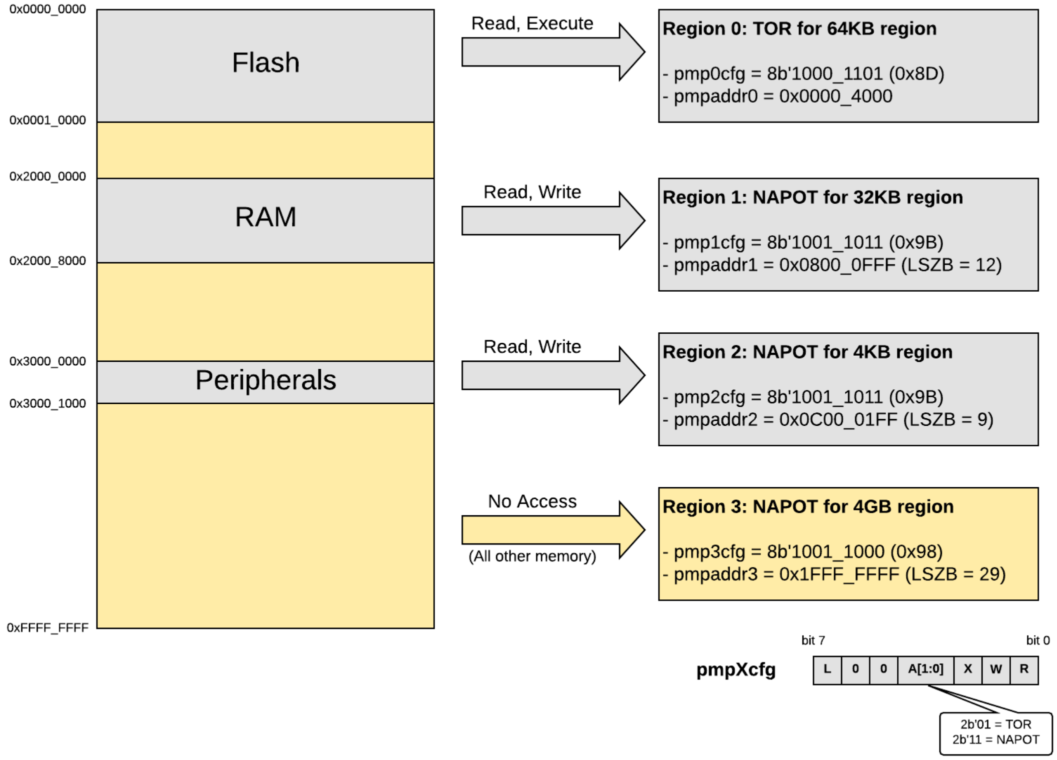

Figure 21: PMP Example Block Diagram

PMP Access Scenarios

The `L`, `R`, `W`, and `X` bits only determine if an access succeeds if all bytes of that access are covered by that PMP entry. For example, if a PMP entry is configured to match the four-byte range 0xC–0xF, then an 8-byte access to the range 0x8–0xF will fail, assuming that PMP entry is the highest-priority entry that matches those addresses.

While operating in machine mode when the lock bit is clear (`L=0`), if a PMP entry matches all bytes of an access, the access succeeds. If the lock bit is set (`L=1`) while in machine mode, then the access depends on the permissions set for that region. Similarly, while in Supervisor mode or User mode, the access depends on permissions set for that region.

Failed read or write accesses generate a load or store access exception, and an instruction access fault would occur on a failed instruction fetch. When an exception occurs while attempting to execute from a region without execute permissions, the fault occurs on the fetch and not the branch, so the `mepc` CSR will reflect the value of the targeted protected region, and not the address of the branch.

It is possible for a single instruction to generate multiple accesses, which may not be mutually atomic. If at least one access generated by an instruction fails, then an exception will occur. It might be possible that other accesses from a single instruction will succeed, with visible side effects. For example, references to virtual memory may be decomposed into multiple accesses.

On some implementations, misaligned loads, stores, and instruction fetches may also be decomposed into multiple accesses, some of which may succeed before an access exception occurs. In particular, a portion of a misaligned store that passes the PMP check may become visible, even if another portion fails the PMP check. The same behavior may manifest for floating-point stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

4.8.6 PMP and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*. When paging is enabled, instructions that access virtual memory may result in multiple physical-memory accesses, including implicit references to the page tables. The PMP checks apply to all of these accesses. The effective privilege mode for implicit page-table accesses is supervisor mode.

Implementations with virtual memory are permitted to perform address translations speculatively and earlier than required by an explicit virtual-memory access. The PMP settings for the resulting physical address may be checked at any point between the address translation and the explicit virtual-memory access. A mis-predicted branch to a non-executable address range does not generate a trap. Hence, when the PMP settings are modified in a manner that affects either the physical memory that holds the page tables or the physical memory to which the page tables point, M-mode software must synchronize the PMP settings with the virtual memory system. This is accomplished by executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0`, after the PMP CSRs are written.

If page-based virtual memory is not implemented, or when it is disabled, memory accesses check the PMP settings synchronously, so no fence is needed.

4.8.7 PMP Limitations

In a system containing multiple harts, each hart has its own PMP device. The PMP permissions on a hart cannot be applied to accesses from other harts in a multi-hart system. In addition, SiFive designs may contain a Front Port to allow external bus masters access to the full memory map of the system. The PMP cannot prevent access from external bus masters on the Front Port.

4.8.8 Behavior for Regions without PMP Protection

If a non-reserved region of the memory map does not have PMP permissions applied, then by default, supervisor or user mode accesses will fail, while machine mode access will be allowed. Access to reserved regions within a device's memory map (an interrupt controller for example) will return `0x0` on reads, and writes will be ignored. Access to reserved regions outside of a device's memory map without PMP protection will result in a bus error. The bus error can generate an interrupt to the hart using the Bus-Error Unit (BEU). See Chapter 12 for more information.

4.8.9 Cache Flush Behavior on PMP Protected Region

When a line is brought into cache and the PMP is set up with the lock (`L`) bit asserted to protect a part of that line, a data cache flush instruction will generate a store access fault exception if the flush includes any part of the line that is protected. The cache flush instruction does an invalidate and write-back, so it is essentially trying to write back to the memory location that is protected. If a cache flush occurs on a part of the line that was not protected, the flush will succeed and not generate an exception. If a data cache flush is required without a write-back, use the cache discard instruction instead, as this will invalidate but not write back the line.

4.9 Hardware Performance Monitor

The U7 processor core supports a basic hardware performance monitoring (HPM) facility. The performance monitoring facility is divided into two classes of counters: fixed-function and event-programmable counters. These classes consist of a set of fixed counters and their counter-enable registers, as well as a set of event-programmable counters and their event selector registers. The registers are available to control the behavior of the counters. Performance monitoring can be useful for multiple purposes, from optimization to debug.

4.9.1 Performance Monitoring Counters Reset Behavior

The `instret` and `cycle` counters are initialized to zero on system reset. The hardware performance monitor event counters are not initialized on system reset, and thus have an arbitrary value. Users can write desired values to the counter control and status registers (CSRs) to start counting at a given, known value.

4.9.2 Fixed-Function Performance Monitoring Counters

A fixed-function performance monitor counter is hardware wired to only count one specific event type. That is, they cannot be reconfigured with respect to the event type(s) they count. The only modification to the fixed-function performance monitoring counters that can be done is to enable or disable counting, and write the counter value itself.

The U7 processor core contains two fixed-function performance monitoring counters.

Fixed-Function Cycle Counter (`mcycle`)

The fixed-function performance monitoring counter `mcycle` holds a count of the number of clock cycles the hart has executed since some arbitrary time in the past. The `mcycle` counter is read-write and 64 bits wide. Reads of `mcycle` return all 64 bits of the `mcycle` CSR.

Fixed-Function Instructions-Retired Counter (`minstret`)

The fixed-function performance monitoring counter `minstret` holds a count of the number of instructions the hart has retired since some arbitrary time in the past. The `minstret` counter is read-write and 64 bits wide. Reads of `minstret` return all 64 bits of the `minstret` CSR.

4.9.3 Event-Programmable Performance Monitoring Counters

Complementing the fixed-function counters are a set of programmable event counters. The U7 HPM includes two additional event counters, `mhpmcounter3` and `mhpmcounter4`. These programmable event counters are read-write and 64 bits wide. The hardware counters themselves are implemented as 40-bit counters on the U7 core series. These hardware counters can be written to in order to initialize the counter value.

4.9.4 Event Selector Registers

To control the event type to count, event selector CSRs `mhpmevent3` and `mhpmevent4` are used to program the corresponding event counters. These event selector CSRs are 64-bit **WARL** registers.

The event selectors are partitioned into three fields; the lower 8 bits select an event class, a middle set of bits that form a mask of events in that class, with the upper 8-bits used for counter overflow and event filtering.

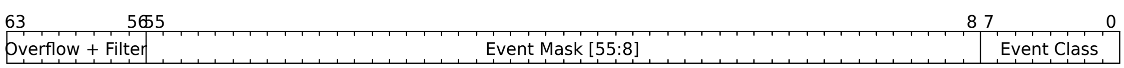

Figure 22: Event Selector Fields

The counter increments if the event corresponding to any set mask bit occurs. For example, if `mhpmevent3` is set to `0x4200`, then `mhpmcounter3` will increment when either a load instruction or a conditional branch instruction retires. An event selector of 0 means "count nothing".

Counter Overflow and Event Filters

The upper 8-bits of the `mhmpevent` register are used for controlling a counter overflow interrupt, as well as mode-based event filtering. The bit layout is shown below:

Machine Hardware Performance Monitor Event Register			
Bits	**Name**	**Attr**	**Description**
[7:0]	`Class`	WARL	Selects the Event Class to make available for counting
[55:8]	`EventSel`	WARL	Bit-mask of Event(s) to count
[57:56]	Reserved	—	
58	`VUINH`	WARL	Reserved
59	`VSINH`	WARL	Reserved
60	`UINH`	WARL	If set, counting of events in U-mode is inhibited
61	`SINH`	WARL	If set, counting of events in S-mode is inhibited
62	`MINH`	RW	If set, then counting of events in M-mode is inhibited
63	`OF`	RW	Overflow status and interrupt disable bit. Set by hardware when counter overflows.

Table 19: `mhpmeventX` Register Bit Layout for Overflow and Filtering

Performance Counter Overflow Operation

Each of the five `xINH` bits inhibits counting of event when the core is in privilege mode `x`. All zeroes in these fields results in counting of events in all modes.

The `OF` bit is set by hardware when the corresponding `hpmcounterX` overflows and remains set until written by software. Since hpmcounter values are unsigned, overflow is defined as incrementing from all-ones to all-zeroes. Note that there is no loss of information after an overflow since the counter wraps around and keeps counting while the sticky `OF` bit remains set.

If an hpmcounter overflows while the associated `OF` bit is zero, then a *count overflow interrupt request* is generated. If an hpmcounter overflows while the associated `OF` bit is one, then no interrupt is generated. Consequently, the `OF` bit also functions as a count overflow interrupt disable for the associated hpmcounter.

Count overflow never results from writes to the `mhpmcounterX` or `mhpmeventX` registers. Overflow occurs only as a result of an event causing a counter to increment.

The counter overflow interrupt is a standard local interrupt that corresponds to bit 13 in the `mip` and `mie` registers. The `mip` `LCOFIP` bit is the interrupt-pending bit for this interrupt and the `mie` `LCOFIE` bit is the interrupt-enable bit for this interrupt. `LCOFI` represents Local Count Overflow Interrupt. If S-mode is present, `sip` and `sie` include the corresponding bits for supervisor interrupt control and status.

Generation of a count overflow interrupt request by an hpmcounter sets the `LCOFIP` bit in the `mip` register and sets the associated `OF` bit. If S-mode is present, the `mideleg` register controls the delegation of this interrupt to S-mode, which sets the `LCOFIP` bit in the `sip` register. The `LCOFIP` bit is cleared by software after servicing the count overflow interrupt.

Multiple simultaneous interrupts destined for the same privilege mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

Note that there are not separate overflow status and overflow interrupt enable bits. In practice, enabling overflow interrupt generation by clearing the `OF` bit is done in conjunction with initializing the counter to a starting value. Once a counter has overflowed, it and the `OF` bit must be reinitialized before another overflow interrupt can be generated.

Software can distinguish newly-overflowed counters which have yet to be serviced by an overflow interrupt handler from overflowed counters that have already been serviced (or that are configured not to generate an interrupt on overflow by maintaining a bit mask reflecting which counters are active and due to overflow eventually.

The Scountovf CSR

The `scountovf` CSR is a 32-bit read-only register that contains shadow copies of the `OF` bits in the 29 `mhpmevent` CSRs. Bit `X` in `scountovf` corresponds to the OF bit in `mhpmeventX`. This register enables supervisor-level overflow interrupt handler software to quickly determine which counter(s) have overflowed without needing to make an environment call to M-mode. The CSR address is 0xDA0.

Read access to `scountovf` bit `X` is subject to the same `mcounteren` CSRs that mediate access to `hpmcounterX` by S-mode software. In S-mode, `scountovf` bit `X` is readable when `mcounteren` bit `X` is set and otherwise reads as zero.

Disabling Counters in Debug Mode

When set, the `dcsr.stopcount` bit stops counters while in debug mode. This is especially important for `mcycle` and `minstret` counters, since in debug mode, the core is executing ROM instructions in a tight loop. The Freedom Studio Performance Monitor View automatically sets the `dcsr.stopcount` bit.

4.9.5 Event Selector Encodings

Table 20 describes the event selector encodings available. Events are categorized into classes based on the Event Class field encoded in `mhpmeventX[7:0]`. One or more events can be programmed by setting the respective Event Mask bit for a given event class. An event selector encoding of 0 means "count nothing". Multiple events will cause the counter to increment any time any of the selected events occur.

Machine Hardware Performance Monitor Event Register	
Instruction Commit Events, `mhpmeventX[7:0]=0x0`	
Bits	**Description**
8	Exception taken
9	Integer load instruction retired
10	Integer store instruction retired
11	Atomic memory operation retired
12	System instruction retired
13	Integer arithmetic instruction retired
14	Conditional branch retired
15	JAL instruction retired
16	JALR instruction retired
17	Integer multiplication instruction retired
18	Integer division instruction retired
19	Floating-point load instruction retired
20	Floating-point store instruction retired
21	Floating-point addition retired
22	Floating-point multiplication retired
23	Floating-point fused multiply-add retired
24	Floating-point division or square-root retired
25	Other floating-point instruction retired
Microarchitectural Events, `mhpmeventX[7:0]=0x1`	
Bits	**Description**
8	Address-generation interlock
9	Long-latency interlock
10	CSR read interlock
11	Instruction cache/ITIM busy
12	Data cache/DTIM busy
13	Branch direction misprediction
14	Branch/jump target misprediction
15	Pipeline flush from CSR write
16	Pipeline flush from other event
17	Integer multiplication interlock
18	Floating-point interlock
Memory System Events, `mhpmeventX[7:0]=0x2`	
Bits	**Description**
8	Instruction cache miss
9	Data cache miss or memory-mapped I/O access
10	Data cache write-back
11	Instruction TLB miss
12	Data TLB miss
13	UTLB miss

Table 20: `mhpmevent` Register

Event mask bits that are writable for any event class are writable for all classes. Setting an event mask bit that does not correspond to an event defined in Table 20 has no effect for current implementations. However, future implementations may define new events in that encoding space, so it is not recommended to program unsupported values into the `mhpmevent` registers.

Combining Events

It is common usage to directly count each respective event. Additionally, it is possible to use combinations of these events to count new, unique events. For example, to determine the average cycles per load from a data memory subsystem, program one counter to count "Data cache/DTIM busy" and another counter to count "Integer load instruction retired". Then, simply divide the "Data cache/DTIM busy" cycle count by the "Integer load instruction retired" instruction count and the result is the average cycle time for loads in cycles per instruction.

It is important to be cognizant of the event types being combined; specifically, event types counting occurrences and event types counting cycles.

4.9.6 Counter-Enable Registers

The 32-bit counter-enable registers `mcounteren` and `scounteren` control the availability of the hardware performance-monitoring counters to the next-lowest privileged mode.

The settings in these registers only control accessibility. The act of reading or writing these enable registers does not affect the underlying counters, which continue to increment when not accessible.

When any bit in the `mcounteren` register is clear, attempts to read the cycle, time, instruction retire, or `hpmcounterX` register while executing in S-mode will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode, S-mode.

The same bit positions in the `scounteren` register analogously control access to these registers while executing in U-mode. If S-mode is permitted to access a counter register and the corresponding bit is set in `scounteren`, then U-mode is also permitted to access that register.

`mcounteren` and `scounteren` are **WARL** registers. Any of the bits may contain a hardwired value of zero, indicating reads to the corresponding counter will cause an illegal instruction exception when executing in a less-privileged mode.

4.10 Ports

This section describes the Port interfaces to the U7 core.

4.10.1 Front Port

The Front Port can be used by external masters to read from and write into the memory system utilizing any Memory Port in the Core Complex. Transactions cannot be routed from the Front Port to any System or Peripheral Ports.

If a Front Port access targets the Memory Port, a coherency manager is responsible for maintaining coherency with the L1 and L2 caches. A read access can be returned directly from the caches without generating an external bus access. If a write from the Front Port targets a location in the L1 data cache, it results in the line being evicted and invalidated. The write will then allocate to the L2 cache.

Any Front Port access that targets the Memory Port and results in an L1 and L2 cache miss will allocate to the L2 cache.

The U74-MC Core Complex User Guide describes the implementation details of the Front Port.

4.10.2 Memory Port

The Memory Port is used to interface with memory that offers the highest performance for the U74-MC Core Complex, such as DDR. It supports cacheable accesses for data and instructions.

Consult Section 5.1 for further information about the Memory Port and its Physical Memory Attributes.

See the U74-MC Core Complex User Guide for a description of the Memory Port implementation in the U74-MC Core Complex.

4.10.3 Peripheral Port

The Peripheral Port is used to interface with lower speed peripherals and also supports code execution. When a device is attached to the Peripheral Port, it is expected that there are no other masters connected to that device.

Consult Section 5.1 for further information about the Peripheral Port and its Physical Memory Attributes.

See the U74-MC Core Complex User Guide for a description of the Peripheral Port implementation in the U74-MC Core Complex.

4.10.4 System Port

The System Port is used to interface with lower performance memory, like SRAM, memory-mapped I/O (MMIO), and higher speed peripherals. The System Port also supports code execution.

Consult Section 5.1 for further information about the System Port and its Physical Memory Attributes.

See the U74-MC Core Complex User Guide for a description of the System Port implementation in the U74-MC Core Complex.

Chapter 5

Physical Memory Attributes and Memory Map

This chapter describes the U74-MC Core Complex physical memory attributes and memory map.

5.1 Physical Memory Attributes Overview

The memory map is divided into different regions covering on-core-complex memory, system memory, peripherals, and empty holes. Physical memory attributes (PMAs) describe the properties of the accesses that can be made to each region in the memory map. These properties encompass the type of access that may be performed: execute, read, or write. As well as other optional attributes related to the access, such as supported access size, alignment, atomic operations, and cacheability.

RISC-V utilizes a simpler approach than other processor architectures in defining the attributes of memory accesses. Instead of defining access characteristics in page table descriptors or memory protection logic, the properties are fixed for memory regions or may only be modified in platform-specific control registers. As most systems don't require the ability to modify PMAs, SiFive cores only support fixed PMAs, which are set at design time. This results in a simpler design with lower gate count and power savings, and an easier programming interface.

External memory map regions are accessed through a specific port type and that port type is used to define the PMAs. The port types are Memory, Peripheral, and System. Memory map regions defined for internal memory and internal control regions also have a predefined PMA based on the underlying contents of the region.

The assigned PMA properties and attributes for U74-MC Core Complex memory regions are shown in Table 21 and Table 22 for external and internal regions, respectively.

The configured memory regions of the U74-MC Core Complex are listed with their attributes in Table 23.

Port Type	Access Properties	Attributes
Memory Port	Read, Write, Execute	Atomics+LR/SC, Data Cacheable, Instruction Cacheable, Instruction Speculation
Peripheral Port	Read, Write, Execute	Atomics, Instruction Cacheable
System Port	Read, Write, Execute	Instruction Cacheable

Table 21: Physical Memory Attributes for External Regions

Region	Access Properties	Attributes
Bus-Error Unit	Read, Write	Atomics
CLINT	Read, Write	Atomics
DTIM	Read, Write, Execute	Atomics
Debug	None	N/A
Error Device	Read, Write, Execute	Atomics
ITIM	Read, Write, Execute	Atomics, Instruction Speculation
L2 Cache Controller	Read, Write	Atomics
L2 LIM	Read, Write, Execute	Atomics
L2 Prefetcher	Read, Write	Atomics
L2 Zero Device	Read, Write, Execute	Atomics, Instruction Cacheable
PLIC	Read, Write	Atomics
Reserved	None	N/A
SLPC	Read, Write	Atomics

Table 22: Physical Memory Attributes for Internal Regions

All memory map regions support word, half-word, and byte size data accesses.

Atomic access support enables the RISC-V standard Atomic (A) Extension for atomic instructions. These atomic instructions are further documented in Section 3.6 for the S7 core and Section 4.5 for the U7 core. The load-reserved (LR) and store-conditional (SC) instructions are only supported on the data cacheable region, marked in Table 21 with "Atomics+LR/SC".

No region supports unaligned accesses. An unaligned access will generate the appropriate trap: instruction address misaligned, load address misaligned, or store/AMO address misaligned.

The Physical Memory Protection unit is capable of controlling access properties based on address ranges, not ports. It has no control over the attributes of an address range, however.

Note

The Debug and Error Device regions have special behavior. The Debug region is reserved for use from a Debugger, and all accesses to it from the core in non-Debug mode will trap. The Error Device will also trap all accesses, as described in Chapter 11.

5.2 Memory Map

The memory map of the U74-MC Core Complex is shown in Table 23.

Base	Top	PMA	Description
0x00_0000_0000	0x00_0000_0FFF		Debug
0x00_0000_1000	0x00_0000_2FFF		Reserved
0x00_0000_3000	0x00_0000_3FFF	RWX A	Error Device
0x00_0000_4000	0x00_00FF_FFFF		Reserved
0x00_0100_0000	0x00_0100_1FFF	RWX A	S7 Hart 0 DTIM (8 KiB)
0x00_0100_2000	0x00_016F_FFFF		Reserved
0x00_0170_0000	0x00_0170_0FFF	RW A	S7 Hart 0 Bus-Error Unit
0x00_0170_1000	0x00_0170_1FFF	RW A	U7 Hart 1 Bus-Error Unit
0x00_0170_2000	0x00_0170_2FFF	RW A	U7 Hart 2 Bus-Error Unit
0x00_0170_3000	0x00_0170_3FFF	RW A	U7 Hart 3 Bus-Error Unit
0x00_0170_4000	0x00_0170_4FFF	RW A	U7 Hart 4 Bus-Error Unit
0x00_0170_5000	0x00_017F_FFFF		Reserved
0x00_0180_0000	0x00_0180_3FFF	RWX A	S7 Hart 0 ITIM
0x00_0180_4000	0x00_01FF_FFFF		Reserved
0x00_0200_0000	0x00_0200_FFFF	RW A	CLINT
0x00_0201_0000	0x00_0201_3FFF	RW A	L2 Cache Controller
0x00_0201_4000	0x00_0202_FFFF		Reserved
0x00_0203_0000	0x00_0203_1FFF	RW A	U7 Hart 1 L2 Prefetcher
0x00_0203_2000	0x00_0203_3FFF	RW A	U7 Hart 2 L2 Prefetcher
0x00_0203_4000	0x00_0203_5FFF	RW A	U7 Hart 3 L2 Prefetcher
0x00_0203_6000	0x00_0203_7FFF	RW A	U7 Hart 4 L2 Prefetcher
0x00_0203_8000	0x00_0300_7FFF		Reserved
0x00_0300_8000	0x00_0300_8FFF	RW A	SLPC
0x00_0300_9000	0x00_07FF_FFFF		Reserved
0x00_0800_0000	0x00_081F_FFFF	RWX A	L2 LIM
0x00_0820_0000	0x00_09FF_FFFF		Reserved
0x00_0A00_0000	0x00_0A1F_FFFF	RWXI A	L2 Zero Device
0x00_0A20_0000	0x00_0BFF_FFFF		Reserved
0x00_0C00_0000	0x00_0FFF_FFFF	RW A	PLIC
0x00_1000_0000	0x00_1FFF_FFFF		Reserved
0x00_2000_0000	0x00_3FFF_FFFF	RWXI A	Peripheral Port (512 MiB)
0x00_4000_0000	0x00_5FFF_FFFF	RWXI	System Port (512 MiB)
0x00_6000_0000	0x00_7FFF_FFFF		Reserved
0x00_8000_0000	0x10_7FFF_FFFF	RWXIDA	Memory Port (64 GiB)
0x10_8000_0000	0xFF_FFFF_FFFF		Reserved

Table 23: U74-MC Core Complex Memory Map. Physical Memory Attributes: **R**–Read, **W**–Write, **X**–Execute, **I**–Instruction Cacheable, **D**–Data Cacheable, **A**–Atomics

Note

Every component that appears in Table 23 is accessible by any core in the Core Complex.

Chapter 6

Programmer's Model

The U74-MC Core Complex implements the 64-bit RISC-V architecture. The following chapter provides a reference for programmers and an explanation of the extensions supported by *RV64GC_Zba_Zbb_Sscofpmf*.

This chapter contains a high-level discussion of the RISC-V instruction set architecture and additional resources which will assist software developers working with RISC-V products. The U74-MC Core Complex is an implementation of the RISC-V *RV64GC_Zba_Zbb_Sscofpmf* architecture, and is guaranteed to be compatible with all applicable RISC-V standards. *RV64GC_Zba_Zbb_Sscofpmf* can emulate almost any other RISC-V ISA extension.

6.1 Base Instruction Formats

RISC-V base instructions are fixed to 32 bits in length and must be aligned on a four-byte boundary in memory. RISC-V ISA keeps the source (`rs1` and `rs2`) and destination (`rd`) registers at the same position in all formats to simplify decoding, with the exception of the 5-bit immediates used in CSR instructions.

The various formats are described in Table 24 below.

Format	Description
R	Format for register-register arithmetic/logical operations.
I	Format for register-immediate ALU operations and loads.
S	Format for stores.
B	Format for branches.
U	Format for 20-bit upper immediate instructions.
J	Format for jumps.

Table 24: Base Instruction Formats

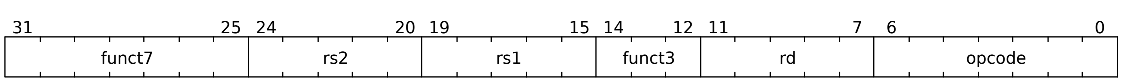

Figure 23: R-Type

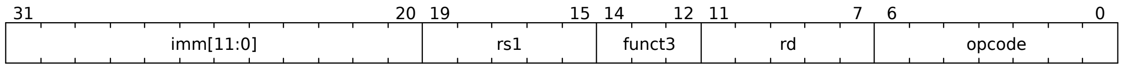

Figure 24: I-Type

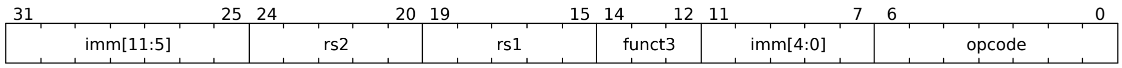

Figure 25: S-Type

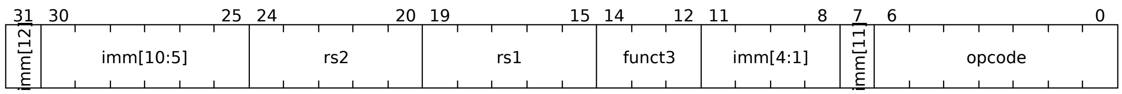

Figure 26: B-Type

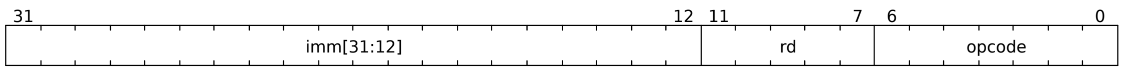

Figure 27: U-Type

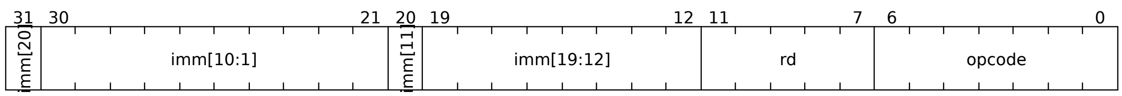

Figure 28: J-Type

The **opcode** field partially specifies an instruction, combined with **funct7** + **funct3** which describe what operation to perform. Each register field (**rs1**, **rs2**, **rd**) holds a 5-bit unsigned integer (0-31) corresponding to a register number (`x0` - `x31`). Sign-extension is one of the most critical operations on immediates (particularly for XLEN>32), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

6.2 RV64I Base Integer Instruction Set

This section discusses the standard integer instructions supported by RISC-V. Integer computational instructions don't cause arithmetic exceptions.

6.2.1 R-Type (Register-Based) Integer Instructions

funct7			funct3		opcode	Instruction
00000000	rs2	rs1	000	rd	0110011	ADD
01000000	rs2	rs1	000	rd	0110011	SUB
00000000	rs2	rs1	001	rd	0110011	SLL
00000000	rs2	rs1	010	rd	0110011	SLT
00000000	rs2	rs1	011	rd	0110011	SLTU
00000000	rs2	rs1	100	rd	0110011	XOR
00000000	rs2	rs1	101	rd	0110011	SRL
01000000	rs2	rs1	101	rd	0110011	SRA
00000000	rs2	rs1	110	rd	0110011	OR
00000000	rs2	rs1	111	rd	0110011	AND

Table 25: R-Type Integer Instructions

Instruction	Description
ADD `rd`, `rs1`, `rs2`	Performs the addition of `rs1` and `rs2`, result stored in `rd`.
SUB `rd`, `rs1`, `rs2`	Performs the subtraction of `rs2` from `rs1`, result stored in `rd`.
SLL `rd`, `rs1`, `rs2`	Logical left shift (zeros are shifted into the lower bits) shift amount is encoded in the lower 5 bits of `rs2`.
SLT `rd`, `x0`, `rs2`	Signed and compare sets `rd` to 1 if `rs2` is not equal to zero, otherwise sets `rd` to zero.
SLTU `rd`, `x0`, `rs2`	Unsigned compare sets `rd` to 1 if `rs2` is not equal to zero, otherwise sets `rd` to zero.
SRL `rd`, `rs1`, `rs2`	Logical right shift (zeros are shifted into the lower bits) shift amount is encoded in the lower 5 bits of `rs2`.
SRA `rd`, `rs1`, `rs2`	Arithmetic right shift, shift amount is encoded in the lower 5 bits of `rs2`.
OR `rd`, `rs1`, `rs2`	Bitwise logical OR.
AND `rd`, `rs1`, `rs2`	Bitwise logical AND.
XOR `rd`, `rs1`, `rs2`	Bitwise logical XOR.

Table 26: R-Type Integer Instruction Description

Below is an example of an ADD instruction.

add x18, x19, x10

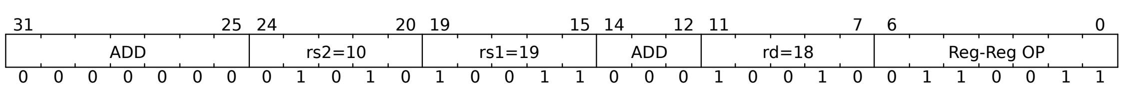

Figure 29: ADD Instruction Example

6.2.2 I-Type Integer Instructions

For I-Type integer instruction, one field is different from R-format. `rs2` and `funct7` are replaced by the 12-bit signed immediate, `imm[11:0]`, which can hold values in range [-2048, +2047]. The immediate is always sign-extended to 32-bits before being used in an arithmetic operation. Bits [31:12] receive the same value as bit 11.

imm			**func3**		**opcode**	**Instruction**
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
00000000	shamnt	rs1	001	rd	0010011	SLLI
00000000	shamnt	rs1	101	rd	0010011	SRLI
01000000	shamnt	rs1	001	rd	0010011	SRAI

Table 27: I-Type Integer Instructions

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI).

Instruction	Description
ADDI	Adds the sign-extended 12-bit immediate to register `rs1`. Arithmetic overflow is ignored and the result is simply the low 64-bits of the result. ADDI `rd`, `rs1`, 0 is used to implement the MV `rd`, `rs1` assembler pseudoinstruction.
SLTI	Set less than immediate. Places the value 1 in register `rd` if register `rs1` is less than the sign extended immediate when both are treated as signed numbers, else 0 is written to `rd`.
SLTIU	Compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 64-bits then treated as an unsigned number). Note: SLTIU `rd`, `rs1`, 1 sets `rd` to 1 if `rs1` equals zero, otherwise sets `rd` to 0 (assembler pseudo instruction SEQZ `rd`, `rs`).
XORI	Bitwise XOR on register `rs1` and the sign-extended 12-bit immediate and place the result in `rd`.
ORI	Bitwise OR on register `rs1` and the sign-extended 12-bit immediate and place the result in `rd`.
ANDI	Bitwise AND on register `rs1` and the sign-extended 12-bit immediate and place the result in `rd`.
SLLI	Shift Left Logical. The operand to be shifted is in `rs1`, and the shift amount is encoded in the lower 5 bits of the I-immediate field.
SRLI	Shift Right Logical. The operand to be shifted is in `rs1`, and the shift amount is encoded in the lower 5 bits of the I-immediate field.
SRAI	Shift Right Arithmetic. The operand to be shifted is in `rs1`, and the shift amount is encoded in the lower 5 bits of the I-immediate field (the original sign bit is copied into the vacated upper bits).

Table 28: I-Type Integer Instruction Description

Shift-by-immediate instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions).

Below is an example of an ADDI instruction.

addi x15, x1, -50

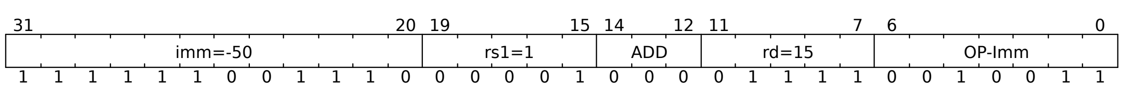

Figure 30: ADDI Instruction Example

6.2.3 I-Type Load Instructions

For I-Type load instructions, a 12-bit signed immediate is added to the base address in register rs1 to form the memory address. In Table 29 below, **funct3** field encodes size and signedness of load data.

imm		func3		opcode	Instruction
imm[11:0]	rs1	000	rd	00000011	LB
imm[11:0]	rs1	001	rd	00000011	LH
imm[11:0]	rs1	010	rd	00000011	LW
imm[11:0]	rs1	100	rd	00000011	LBU
imm[11:0]	rs1	101	rd	00000011	LHU

Table 29: I-Type Load Instructions

Instruction	Description
LB `rd`, `rs1`, `imm`	Load Byte, loads 8 bits (1 byte) and sign-extends to fill destination 32-bit register.
LH `rd`, `rs1`, `imm`	Load Half-Word. Loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register.
LW `rd`, `rs1`, `imm`	Load Word, 32 bits.
LBU `rd`, `rs1`, `imm`	Load Unsigned Byte (8-bit).
LHU `rd`, `rs1`, `imm`	Load Unsigned Half-Word, which zero-extends 16 bits to fill destination 32-bit register.

Table 30: I-Type Load Instruction Description

Below is an example of a LW instruction.

lw x14, 8(x2)

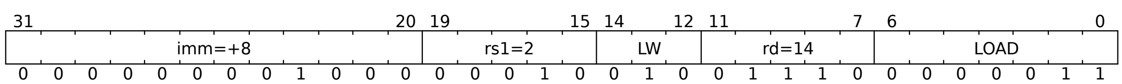

Figure 31: LW Instruction Example

6.2.4 S-Type Store Instructions

Store instructions need to read two registers: `rs1` for base memory address and `rs2` for data to be stored, as well as an immediate offset. The effective byte address is obtained by adding register `rs1` to the sign-extended 12-bit offset. Note that stores don't write a value to the register file, as there is no `rd` register used by the instruction. In RISC-V, the lower 5 bits of immediate are moved to where the `rd` field was in other instructions, and the `rs1`/`rs2` fields are kept in same place. The registers are kept always in the same place because a critical path for all operations includes fetching values from the registers. By always placing the read sources in the same place, the register file can read the registers without hesitation. If the data ends up being unnecessary (e.g., I-Type), it can be ignored.

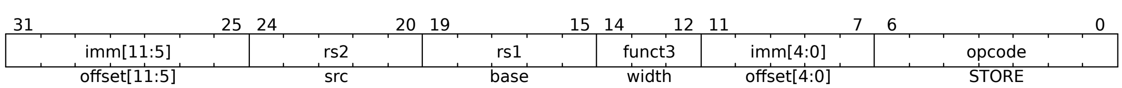

Figure 32: Store Instructions

imm			func3	imm	opcode	Instruction
imm[11:5]	rs2	rs1	000	imm[4:0]	01000011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	01000011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	01000011	SW

Table 31: S-Type Store Instructions

Instruction	Description
SB `rs2`, `imm[11:0](rs1)`	Store 8-bit value from the low bits of register `rs2` to memory.
SH `rs2`, `imm[11:0](rs1)`	Store 16-bit value from the low bits of register `rs2` to memory.
SW `rs2`, `imm[11:0](rs1)`	Store 32-bit value from the low bits of register `rs2` to memory.

Table 32: S-Type Store Instruction Description

Below is an example SW instruction.

sw x14, 8(x2)

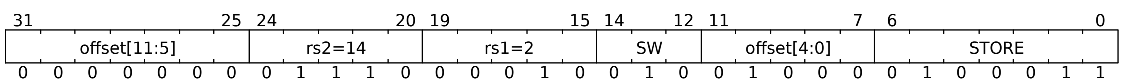

Figure 33: SW Instruction Example

6.2.5 Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register `rd`. The standard software calling convention uses `x1` as the return address register and `x5` as an alternate link register.

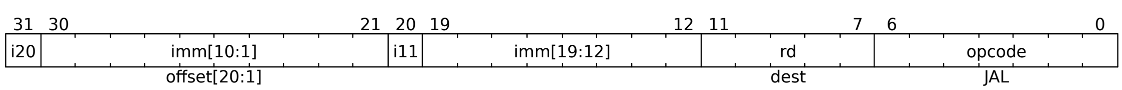

Figure 34: JAL Instruction

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.

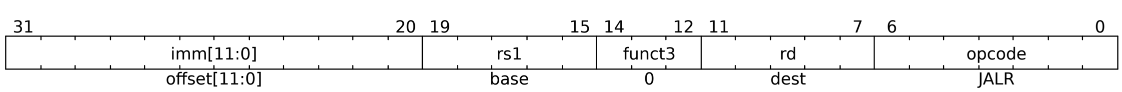

Figure 35: JALR Instruction

Both JAL and JALR instructions will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary.

Instruction	Description
JAL `rd`, `imm[20:1]`	Jump and link
JALR `rd`, `rs1`, `imm[11:0]`	Jump and link register

Table 33: J-Type Instruction Description

6.2.6 Conditional Branches

All branch instructions use the B-Type instruction format. The 12-bit immediate represents values -4096 to +4094 in 2-byte increments. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±4 KiB.

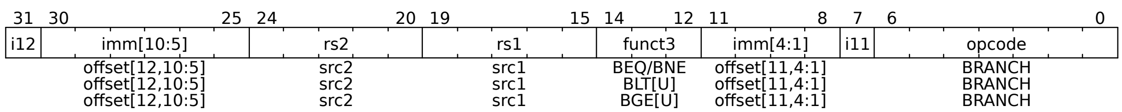

Figure 36: Branch Instructions

imm			func3	imm	opcode	Instruction
imm[12,10:5]	rs2	rs1	000	imm[4:1,11]	110011	BEQ
imm[12,10:5]	rs2	rs1	001	imm[4:1,11]	110011	BNE
imm[12,10:5]	rs2	rs1	100	imm[4:1,11]	110011	BLT
imm[12,10:5]	rs2	rs1	101	imm[4:1,11]	110011	BGE
imm[12,10:5]	rs2	rs1	110	imm[4:1,11]	110011	BLTU
imm[12,10:5]	rs2	rs1	111	imm[4:1,11]	110011	BGEU

Table 34: B-Type Instructions

Instruction	Description
BEQ `rs1`, `rs2`, `imm[12:1]`	Take the branch if registers `rs1` and `rs2` are equal.
BNE `rs1`, `rs2`, `imm[12:1]`	Take the branch if registers `rs1` and `rs2` are unequal.
BLT `rs1`, `rs2`, `imm[12:1]`	Take the branch if `rs1` is less than `rs2`.
BGE `rs1`, `rs2`, `imm[12:1]`	Take the branch if `rs1` is greater than or equal to `rs2`.
BLTU `rs1`, `rs2`, `imm[12:1]`	Take the branch if `rs1` is less than `rs2` (unsigned).
BGEU `rs1`, `rs2`, `imm[12:1]`	Take the branch if `rs1` is greater than or equal to `rs2` (unsigned).

Table 35: B-Type Instruction Description

ISA Base Instruction	Pseudoinstruction	Description
BEQ `rs,x0,offset`	BEQZ `rs,offset`	Take the branch if `rs` is equal to zero.

Table 36: RISC-V Base Instruction to Assembly Pseudoinstruction Example

> **Note**
>
> Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

6.2.7 Upper-Immediate Instructions

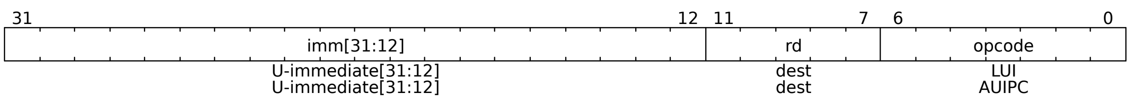

Figure 37: Upper-Immediate Instructions

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register `rd`, filling in the lowest 12 bits with zeros. Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

For example:

LUI x10, 0x87654 # x10 = `0x8765_4000`

ADDI x10, x10, 0x321 # x10 = `0x8765_4321`

AUIPC (add upper immediate to `pc`) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, and adds this offset to the address of the AUIPC instruction, then places the result in register `rd`.

6.2.8 Memory Ordering Operations

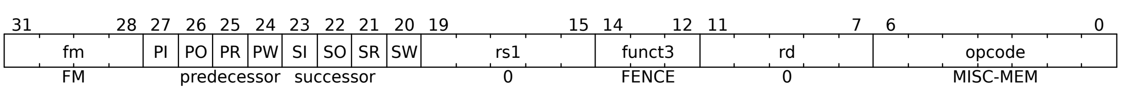

Figure 38: FENCE Instructions

The `FENCE` instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. These operations are discussed further in Section 6.14.

6.2.9 Environment Call and Breakpoints

`SYSTEM` instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions.

6.2.10 NOP Instruction

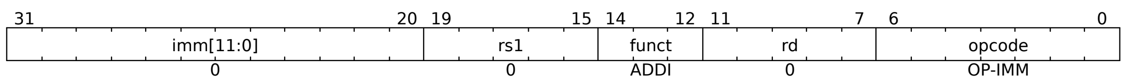

Figure 39: NOP Instructions

The NOP instruction does not change any architecturally visible state, except for advancing the pc and incrementing any applicable performance counters. NOP is encoded as **ADDI x0, x0, 0**.

6.3 M Extension: Multiplication Operations

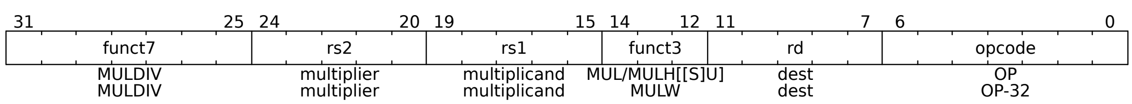

Figure 40: Multiplication Operations

Instruction	Description
MUL `rd`, `rs1`, `rs2`	Multiplication of `rs1` by `rs2` and places the lower 64-bits in the destination register.
MULH `rd`, `rs1`, `rs2`	Multiplication that return the upper 64-bits of the full 2×64-bit product.
MULHU `rd`, `rs1`, `rs2`	Unsigned multiplication that return the upper 64-bits of the full 2×64-bit product.
MULHSU `rd`, `rs1`, `rs2`	Signed rs1 multiple unsigned rs2 that return the upper 64-bits of the full 2×64-bit product.
MULW `rd`, `rs1`, `rs2`	RV64 instruction that multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register.

Table 37: Multiplication Operation Description

Combining MUL and MULH together creates one multiplication operation.

6.3.1 Division Operations

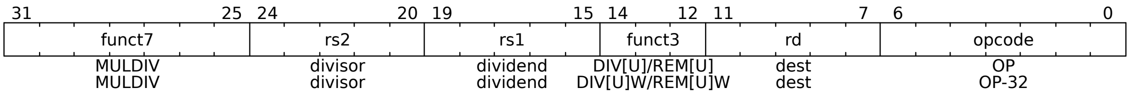

Figure 41: Division Operations

Instruction	Description
DIV `rd`, `rs1`, `rs2`	64-bits by 64-bits signed division of `r1` by `rs2` rounding towards zero.
DIVU `rd`, `rs1`, `rs2`	64-bits by 64-bits unsigned division of `r1` by `rs2` rounding towards zero.
REM `rd`, `rs1`, `rs2`	Remainder of the corresponding division.
REMU `rd`, `rs1`, `rs2`	Unsigned remainder of the corresponding division.
DIVW `rd`, `rs1`, `rs2`	RV64 instruction. Signed divide the lower 32 bits of `rs1` by the lower 32 bits of `rs2`.
DIVUW `rd`, `rs1`, `rs2`	RV64 instruction. Unsigned divide the lower 32 bits of `rs1` by the lower 32 bits of `rs2`.
REMW `rd`, `rs1`, `rs2`	Singed remainder.
REMUW `rd`, `rs1`, `rs2`	Unsigned remainder sign-extend the 32-bit result to 64 bits, including on a divide by zero.
MULDIV `rd`, `rs1`, `rs`	Multiply Divide.

Table 38: Division Operation Description

Combining DIV and REM together creates one division operation.

6.4 A Extension: Atomic Operations

Atomic operations are defined as operations that automatically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space.

6.4.1 Atomic Load-Reserve and Store-Conditional Instructions

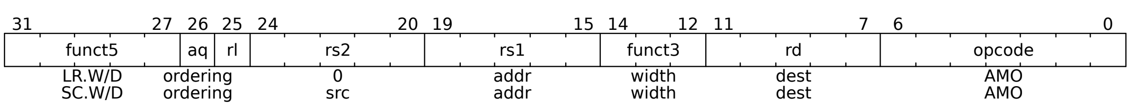

Figure 42: Atomic Operations

Instruction	Description
LR.W	Load Reserve. Loads a word from the address in `rs1`, places the sign-extended value in `rd`, and registers a reservation set—a set of bytes that subsumes the bytes in the addressed word.
SC.W	Store Conditional Conditionally writes a word in `rs2` to the address in `rs1`: the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the SC.W succeeds, the instruction writes the word in `rs2` to memory, and it writes zero to `rd`. If the SC.W fails, the instruction does not write to memory, and it writes a nonzero value to `rd`. Executing an SC.W instruction invalidates any reservation held by this hart.
LR.D	RV64 - Loads doubleword.
SC.D	RV64 - Stores doubleword.

Table 39: Atomic Load-Reserve and Store-Conditional Instruction Description

For RV64, the sign-extended value of LR.W and SC.W is placed in `rd`.

> **Note**
>
> Only cores with data caches support the LR/SC instructions used by the A-Extension. Cores with DTIMs will *NOT*.

6.4.2 Atomic Memory Operations (AMOs)

The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization. These AMO instructions atomically load a data value from the address in `rs1`, place the value into register `rd`, apply a binary operator to the loaded value and the original value in `rs2`, then store the result back to the address in `rs1`.

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
AMOSWAP.W/D	ordering		src	addr	width	dest	AMO
AMOADD.W/D	ordering		src	addr	width	dest	AMO
AMOAND.W/D	ordering		src	addr	width	dest	AMO
AMOOR.W/D	ordering		src	addr	width	dest	AMO
AMOXOR.W/D	ordering		src	addr	width	dest	AMO
AMOMAX[U].W/D	ordering		src	addr	width	dest	AMO
AMOMIN[U].W/D	ordering		src	addr	width	dest	AMO

Figure 43: Atomic Memory Operations

Instruction	Description
AMOSWAPW/D	Word / doubleword swap.
AMOADD.W/D	Word / doubleword add.
AMOAND.W/D	Word / doubleword and.
AMOOR.W/D	Word / doubleword or.
AMOXOR.W/D	Word / doubleword xor.
AMOMIN.W/D	Word / doubleword minimum.
AMOMINU.W/D	Unsigned word / doubleword minimum.
AMOMAX.W/D	Word / doubleword maximum.
AMOMAXU.W/D	Unsigned word / doubleword maximum.

Table 40: Atomic Memory Operation Description

For RV64, 32-bit AMOs always sign-extend the value placed in `rd`.

6.5 F Extension: Single-Precision Floating-Point Instructions

The F Extension implements single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The F Extension adds 32 floating-point registers, `f0`–`f31`, each 32 bits wide, and a floating-point control and status register `fcsr`. Floating-point load and store instructions transfer floating-point values between registers and memory, and instructions to transfer values to and from the integer register file are also provided.

6.5.1 Floating-Point Control and Status Registers

Floating-Point Control and Status Register, `fcsr`, is a RISC-V control and status register (CSR). The register selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags.

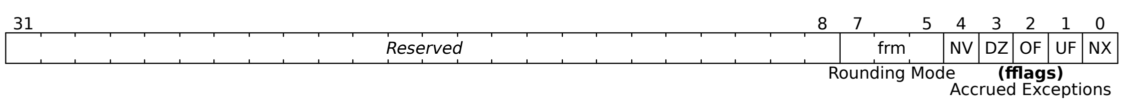

Figure 44: Floating-Point Control and Status Register

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Table 41: Accrued Exception Flags

The `fcsr` register can be read and written with the FRCSR and FSCSR instructions. The FRRM instruction reads the Rounding Mode field `frm`. FSRM swaps the value in `frm` with an integer register. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field `fflags`.

6.5.2 Rounding Modes

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. A value of 111 in the instruction's rm field selects the dynamic rounding mode held in `frm`. If `frm` is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will raise an illegal instruction exception. Some instructions, including widening conversions, have the `rm` field, but are nevertheless unaffected by the rounding mode. Software should set their `rm` field to RNE (000).

Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even.
001	RTZ	Round towards Zero.
010	RDN	Round Down (towards - ∞).
011	RUP	Round Up (towards + ∞).
100	RMM	Round to Nearest, ties to Max Magnitude.
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111	DYN	In instruction's `rm` field, selects dynamic rounding mode; In Rounding Mode register, *Invalid*.

Table 42: Floating-Point Rounding Modes

6.5.3 Single-Precision Floating-Point Load and Store Instructions

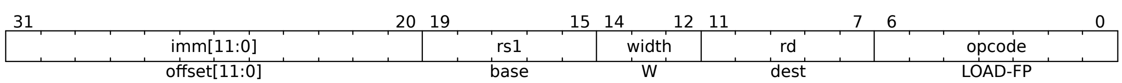

Figure 45: Single-Precision FP Load Instruction

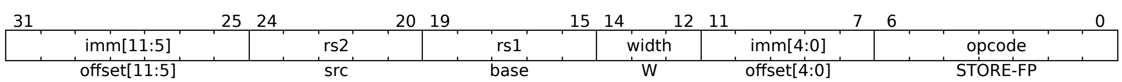

Figure 46: Single-Precision FP Store Instruction

Instruction	Operation	Description
FLW `rd,rs1,imm`	`f[rd] = M[x[rs1] + sext(offset)][31:0]`	Loads a single-precision floating-point value from memory into floating-point register `rd`.
FSW `imm,rs1,rs2`	`M[x[rs1] + sext(offset)] = f[rs2][31:0]`	Stores a single-precision value from floating-point register `rs2` to memory.

Table 43: Single-Precision FP Load and Store Instructions Description

6.5.4 Single-Precision Floating-Point Computational Instructions

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FSQRT	S	0	src	RM	dest	OP-FP
FMIN-MAX	S	src2	src1	MIN/MAX	dest	OP-FP

Figure 47: Single-Precision FP Computational Instructions

31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	fmt	rs2	rs1	rm	rd	opcode
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

Figure 48: Single-Precision FP Fused Computational Instructions

Instruction	Operation	Description
FADD.S `rd,rs1,rs2`	`f[rd] = f[rs1] + f[rs2]`	Single-precision floating-point addition.
FSUB.S `rd,rs1,rs2`	`f[rd] = f[rs1] - f[rs2]`	Single-precision floating-point subtraction.
FMUL.S `rd,rs1,rs2`	`f[rd] = f[rs1] × f[rs2]`	Single-precision floating-point multiplication.
FDIV.S `rd,rs1,rs2`	`f[rd] = f[rs1] ÷ f[rs2]`	Single-precision floating-point division.
FSQRT.S `rd,rs1`	`f[rd] = √f[rs1]`	Single-precision floating-point square root.
FMIN.S `rd,rs1,rs2`	`f[rd] = min(f[rs1], f[rs2])`	Single-precision floating-point minimum-number.
FMAX.S `rd,rs1,rs2`	`f[rd] = max(f[rs1], f[rs2])`	Single-precision floating-point maximum-number.
FMADD.S `rd,rs1,rs2,rs3`	`f[rd] = (f[rs1] × f[rs2]) + f[rs3]`	Single-precision floating-point multiply and add.
FMSUB.S `rd,rs1,rs2,rs3`	`f[rd] = (f[rs1] × f[rs2]) - f[rs3]`	Single-precision floating-point multiply and subtract.
FNMADD.S `rd,rs1,rs2,rs3`	`f[rd]= -(f[rs1] × f[rs2]) + f[rs3]`	Single-precision floating-point multiply, negate, and add.
FNMSUB.S `rd,rs1,rs2,rs3`	`f[rd]= -(f[rs1] × f[rs2]) - f[rs3]`	Single-precision floating-point multiply, negate, and subtract.

Table 44: Single-Precision FP Computational Instructions Description

6.5.5 Single-Precision Floating-Point Conversion and Move Instructions

Single-Precision Floating-Point Conversion Instructions

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
FCVT.*int*.S	S	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.S.*int*	S	W[U]/L[U]	src	RM	dest	OP-FP

Figure 49: Single-Precision FP to Integer and Integer to FP Conversion Instructions

Instruction	Operation	Description
FCVT.W.S rd,rs1	x[rd] = sext(s32$_{f32}$(f[rs1]))	Converts a single-precision floating-point number to a signed 32-bit integer. Sign-extends the 32-bit result to the destination register width.
FCVT.S.W rd,rs1	f[rd] = f32$_{s32}$(x[rs1])	Converts a signed 32-bit integer to a single-precision floating-point number.
FCVT.WU.S rd,rs1	x[rd] = sext(u32$_{f32}$(f[rs1]))	Converts a single-precision floating-point number to an unsigned 32-bit integer. Sign-extends the 32-bit result to the destination register width.
FCVT.S.WU rd,rs1	f[rd] = f32$_{u32}$(x[rs1])	Converts an unsigned 32-bit integer to a single-precision floating-point number.
FCVT.L.S rd,rs1	x[rd] = s64$_{f32}$(f[rs1])	Converts a single-precision floating-point number to a signed 64-bit integer.
FCVT.S.L rd,rs1	f[rd] = f32$_{s64}$(x[rs1])	Converts a signed 64-bit integer to a single-precision floating-point number.
FCVT.LU.S rd,rs1	x[rd] = u64$_{f32}$(f[rs1])	Converts a single-precision floating-point number to an unsigned 64-bit integer.
FCVT.S.LU rd,rs1	f[rd] = f32$_{u64}$(x[rs1])	Converts an unsigned 64-bit integer to a single-precision floating-point number.

Table 45: Single-Precision FP Conversion Instructions Description

If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set.

Single-Precision Floating-Point-to-Floating-Point Sign-Injection Instructions

The floating-point-to-floating-point sign-injection instructions produce a result that takes all bits except the sign bit from rs1. The sign-injection instructions provide floating-point MV, ABS and NEG.

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
FSGNJ	S	src2	src1	J[N]/JX	dest	OP-FP

Figure 50: Single-Precision FP to FP Sign-Injection Instructions

Instruction	Operation	Description
FSGNJ.S `rd,rs1,rs2`	`f[rd] = {f[rs2][31], f[rs1][30:0]}`	Produces a result that takes all bits except the sign bit from `rs1`. The result's sign bit is `rs2`'s sign bit.
FSGNJN.S `rd,rs1,rs2`	`f[rd] = {~f[rs2][31], f[rs1][30:0]}`	Produces a result that takes all bits except the sign bit from `rs1`. The result's sign bit is the opposite of `rs2`'s sign bit.
FSGNJX.S `rd,rs1,rs2`	`f[rd] = {f[rs1][31] ^ f[rs2][31], f[rs1][30:0]}`	Produces a result that takes all bits except the sign bit from `rs1`. The sign bit is the XOR of the sign bits of `rs1` and `rs2`.

Table 46: Single-Precision FP to FP Sign-Injection Instructions Description

ISA Base Instruction	Pseudoinstruction	Description
FSGNJ.S `rx,ry,ry`	FMV.S `rx,ry`	Moves `ry` to `rx`.
FSGNJN.S `rx,ry,ry`	FNEG.S `rx,ry`	Moves the negation of `ry` to `rx`.
FSGNJX.S `rx,ry,ry`	FABS.S `rx,ry`	Moves the absolute value of `ry` to `rx`.

Table 47: RISC-V Base Instruction to Assembly Pseudoinstruction Example

Single-Precision Floating-Point Move Instructions

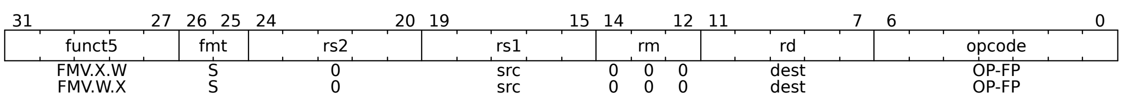

Figure 51: Single-Precision FP Move Instructions

Instruction	Operation	Description
FMV.X.W `rd,rs1`	`x[rd] = sext(f[rs1][31:0])`	Moves the single-precision value in floating-point register `rs1` represented in IEEE 754-2008 encoding to the lower 32 bits of integer register `rd`. The higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.
FMV.W.X `rd,rs1`	`f[rd] = x[rs1][31:0]`	Moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register `rs1` to the floating-point register `rd`.

Table 48: Single-Precision FP Move Instructions Description

6.5.6 Single-Precision Floating-Point Compare Instructions

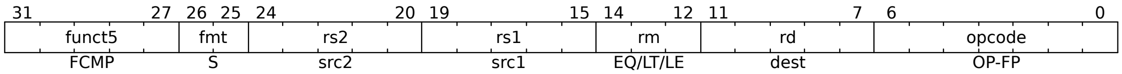

Figure 52: Single-Precision FP Compare Instructions

Instruction	Operation	Description
FEQ.S `rd,rs1,rs2`	`x[rd] = f[rs1] == f[rs2]`	Writes 1 to the integer register `rd` if `rs1` is equal to `rs2`, 0 otherwise. Performs a quiet comparison; only sets the invalid operation exception flag if either input is a signaling NaN.
FLT.S `rd,rs1,rs2`	`x[rd] = f[rs1] < f[rs2]`	Writes 1 to the integer register `rd` if `rs1` less then `rs2`, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN.
FLE.S `rd,rs1,rs2`	`x[rd] = f[rs1] ≤ f[rs2]`	Writes 1 to the integer register `rd` if `rs1` less than or equal to `rs2`, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN.

Table 49: Single-Precision FP Compare Instructions Description

Single-Precision Floating-Point Classify Instruction

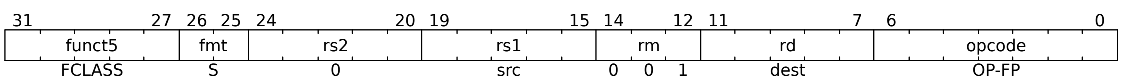

Figure 53: Single-Precision FP Classify Instruction

Instruction	Operation	Description
FCLASS.S `rd,rs1`	`x[rd] =` $\texttt{classify}_S$`(f[rs1])`	Examines the value in floating-point register `rs1` and writes to integer register `rd` a 10-bit mask that indicates the class of the floating-point number.

Table 50: Single-Precision FP Classify Instruction Description

`rd` bit	Meaning
0	`rs1` is -∞
1	`rs1` is negative normal number
2	`rs1` is a negative subnormal number
3	`rs1` is -0
4	`rs1` is +0
5	`rs1` is a positive subnormal number
6	`rs1` is a positive normal number
7	`rs1` is +∞
8	`rs1` is a signaling NaN
9	`rs1` is a quiet NaN

Table 51: Floating-Point Number Classes

6.6 D Extension: Double-Precision Floating-Point Instructions

The D extension widens the 32 floating-point registers, `f0–f31`, to 64 bits. The f registers can now hold either 32-bit or 64-bit floating-point values. When multiple floating-point precisions are supported, then valid values of narrower n-bit types, n < FLEN, are represented in the lower n bits of an FLEN-bit. Any operation that writes a narrower result to an f register must write all 1s to the uppermost FLEN-n bits to yield a legal NaN-boxed value. Floating-point n-bit transfer operations move external values held in IEEE standard formats into and out of the f registers, and comprise floating-point loads and stores and floating-point move instructions.

6.6.1 Double-Precision Floating-Point Load and Store Instructions

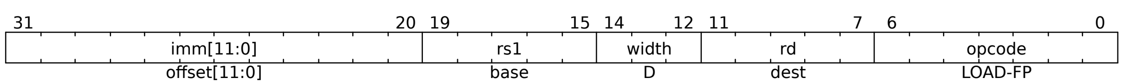

Figure 54: Double-Precision FP Load Instruction

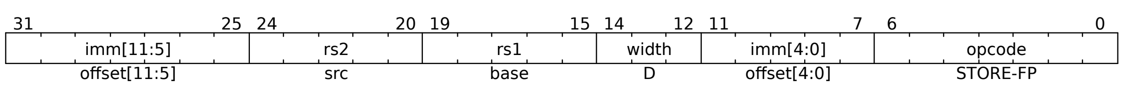

Figure 55: Double-Precision FP Store Instruction

Instruction	Operation	Description
FLD `rd,rs1,imm`	`f[rd] = M[x[rs1] + sext(offset)][63:0]`	Loads a double-precision floating-point value from memory into floating-point register `rd`.
FSD `imm,rs1,rs2`	`M[x[rs1] + sext(offset)] = f[rs2][63:0]`	Stores a double-precision value from the floating-point register `rs2` to memory.

Table 52: Double-Precision FP Load and Store Instructions Description

FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64. These instructions do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

6.6.2 Double-Precision Floating-Point Computational Instructions

The double-precision floating-point computational instructions are defined analogously to their single-precision counterparts, but operate on double-precision operands and produce double-precision results.

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
FADD/FSUB	D	src2	src1	RM	dest	OP-FP
FMUL/FDIV	D	src2	src1	RM	dest	OP-FP
FMIN-MAX	D	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	D	0	src	RM	dest	OP-FP
						OP-FP

Figure 56: Double-Precision FP Computational Instructions

31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	fmt	rs2	rs1	rm	rd	opcode
src3	D	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

Figure 57: Double-Precision FP Fused Computational Instructions

Instruction	Operation	Description
FADD.D `rd,rs1,rs2`	`f[rd] = f[rs1] + f[rs2]`	Double-precision floating-point addition.
FSUB.D `rd,rs1,rs2`	`f[rd] = f[rs1] - f[rs2]`	Double-precision floating-point subtraction.
FMUL.D `rd,rs1,rs2`	`f[rd] = f[rs1] × f[rs2]`	Double-precision floating-point multiplication.
FDIV.D `rd,rs1,rs2`	`f[rd] = f[rs1] ÷ f[rs2]`	Double-precision floating-point division.
FSQRT.D `rd,rs1`	`f[rd] = √f[rs1]`	Double-precision floating-point square root.
FMIN.D `rd,rs1,rs2`	`f[rd] = min(f[rs1], f[rs2])`	Double-precision floating-point minimum-number.
FMAX.D `rd,rs1,rs2`	`f[rd] = max(f[rs1], f[rs2])`	Double-precision floating-point maximum-number.
FMADD.D `rd,rs1,rs2,rs3`	`f[rd] = (f[rs1] × f[rs2]) + f[rs3]`	Double-precision floating-point multiply and add.
FMSUB.D `rd,rs1,rs2,rs3`	`f[rd] = (f[rs1] × f[rs2]) - f[rs3]`	Double-precision floating-point multiply and subtract.
FNMADD.D `rd,rs1,rs2,rs3`	`f[rd] = -(f[rs1] × f[rs2]) + f[rs3]`	Double-precision floating-point multiply, negate, and add.
FNMSUB.D `rd,rs1,rs2,rs3`	`f[rd] = -(f[rs1] × f[rs2]) - f[rs3]`	Double-precision floating-point multiply, negate, and subtract.

Table 53: Double-Precision FP Computational Instructions Description

6.6.3 Double-Precision Floating-Point Conversion and Move Instructions

Double-Precision Floating-Point Conversion Instructions

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
FCVT.*int*.D	D	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.D.*int*	D	W[U]/L[U]	src	RM	dest	OP-FP

Figure 58: Double-Precision FP to Integer and Integer to FP Conversion Instructions

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
FCVT.S.D	S	D	src	RM	dest	OP-FP
FCVT.D.S	D	S	src	RM	dest	OP-FP

Figure 59: Double-Precision to Single-Precision and Single-Precision to Double-Precision FP Conversion Instructions

Instruction	Operation	Description
FCVT.W.D `rd,rs1`	`x[rd] = sext(s32`$_{f64}$`(f[rs1]))`	Converts a double-precision floating-point number to a signed 32-bit integer. Sign-extends the 32-bit result to the destination register width.
FCVT.D.W `rd,rs1`	`f[rd] = f64`$_{s32}$`(x[rs1])`	Converts a signed 32-bit integer to a double-precision floating-point number. Always produces an exact result and is unaffected by rounding mode.
FCVT.WU.D `rd,rs1`	`x[rd] = sext(u32`$_{f64}$`(f[rs1]))`	Converts a double precision floating-point number to an unsigned 32-bit integer. Sign-extends the 32-bit result to the destination register width.
FCVT.D.WU `rd,rs1`	`f[rd] = f64`$_{u32}$`(x[rs1])`	Converts an unsigned 32-bit integer to a double-precision floating-point number. Always produces an exact result and is unaffected by rounding mode.
FCVT.L.D `rd,rs1`	`x[rd] = s64`$_{f64}$`(f[rs1])`	Converts a double-precision floating-point number to a signed 64-bit integer.
FCVT.D.L `rd,rs1`	`f[rd] = f64`$_{s64}$`(x[rs1])`	Converts a signed 64-bit integer to a double-precision floating-point number.
FCVT.LU.D `rd,rs1`	`x[rd] = u64`$_{f64}$`(f[rs1])`	Converts a double-precision floating-point number to an unsigned 64-bit integer.
FCVT.D.LU `rd,rs1`	`f[rd] = f64`$_{u64}$`(x[rs1])`	Converts an unsigned 64-bit integer to a double-precision floating-point number.
FCVT.S.D `rd,rs1`	`f[rd] = f32`$_{f64}$`(f[rs1])`	Converts a double-precision floating-point number to a single-precision floating-point number.
FCVT.D.S `rd,rs1`	`f[rd] = f64`$_{f32}$`(f[rs1])`	Converts a single-precision floating-point number to a double-precision floating-point number.

Table 54: Double-Precision FP Conversion Instructions Description

Double-Precision Floating-Point-to-Floating-Point Sign-Injection Instructions

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
FSGNJ	D	src2	src1	J[N]/JX	dest	OP-FP

Figure 60: Double-Precision FP to FP Sign-Injection Instructions

Instruction	Operation	Description
FSGNJ.D `rd,rs1,rs2`	`f[rd] = {f[rs2][63], f[rs1][62:0]}`	Produces a result that takes all bits except the sign bit from `rs1`. The result's sign bit is `rs2`'s sign bit.
FSGNJN.D `rd,rs1,rs2`	`f[rd] = {~f[rs2][63], f[rs1][62:0]}`	Produces a result that takes all bits except the sign bit from `rs1`. The result's sign bit is the opposite of `rs2`'s sign bit.
FSGNJX.D `rd,rs1,rs2`	`f[rd] = {f[rs1][63] ^ f[rs2][63], f[rs1][62:0]}`	Produces a result that takes all bits except the sign bit from `rs1`. The sign bit is the XOR of the sign bits of `rs1` and `rs2`.

Table 55: Double-Precision FP to FP Sign-Injection Instructions Description

ISA Base Instruction	Pseudoinstruction	Description
FSGNJ.D `rx,ry,ry`	FMV.D `rx,ry`	Moves `ry` to `rx`.
FSGNJN.D `rx,ry,ry`	FNEG.D `rx,ry`	Moves the negation of `ry` to `rx`.
FSGNJX.D `rx,ry,ry`	FABS.D `rx,ry`	Moves the absolute value of `ry` to `rx`.

Table 56: RISC-V Base Instruction to Assembly Pseudoinstruction Example

Double-Precision Floating-Point Move Instructions

The RV64 architecture provides instructions to move bit patterns between the floating-point and integer registers.

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
FMV.X.D	D	0	src	0 0 0	dest	OP-FP
FMV.D.X	D	0	src	0 0 0	dest	OP-FP

Figure 61: Double-Precision FP Move Instructions

Instruction	Operation	Description
FMV.X.D `rd,rs1`	`x[rd] = f[rs1][63:0]`	Moves the double-precision value in floating-point register `rs1` to a representation in IEEE 754-2008 standard encoding in integer register `rd`.
FMV.D.X `rd,rs1`	`f[rd] = x[rs1][63:0]`	Moves the double-precision value encoded in IEEE 754-2008 standard encoding from the integer register `rs1` to the floating-point register `rd`.

Table 57: Double-Precision FP Move Instructions Description

FMV.X.D and FMV.D.X do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

6.6.4 Double-Precision Floating-Point Compare Instructions

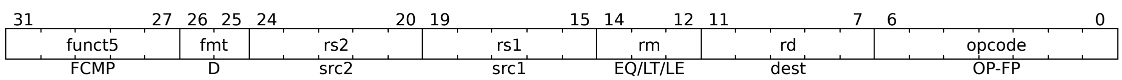

Figure 62: Double-Precision FP Compare Instructions

Instruction	Operation	Description
FEQ.D `rd,rs1,rs2`	`x[rd] = f[rs1] == f[rs2]`	Writes 1 to the integer register `rd` if `rs1` is equal to `rs2`, 0 otherwise. Performs a quiet comparison; only sets the invalid operation exception flag if either input is a signaling NaN.
FLT.D `rd,rs1,rs2`	`x[rd] = f[rs1] < f[rs2]`	Writes 1 to the integer register `rd` if `rs1` less than `rs2`, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN.
FLE.D `rd,rs1,rs2`	`x[rd] = f[rs1] ≤ f[rs2]`	Writes 1 to the integer register `rd` if `rs1` less than or equal to `rs2`, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN.

Table 58: Double-Precision FP Compare Instructions Description

6.6.5 Double-Precision Floating-Point Classify Instruction

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
FCLASS	D	0	src	0 0 1	dest	OP-FP

Figure 63: Double-Precision FP Classify Instruction

Instruction	Operation	Description
FCLASS.D `rd,rs1`	`x[rd] = classify`$_d$`(f[rs1])`	Examines the value in floating-point register `rs1` and writes to integer register `rd` a 10-bit mask that indicates the class of the floating-point number.

Table 59: Double-Precision FP Classify Instruction Description

6.7 C Extension: Compressed Instructions

The C Extension reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs (RV32, RV64, RV128), and we use the generic term "RVC" to cover any of these. Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction. The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions. It is important to note that the C extension is not designed to be a stand-alone ISA, and is meant to be used alongside a base ISA. The compressed 16-bit instruction format is designed around the assumption that `x1` is the return address register and `x2` is the stack pointer.

6.7.1 Compressed 16-bit Instruction Formats

15–12	11–7	6–2	1–0
funct4	rd/rs1	rs2	op

Figure 64: CR Format - Register

15–13	12	11–7	6–2	1–0
funct3	imm	rd/rs1	imm	op

Figure 65: CI Format - Immediate

15–13	12–7	6–2	1–0
funct3	imm	rs2	op

Figure 66: CSS Format - Stack-relative Store

15–13	12–5	4–2	1–0
funct3	imm	rd´	op

Figure 67: CIW Format - Wide Immediate

15–13	12–10	9–7	6–5	4–2	1–0
funct3	imm	rs1´	imm	rd´	op

Figure 68: CL Format - Load

15–13	12–10	9–7	6–5	4–2	1–0
funct3	imm	rs1´	imm	rs2´	op

Figure 69: CS Format - Store

15–10	9–7	6–5	4–2	1–0
funct6	rd´/ rs1´	funct2	rs2´	op

Figure 70: CA Format - Arithmetic

15–13	12–10	9–7	6–2	1–0
funct3	offset	rs1´	offset`	op

Figure 71: CJ Format - Jump

6.7.2 Stack-Pointed-Based Loads and Stores

The compressed load instructions are expressed in CI format.

15–13	12	11–7	6–2	1–0
funct3	imm	rd	imm	op
C.LWSP	offset[5]	dest != 0	offset[4:2|7:6]	C2
C.LDSP	offset[5]	dest != 0	offset[4:3|8:6]	C2
C.LQSP	offset[5]	dest != 0	offset[4|9:6]	C2
C.FLWSP	offset[5]	dest	offset[4:2|7:6]	C2
C.FLDSP	offset[5]	dest	offset[4:3|8:6]	C2

Figure 72: Stack-Pointed-Based Loads

Instruction	Description
C.LWSP	Loads a 32-bit value from memory into register `rd`.
C.LDSP	RV64C Instruction which loads a 64-bit value from memory into register `rd`.
C.LQSP	RV128C loads a 128-bit value from memory into register `rd`.
C.FLWSP	RV32FC Instruction that loads a single-precision floating-point value from memory into floating-point register `rd`.
C.FLDSP	RV32DC/RV64DC Instruction that loads a double-precision floating-point value from memory into floating-point register `rd`.

Table 60: Stack-Pointed-Based Load Instruction Description

The compressed store instructions are expressed in CSS format.

15–13	12–7	6–2	1–0
funct3	imm	rs2	op
C.SWSP	offset[5:2|7:6]	src	C2
C.SDSP	offset[5:3|8:6]	src	C2
C.SQSP	offset[5:4|9:6]	src	C2
C.FSWSP	offset[5:2|7:6]	src	C2
C.FSDSP	offset[5:3|8:6]	src	C2

Figure 73: Stack-Pointed-Based Stores

Instruction	Description
C.LWSP	Loads a 32-bit value from memory into register `rd`.
C.SWSP	Stores a 32-bit value in register `rs2` to memory.
C.SDSP	RV64C/RV128C instruction that stores a 64-bit value in register `rs2` to memory.
C.SQSP	RV128C instruction that stores a 128-bit value in register `rs2` to memory.
C.FSWSP	RV32FC instruction that stores a single-precision floating-point value in floating-point register `rs2` to memory.
C.FSDSP	RV32DC/RV64DC instruction that stores a double-precision floating-point value in floating-point register `rs2` to memory.

Table 61: Stack-Pointed-Based Store Instruction Description

6.7.3 Register-Based Loads and Stores

The compressed register-based load instructions are expressed in CL format.

15–13	12–10	9–7	6–5	4–2	1–0
funct3	imm	rs1′	imm	rd′	op
C.LW	offset[5:3]	base	offset[2|6]	dest	C0
C.LD	offset[5:3]	base	offset[7:6]	dest	C0
C.LQ	offset[5|4|8]	base	offset[7:6]	dest	C0
C.FLW	offset[5:3]	base	offset[2|6]	dest	C0
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0

Figure 74: Register-Based Loads

Instruction	Description
C.LW	Loads a 32-bit value from memory into register rd.
C.LD	RV64C/RV128C-only instruction that loads a 64-bit value from memory into register `rd`.
C.LQ	RV128C-only instruction that loads a 128-bit value from memory into register `rd`.
C.FLW	RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register `rd`.
C.FLD	RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register `rd`.

Table 62: Register-Based Load Instruction Description

The compressed register-based store instructions are expressed in CS format.

15–13	12–10	9–7	6–5	4–2	1–0
funct3	imm	rs1′	imm	rs2′	op
C.SW	offset[5:3]	base	offset[2|6]	src	C0
C.SD	offset[5:3]	base	offset[7:6]	src	C0
C.SQ	offset[5|4|8]	base	offset[7:6]	src	C0
C.FSW	offset[5:3]	base	offset[2|6]	src	C0
C.FSD	offset[5:3]	base	offset[7:6]	src	C0

Figure 75: Register-Based Stores

Instruction	Description
C.SW	Stores a 32-bit value in register `rs2` to memory.
C.SD	RV64C/RV128C instruction that stores a 64-bit value in register `rs2` to memory.
C.SQ	RV128C instruction that stores a 128-bit value in register `rs2` to memory.
C.FSW	RV32FC instruction that stores a single-precision floating-point value in floating-point register `rs2` to memory.
C.FSD	RV32DC/RV64DC instruction that stores a double-precision floating-point value in floating-point register `rs2` to memory.

Table 63: Register-Based Store Instruction Description

6.7.4 Control Transfer Instructions

RVC provides unconditional jump instructions and conditional branch instructions.

The unconditional jump instructions are expressed in CJ format.

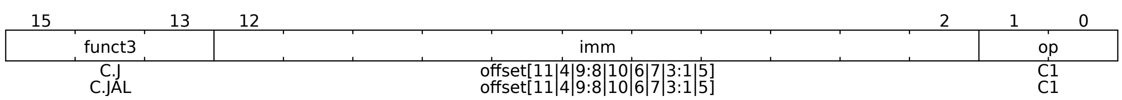

Figure 76: Unconditional Jump Instructions

Instruction	Description
C.J	Unconditional control transfer.
C.JAL	RV32C instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (pc+2) to the link register, `x1`.

Table 64: Unconditional Jump Instruction Description

The unconditional control transfer instructions are expressed in CR format.

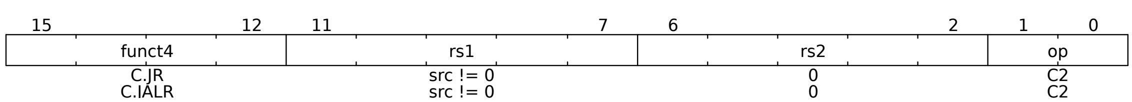

Figure 77: Unconditional Control Transfer Instructions

Instruction	Description
C.JR	Performs an unconditional control transfer to the address in register `rs1`.
C.JALR	Performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (pc+2) to the link register, `x1`.

Table 65: Unconditional Control Transfer Instruction Description

The conditional control transfer instructions are expressed in CB format.

15–13	12–10	9–7	6–2	1–0
funct3	imm	rs1′	imm	op
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1

Figure 78: Conditional Control Transfer Instructions

Instruction	Description
C.BEQZ	Conditional control transfers. Takes the branch if the value in register `rs1'` is zero.
C.BNEZ	Conditional control transfers. Takes the branch if `rs1'` contains a nonzero value.

Table 66: Conditional Control Transfer Instruction Description

6.7.5 Integer Computational Instructions

Integer Constant-Generation Instructions

15–13	12	11–7	6–2	1–0
funct3	imm[5]	rd	imm	op
C.LI	imm[5]	dest != 0	imm[4:0]	C1
CI.LUI	nzimm[17]	dest != {0,2}	imm[16:12]	C1

Figure 79: Integer Constant-Generation Instructions

Instruction	Description
C.LI	Loads the sign-extended 6-bit immediate, `imm`, into register `rd`.
C.LUI	Loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination

Table 67: Integer Constant-Generation Instruction Description

Integer Register-Immediate Operations

15–13	12	11–7	6–2	1–0
funct3	imm[5]	rd/rs1	imm[4:0]	op
C.ADDI	nzimm[5]	dest != 0	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest != 0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

Figure 80: Integer Register-Immediate Operations

Instruction	Description
C.ADDI	Adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to `rd`.
C.ADDIW	RV64C/RV128C instruction that performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits.
C.ADDI16SP	Adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (`sp`=x2), where the immediate is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues.

Table 68: Integer Register-Immediate Operation Description

15–13	12–5	4–2	1–0
funct3	imm	rd´	op
C.ADDI4SPN	nzuimm[5:4|9:6|2|3]	dest	C0

Figure 81: Integer Register-Immediate Operations (cont.)

Instruction	Description
C.ADDI4SPN	Adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, `x2`, and writes the result to `rd′`.

Table 69: Integer Register-Immediate Operation Description (cont.)

15–13	12	11–7	6–2	1–0
funct3	shamt[5]	rd/rs1	shamt[4:0]	op
C.SLLI	shamt[5]	dest != 0	shamt[4:0]	C2

Figure 82: Integer Register-Immediate Operations (cont.)

Instruction	Description
C.SLLI	Performs a logical left shift of the value in register `rd` then writes the result to `rd`. The shift amount is encoded in the `shamt` field.

Table 70: Integer Register-Immediate Operation Description (cont.)

15–13	12	11–10	9–7	6–2	1–0
funct3	shamt[5]	funct2	rd´/rs1´	shamt[4:0]	op
C.SRLI	shamt[5]	C.SRLI	dest	shamt[4:0]	C1
C.SRAI	shamt[5]	C.SRAI	dest	shamt[4:0]	C1

Figure 83: Integer Register-Immediate Operations (cont.)

Instruction	Description
C.SRLI	Logical right shift of the value in register `rd′` then writes the result to `rd′`. The shift amount is encoded in the `shamt` field.
C.SRAI	Arithmetic right shift of the value in register `rd′` then writes the result to `rd′`. The shift amount is encoded in the `shamt` field.

Table 71: Integer Register-Immediate Operation Description (cont.)

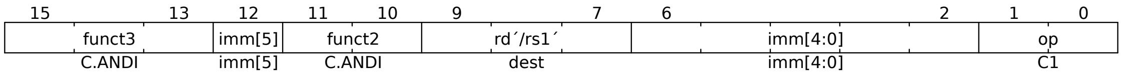

Figure 84: Integer Register-Immediate Operations (cont.)

Instruction	Description
C.ANDI	Computes the bitwise AND of the value in register `rd`′ and the sign-extended 6-bit immediate, then writes the result to `rd`′.

Table 72: Integer Register-Immediate Operation Description (cont.)

Integer Register-Register Operations

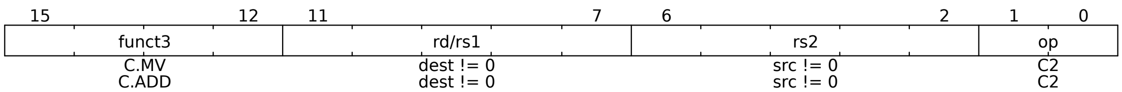

Figure 85: Integer Register-Register Operations

Instruction	Description
C.MV	Copies the value in register `rs2` into register `rd`.
C.ADD	Adds the values in registers `rd` and `rs2` and writes the result to register rd.

Table 73: Integer Register-Register Operation Description

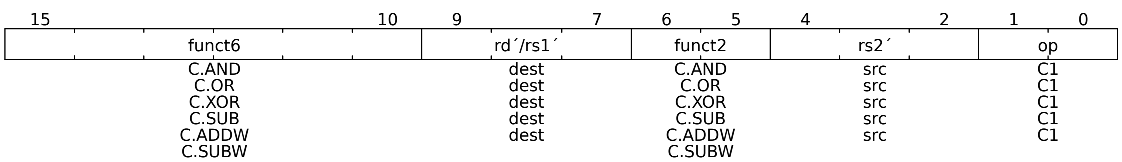

Figure 86: Integer Register-Register Operations (cont.)

Instruction	Description
C.AND	Computes the bitwise AND of the values in registers `rd`′ and `rs2`′.
C.OR	Computes the bitwise OR of the values in registers `rd`′ and `rs2`′.
C.XOR	Computes the bitwise XOR of the values in registers `rd`′ and `r2`′.
C.SUB	Subtracts the value in register `rs2`′ from the value in register `rd`′.
C.ADDW	RV64C/RV128C-only instruction that adds the values in registers `rd`′ and `rs2`′, then sign-extends the lower 32 bits of the sum before writing the result to register `rd`.
C.SUBW	RV64C/RV128C-only instruction that subtracts the value in register `rs2`′ from the value in register `rd`′, then sign-extends the lower 32 bits of the difference before writing the result to register `rd`.

Table 74: Integer Register-Register Operation Description (cont.)

Defined Illegal Instruction

A 16-bit instruction with all bits zero is permanently reserved as an illegal instruction.

15–13	12	11–7	6–2	1–0
0	0	0	0	0
0	0	0	0	0

Figure 87: Defined Illegal Instruction

6.8 Zba Extension: Address Calculation Instructions

The Zba instructions are used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and 64-sized indices; that is, a shifted index is added to a base address.

6.8.1 Address Calculation Instructions

31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
SH1ADD			SH1ADD		OP
SH2ADD			SH2ADD		OP
SH3ADD			SH3ADD		OP

Figure 88: Address Calculation Instructions

31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
SH1ADD.UW			SH1ADD.UW		OP-32
SH2ADD.UW			SH2ADD.UW		OP-32
SH3ADD.UW			SH3ADD.UW		OP-32

Figure 89: Address Calculation Instructions (cont.)

Instruction	**Description**
SH1ADD `rd,rs1,rs2`	Shifts `rs1` by 1 bit, then adds the result to `rs2`
SH2ADD `rd,rs1,rs2`	Shifts `rs1` by 2 bits, then adds the result to `rs2`
SH3ADD `rd,rs1,rs2`	Shifts `rs1` by 3 bits, then adds the result to `rs2`
SH1ADD.UW `rd,rs1,rs2`	Performs an 64-wide addition of `rs2`, and the unsigned value formed by extracting the least-significant word of `rs1` and shifting it left by 1 bit
SH2ADD.UW `rd,rs1,rs2`	Performs an 64-wide addition of `rs2`, and the unsigned value formed by extracting the least-significant word of `rs1` and shifting it left by 2 bits
SH3ADD.UW `rd,rs1,rs2`	Performs an 64-wide addition of `rs2`, and the unsigned value formed by extracting the least-significant word of `rs1` and shifting it left by 3 bits

Table 75: Address Calculation Instructions Description

6.8.2 Add/Shift with Prefix Zero-Extend Instructions

31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
ADD.UW	src	addr	ADD.UW	dest	OP-32

Figure 90: Add with Prefix Zero-Extend Instruction

31–26	25–20	19–15	14–12	11–7	6–0
imm	shamt	rs1	funct3	rd	opcode
SLLI.UW			SLLI.UW		OP-IMM-32

Figure 91: Shift with Prefix Zero-Extend Instruction

Instruction	Description
ADD.UW `rd,rs1,rs2`	Performs a 64-wide addition between `rs2` and the zero-extended least-significant word of `rs1`
SLLI.UW `rd,rs1,shamt`	Takes the least-significant word of `rs1`, zero-extends it, and shifts it left by the immediate

Table 76: Add/Shift with Prefix Zero-Extend Instructions Description

6.9 Zbb Extension: Basic Bit Manipulation Instructions

The Zbb instructions are used for basic bit manipulation.

6.9.1 Count Leading/Trailing Zeroes Instructions

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
CLZ CTZ	CLZ CTZ		CLZ CTZ		OP-IMM OP-IMM

Figure 92: Count Leading/Trailing Zeroes Instructions

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
CLZW CTZW	CLZW CTZW		CLZW CTZW		OP-IMM-32 OP-IMM-32

Figure 93: Count Leading/Trailing Zeroes Instructions (cont.)

Instruction	Description
CLZ `rd,rs`	Counts the number of 0 bits before the first 1 bit, starting at the most-significant bit and progressing to bit 0. If the input is 0, the output is 64. If the most-significant bit of the input is 1, the output is 0.
CTZ `rd,rs`	Counts the number of 0 bits before the first 1 bit, starting at the least-significant bit and progressing to the most-significant bit. If the input is 0, the output is 64. If the least-significant bit of the input is 1, the output is 0.
CLZW `rd,rs`	Counts the number of 0 bits before the first 1 bit, starting at bit 31 and progressing to bit 0. If the least-significant word is 0, the output is 32. If the most-significant bit of the word is 1, the output is 0.
CTZW `rd,rs`	Counts the number of 0 bits before the first 1 bit, starting at the least-significant bit and progressing to the most-significant word. If the least-significant word is 0, the output is 32. If the least-significant bit of the input is 1, the output is 0.

Table 77: Count Leading/Trailing Zeroes Instructions Description

6.9.2 Count Population Instructions

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
CPOP	CPOP		CPOP		OP-IMM

Figure 94: Count Population Instruction

31–25	24–20	19–15	14–12	11–7	6–0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
CPOPW	CPOPW		CPOPW		OP-IMM-32

Figure 95: Count Population Instruction (cont.)

Instruction	Description
CPOP `rd,rs`	Counts the number of 1 bits in the source register
CPOPW `rd,rs`	Counts the number of 1 bits in the least-significant word of the source register

Table 78: Count Population Instructions Description

6.9.3 Logic-With-Negate Instructions

31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
ANDN			ANDN		OP
ORN			ORN		OP
XNOR			XNOR		OP

Figure 96: Logic-With-Negate Instructions

Instruction	Description
ANDN `rd,rs1,rs2`	Performs bitwise logical AND between `rs1` and the bitwise inversion of `rs2`
ORN `rd,rs1,rs2`	Performs bitwise logical OR between `rs1` and the bitwise inversion of `rs2`
XNOR `rd,rs1,rs2`	Performs bitwise exclusive-NOR on `rs1` and `rs2`

Table 79: Logic-With-Negate Instructions Description

6.9.4 Comparison Instructions

These instructions are arithmetic R-type instructions that return the smaller or larger value of two operands.

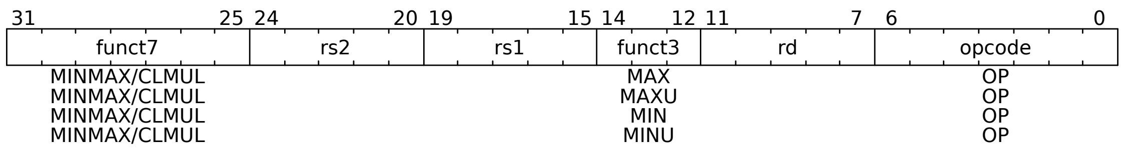

Figure 97: Comparison Instructions

Instruction	Description
MIN `rd,rs1,rs2`	Returns the smaller of two signed integers
MINU `rd,rs1,rs2`	Returns the smaller of two unsigned integers
MAX `rd,rs1,rs2`	Returns the larger of two signed integers
MAXU `rd,rs1,rs2`	Returns the larger of two unsigned integers

Table 80: Comparison Instructions Description

6.9.5 Sign-Extend and Zero-Extend Instructions

These instructions perform the sign-extension or zero-extension of the least-significant 8 or 16 bits of the source register.

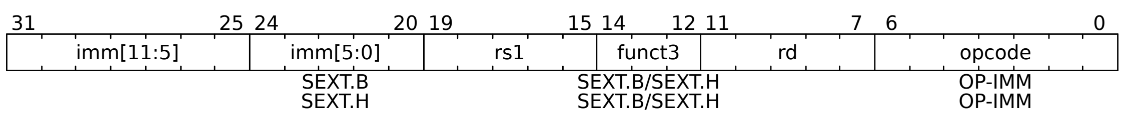

Figure 98: Sign-Extend Instructions

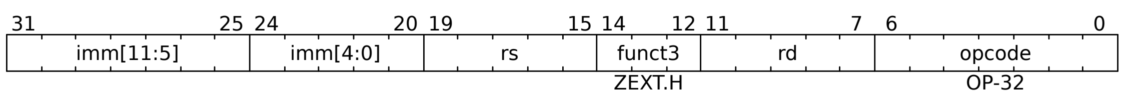

Figure 99: Zero-Extend Instruction

Instruction	Description
SEXT.B `rd,rs`	Sign-extends the least-significant byte in the source to 64 by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits
SEXT.H `rd,rs`	Sign-extends the least-significant halfword in $rs$to 64 by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits
ZEXT.H `rd,rs`	Zero-extends the least-significant halfword of the source to 64 by inserting 0's into all of the bits more significant than 15

Table 81: Sign- and Zero-Extend Instructions

6.9.6 Bitwise Rotation Instructions

Bitwise rotation instructions are similar to the shift-logical operations from the base ISA specification. However, where the shift-logical instructions shift in zeros, the rotate instructions shift in the bits that were shifted out of the other side of the values.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
ROR ROL			ROR ROL		OP OP

Figure 100: Bitwise Rotation Instructions

31 25	24 20	19 15	14 12	11 7	6 0
imm	shamt	rs1	funct3	rd	opcode
RORI			RORI		OP-IMM

Figure 101: Bitwise Rotation Instructions (cont.)

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
RORW ROLW			RORW ROLW		OP-32 OP-32

Figure 102: Bitwise Rotation Instructions (cont.)

31 25	24 20	19 15	14 12	11 7	6 0
imm	shamt	rs1	funct3	rd	opcode
RORIW			RORIW		OP-IMM-32

Figure 103: Bitwise Rotation Instructions (cont.)

Instruction	Description
ROR `rd,rs1,rs2`	Performs a rotate right shift of `rs1` by the amount in the least-significant 6 bits of `rs2`
ROL `rd,rs1,rs2`	Performs a rotate left shift of `rs1` by the amount in the least-significant 6 bits of `rs2`
RORI `rd,rs1,shamt`	Performs a rotate right shift of `rs1` by the amount in the least significant 6 bits of `shamt`
RORW `rd,rs1,rs2`	Performs a rotate right shift of the least-significant word of `rs1` byt the amount in the least-significant 5 bits of `rs2`. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.
ROLW `rd,rs1,rs2`	Performs a rotate left shift on the least-significant word of `rs1` by the amount in the least-significant 5 bits of `rs2`. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.
RORIW `rd,rs1,imm`	Performs a rotate right shift on the least-significant word of `rs1` by the amount in the least-significant 6 bits of `shamt`. The resulting word value is sign-extended by copying bit 31 to all of the more significant bits.

Table 82: Bitwise Rotation Instructions Description

6.9.7 OR Combine Instruction

Instruction	Description
ORC.B `rd,rs`	Combines the bits within every byte through a reciprocal bitwise logical OR. This sets the bits of each byte in the result `rd` to all zeros if no bit within the respective byte of `rs` is set, otherwise it sets the bits to all ones if any bit within the respective byte of `rs` is set.

Table 83: OR Combine Instruction Description

6.9.8 Byte-Reverse Instruction

Instruction	Description
REV8 `rd,rs`	Reverses the order of the bytes in a register

Table 84: Byte-Reverse Instruction Description

6.10 Zicsr Extension: Control and Status Register Instructions

RISC-V defines a separate address space of 4096 Control and Status registers associated with each hart. The defined instructions access counter, timers and floating-point status registers.

31–20	19–15	14–12	11–7	6–0
csr	rs1	funct3	rd	opcode
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Figure 104: Zicsr Instructions

Instruction	**Description**
CSRRW `rd`, `rs1` `csr`	Instruction atomically swaps values in the CSRs and integer registers.
CSRRS `rd`, `rs1` `csr`	Instruction reads the value of the CSR, zero-extends the value to 64-bits, and writes it to integer register rd. The initial value in integer register `rs1` is treated as a bit mask that specifies bit positions to be set in the CSR.
CSRRC `rd`, `rs1` `csr`	Instruction reads the value of the CSR, zero-extends the value to 64-bits, and writes it to integer register rd. The initial value in integer register `rs1` is treated as a bit mask that specifies bit positions to be cleared in the CSR.
CSRRWI `rd`, `rs1` `csr`	Update the CSR using an 64-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register.
CSRRSI `rd`, `rs1` `csr`	Update the CSR using an 64-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register.
CSRRCI `rd`, `rs1` `csr`	If the uimm[4:0] field is zero, then these instructions will not write to the CSR.

Table 85: Control and Status Register Instruction Description

The CSRRWI, CSRRSI, and CSRRCI instructions are similar in kind to CSRRW, CSRRS, and CSRRC respectively, except in that they update the CSR using an 64-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. For CSRRSI and CSRRCI, these instructions will not write to the CSR if the uimm[4:0] field is zero, and they shall not cause any of the size effects that might otherwise occur on a CSR write. For CSRRWI, if `rd = x0`, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read. Both CSRRSI and CSRRCI will always read the CSR and cause any read side effects regardless of the `rd` and `rs1` fields.

Table 86 shows if a CSR reads or writes given a particular CSR.

Register Operand				
Instruction	**rd**	**rs1**	**read CSR?**	**write CSR?**
CSRRW	`x0`	-	no	yes
CSRRW	`!x0`	-	yes	yes
CSRRS/C	-	`x0`	yes	no
CSRRS/C	-	`!x0`	yes	yes
Immediate Operand				
Instruction	**rd**	**uimm**	**read CSR?**	**write CSR?**
CSRRWI	`x0`	-	no	yes
CSRRWI	`!x0`	-	yes	yes
CSRRS/CI	-	0	yes	no
CSRRS/CI	-	!0	yes	yes

Table 86: CSR Reads and Writes

6.10.1 Control and Status Registers

The control and status registers (CSRs) are only accessible using variations of the CSRR (Read) and CSRRW (Write) instructions. Only the CPU executing the csr instruction can read or write these registers, and they are not visible by software outside of the core they reside on. The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs. Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction. A read/write register might also contain some bits that are read-only, in which case, writes to the read-only bits are ignored. Each core functionality has its own control and status registers which are described in the corresponding section.

6.10.2 Defined CSRs

The following tables describe the currently defined CSRs, categorized by privilege level. The usage of the CSRs below is implementation specific. CSRs are only accessible when operating within a specific access mode (user mode, debug mode, supervisor mode, or machine mode). Therefore, attempts to access a non-existent CSR raise an illegal instruction exception, and attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.

Number	Privilege	Name	Description
User Trap Setup			
`0x000`	RW	`ustatus`	User status register.
`0x004`	RW	`uie`	User interrupt-enable register.
`0x005`	RW	`utvec`	User trap handler base address.
User Trap Handling			
`0x040`	RW	`uscratch`	Scratch register for use trap handlers.
`0x041`	RW	`uepc`	User exception program counter.
`0x042`	RW	`ucause`	User trap cause.
`0x043`	RW	`ubadaddr`	User bad address.
`0x044`	RW	`uip`	User interrupt pending.
User Floating-Point CSRs			
`0x001`	RW	`fflags`	Floating-Point Accrued Exceptions.
`0x002`	RW	`frm`	Floating-Point Dynamic Rounding Mode.
`0x003`	RW	`fcsr`	Floating-Point Control and Status Register (`frm` + `fflags`).
User Counter/Timers			
`0xC00`	RO	`cycle`	Cycle counter for RDCYCLE instruction.
`0xC01`	RO	`time`	Timer for RDTIME instruction.
`0xC02`	RO	`instret`	Instructions-retired counter for RDINSTRET instruction.
`0xC03`	RO	`hpmcounter3`	Performance-monitoring counter.
`0xC04`	RO	`hpmcounter4`	Performance-monitoring counter.
		…	
`0xC1F`	RO	`hpmcounter31`	Performance-monitoring counter.

Table 87: User Mode CSRs

Number	Privilege	Name	Description
		Supervisor Trap Setup	
`0x100`	RW	`sstatus`	Supervisor status register.
`0x102`	RW	`sedeleg`	Supervisor exception delegation register.
`0x103`	RW	`sideleg`	Supervisor interrupt delegation register.
`0x104`	RW	`sie`	Supervisor interrupt-enable register.
`0x105`	RW	`stvec`	Supervisor trap handler base address.
`0x106`	RW	`scounteren`	Supervisor counter enable.
		Supervisor Trap Handling	
`0x140`	RW	`sscratch`	Scratch register for supervisor trap handlers.
`0x141`	RW	`sepc`	Supervisor exception program counter.
`0x142`	RW	`scause`	Supervisor trap cause.
`0x143`	RW	`stval`	Supervisor bad address or instruction.
`0x144`	RW	`sip`	Supervisor interrupt pending.
		Supervisor Protection and Translation	
`0x180`	RW	`satp`	Supervisor address translation and protection.

Table 88: Supervisor Mode CSRs

Number	Privilege	Name	Description
Machine Information Registers			
`0xF11`	RO	`mvendorid`	Vendor ID.
`0xF12`	RO	`marchid`	Architecture ID.
`0xF13`	RO	`mimpid`	Implementation ID.
`0xF14`	RO	`mhartid`	Hardware thread ID.
Machine Trap Setup			
`0x300`	RW	`mstatus`	Machine status register.
`0x301`	RW	`misa`	ISA and extensions.
`0x302`	RW	`medeleg`	Machine exception delegation register.
`0x303`	RW	`mideleg`	Machine interrupt delegation register.
`0x304`	RW	`mie`	Machine interrupt-enable register.
`0x305`	RW	`mtvec`	Machine trap-handler base address.
`0x306`	RW	`mcounteren`	Machine counter enable.
Machine Trap Handling			
`0x340`	RW	`mscratch`	Scratch register for machine trap handlers.
`0x341`	RW	`mepc`	Machine exception program counter.
`0x342`	RW	`mcause`	Machine trap cause.
`0x343`	RW	`mtval`	Machine bad address or instruction.
`0x344`	RW	`mip`	Machine interrupt pending.
Machine Memory Protection			
`0x3A0`	RW	`pmpcfg0`	Physical memory protection configuration.
`0x3A1`	RW	`pmpcfg1`	Physical memory protection configuration, RV32 only.
`0x3A2`	RW	`pmpcfg2`	Physical memory protection configuration.
`0x3A3`	RW	`pmpcfg3`	Physical memory protection configuration, RV32 only.
`0x3B0`	RW	`pmpaddr0`	Physical memory protection address register.
`0x3B1`	RW	`pmpaddr1`	Physical memory protection address register.
		…	
`0x3BF`	RW	`pmpaddr15`	Physical memory protection address register.
Machine Counter/Timers			
`0xB00`	RW	`mcycle`	Machine cycle counter.
`0xB02`	RW	`minstret`	Machine instruction-retired counter.
Machine Counter Setup			
`0x320`	RW	`mcountinhibit`	Machine counter-inhibit register.
`0x323`	RW	`mhpmevent3`	Machine performance-monitoring event selector.
`0x324`	RW	`mhpmevent4`	Machine performance-monitoring event selector.
		…	
`0x33F`	RW	`mhpmevent31`	Machine performance-monitoring event selector.
Debug/Trace Register (shared with Debug Mode)			
`0x7A0`	RW	`tselect`	Debug/Trace trigger register select.

Table 89: Machine Mode CSRs

Number	Privilege	Name	Description
`0x7A1`	RW	`tdata1`	First Debug/Trace trigger data register.
`0x7A2`	RW	`tdata2`	Second Debug/Trace trigger data register.
`0x7A3`	RW	`tdata3`	Third Debug/Trace trigger data register.

Table 89: Machine Mode CSRs

Number	Privilege	Name	Description
`0x7B0`	RW	`dcsr`	Debug control and status register.
`0x7B1`	RW	`dpc`	Debug PC.
`0x7B2`	RW	`dscratch`	Debug scratch register.

Table 90: Debug Mode Registers

6.10.3 CSR Access Ordering

On a given hart, explicit and implicit CSR access are performed in program order with respect to those instructions whose execution behavior is affected by the state of the accessed CSR. In particular, a CSR access is performed after the execution of any prior instructions in program order whose behavior modifies or is modified by the CSR state and before the execution of any subsequent instructions in program order whose behavior modifies or is modified by the CSR state.

Furthermore, a CSR read access instruction returns the accessed CSR state before the execution of the instruction, while a CSR write access instruction updates the accessed CSR state after the execution of the instruction. Where the above program order does not hold, CSR accesses are weakly ordered, and the local hart or other harts may observe the CSR accesses in an order different from program order. In addition, CSR accesses are not ordered with respect to explicit memory accesses, unless a CSR access modifies the execution behavior of the instruction that performs the explicit memory access or unless a CSR access and an explicit memory access are ordered by either the syntactic dependencies defined by the memory model or the ordering requirements defined by the Memory-Ordering PMAs. To enforce ordering in all other cases, software should execute a FENCE instruction between the relevant accesses. For the purposes of the `FENCE` instruction, CSR read accesses are classified as device input (I), and CSR write accesses are classified as device output (O). For more about the FENCE instructions, see Section 6.14. For CSR accesses that cause side effects, the above ordering constraints apply to the order of the initiation of those side effects but does not necessarily apply to the order of the completion of those side effects.

6.10.4 SiFive RISC-V Implementation Version Registers

`mvendorid`

The value in `mvendorid` is `0x489`, corresponding to SiFive's JEDEC number.

marchid

The value in `marchid` indicates the overall microarchitecture of the core and at SiFive we use this to distinguish between core generators. The RISC-V standard convention separates `marchid` into open-source and proprietary namespaces using the most-significant bit (MSB) of the `marchid` register; where if the MSB is clear, the `marchid` is for an open-source core, and if the MSB is set, then `marchid` is a proprietary microarchitecture. The open-source namespace is managed by the RISC-V Foundation and the proprietary namespace is managed by SiFive.

SiFive’s E3 and S5 cores are based on the open-source 3/5-Series microarchitecture, which has a Foundation-allocated `marchid` of 1. Our other generators are numbered according to the core series.

Value	Core Generator
`0x8000_0007`	6/7/P200/X200-Series Processor

Table 91: Core Generator Encoding of `marchid`

mimpid

The value in `mimpid` holds an encoded value that uniquely identifies the version of the generator used to build this implementation. If your release version is not included in Table 92, contact your SiFive account manager for more information.

Value	Generator Release Version
`0x0000_0000`	Pre-19.02
`0x2019_0228`	19.02
`0x2019_0531`	19.05
`0x2019_0919`	19.08p0p0 / 19.08.00
`0x2019_1105`	19.08p1p0 / 19.08.01.00
`0x2019_1204`	19.08p2p0 / 19.08.02.00
`0x2020_0423`	19.08p3p0 / 19.08.03.00
`0x0120_0626`	19.08p4p0 / 19.08.04.00
`0x0220_0515`	koala.00.00-preview and koala.01.00-preview
`0x0220_0603`	koala.02.00-preview
`0x0220_0630`	20G1.03.00 / koala.03.00-general
`0x0220_0710`	20G1.04.00 / koala.04.00-general
`0x0220_0826`	20G1.05.00 / koala.05.00-general
`0x0320_0908`	kiwi.00.00-preview
`0x0220_1013`	20G1.06.00 / koala.06.00-general
`0x0220_1120`	20G1.07.00 / koala.07.00-general
`0x0421_0205`	llama.00.00-preview
`0x0421_0324`	21G1.01.00 / llama.01.00-general
`0x0421_0427`	21G1.02.00 / llama.02.00-general
`0x0521_0528`	mongoose.00.00-preview
`0x0521_0714`	21G2.01.00 / mongoose.01.00-general
`0x0521_1008`	21G2.02.00 / mongoose.02.00-general
`0x0621_1027`	narwhal.00.00-preview
`0x0621_1203`	narwhal.01.00-preview
`0x0621_1222`	21G3.02.00 / narwhal.02.00-general

Table 92: Generator Release Encoding of `mimpid`

Reading Implementation Version Registers

To read the `mvendorid`, `marchid`, and `mimpid` registers, simply replace `mimpid` with `mvendorid` or `marchid` as needed.

In C:

```
uintptr_t mimpid;
__asm__ volatile("csrr %0, mimpid" : "=r"(mimpid));
```

In Assembly:

```
csrr a5, mimpid
```

6.10.5 Custom CSRs

SiFive implements some custom CSRs that are specific to the implementation. For these CSRs, including the Feature Disable CSR, consider Chapter 7.

6.11 Base Counters and Timers

RISC-V ISAs provide a set of up to 32×64-bit performance counters and timers that are accessible via unprivileged 64-bit read-only CSR registers `0xC00–0xC1F`. The first three of these (CYCLE, TIME, and INSTRET) have dedicated functions; while the remaining counters, if implemented, provide programmable event counting.

The U74-MC Core Complex implements `mcycle`, `mtime`, and `minstret` counters, which have dedicated functions: cycle count, real-time clock, and instructions-retired, respectively. The timer functionality is based on the `mtime` register. Additionally, the U74-MC Core Complex implements event counters in the form of `mhpmcounter`, which is used to monitor user requested events.

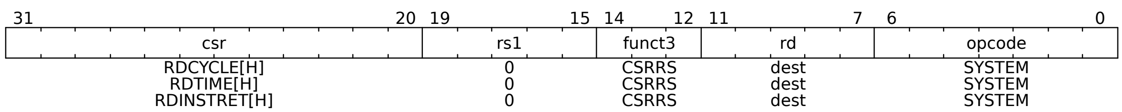

Figure 105: Timer and Counter Pseudoinstructions

Instruction	Description
RDCYCLE `rd`	Reads the64-bits of the cycle CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past.
RDTIME `rd`	Generates an illegal instruction exception. The `mtime` register is memory mapped to the CLINT register space and can be read using a regular load instruction.
RDINSTRET `rd`	Reads the64-bits of the instret CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past.

Table 93: Timer and Counter Pseudoinstruction Description

RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions read the full 64 bits of the `cycle`, `time`, and `instret` counters. The RDCYCLE pseudoinstruction reads the low 64-bits of the cycle CSR (`mcycle`), which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. The RDTIME pseudoinstruction reads the low 64-bits of the time CSR (`mtime`), which counts wall-clock real time that has passed from an arbitrary start time in the past. The RDINSTRET pseudoinstruction reads the low 64-bits of the instret CSR (`minstret`), which counts the number of instructions retired by this hart from some arbitrary start point in the past The rate at which the cycle counter advances is `rtc_clock`. To determine the current rate (cycles per second) of instruction execution, call the `metal_timer_get_timebase_frequency` API. The `metal_timer_get_timebase_frequency` and additional APIs are described in Section 6.11.2 below.

Number	Privilege	Name	Description
`0xC00`	RO	`cycle`	Cycle counter for RDCYCLE instruction
`0xC01`	RO	`time`	Timer for RDTIME instruction
`0xC02`	RO	`instret`	Instruction-retired counter for RDINSTRET instruction

Table 94: Timer and Counter CSRs

6.11.1 Timer Register

`mtime` is a 64-bit read-write register that contains the number of cycles counted from the `rtc_toggle` signal described in the U74-MC Core Complex User Guide. On reset, `mtime` is cleared to zero.

6.11.2 Timer API

The APIs below are used for reading and manipulating the machine timer. Other APIs are described in more detail within the Freedom Metal documentation. https://sifive.github.io/freedom-metal-docs/

Functions

int `metal_timer_get_cyclecount`(int `hartid`, unsigned long long *`cyclecount`)

Read the machine cycle count.

Return

0 upon success

Parameters

- `hartid`: The hart ID to read the cycle count of
- `cyclecount`: The variable to hold the value

int `metal_timer_get_timebase_frequency`(int `hartid`, unsigned long long *`timebase`)

Get the machine timebase frequency.

Return

0 upon success

Parameters

- `hartid`: The hart ID to read the cycle count of
- `timebase`: The variable to hold the value

int `metal_timer_set_tick`(int `hartid`, int `second`)

Set the machine timer tick interval in seconds.

Return

0 upon success

Parameters

- `hartid`: The hart ID to read the cycle count of
- `second`: The number of seconds to set the tick interval to

6.12 Privileged Instructions

The RISC-V architecture implements privileged instructions that can only be executed when the U74-MC Core Complex is operating in a privileged mode. The SYSTEM major opcode is used to encode all of the privileged instructions.

6.12.1 Machine-Mode Privileged Instructions

Environment Call and Breakpoint

These ECALL and EBREAK instructions cause a precise requested trap to the supporting execution environment. The ECALL instruction is used to make a service request to the execution environment. The EBREAK instruction is used to return control to a debugging environment.

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

Figure 106: ECALL and EBREAK Instructions

Trap-Return Instructions

To return after handling a trap, there are separate trap return instructions per privilege level: MRET and SRET. MRET is always provided, while SRET is provided if the respective privilege mode is supported. An xRET instruction can be executed in privilege mode x or higher, where executing a lower-privilege xRET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack.

In addition to manipulating the privilege stack, xRET instructions sets the `pc` to the value stored in the corresponding `xepc` register.

Wait for Interrupt

The Wait for Interrupt (WFI) instruction provides a hint to the U74-MC Core Complex that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart.

31–20	19–15	14–12	11–7	6–0
funct12	rs1	funct3	rd	opcode
WFI	0	PRIV	0	SYSTEM

Figure 107: Wait for Interrupt Instruction

If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt exception will be taken on the following instruction, i.e., execution resumes in the trap handler and `mepc` = `pc` + 4. The WFI instruction can also be executed when interrupts are disabled. The operation of WFI must be unaffected by the global interrupt bits in `mstatus` (MIE/SIE) (i.e., the hart must resume if a locally enabled interrupt becomes pending), but should honor the individual interrupt enables (e.g., MTIE). WFI is also required to resume execution for locally enabled interrupts pending at any privilege level, regardless of the global interrupt enable at each privilege level. If the event that causes the hart to resume execution does not cause an interrupt to be taken, execution will resume at `pc` + 4, and software must determine what action to take, including looping back to repeat the WFI if there was no actionable event.

The suggested way to call WFI is inside an infinite loop as described below.

```
while (1) {
    __asm__ volatile ("wfi");
}
```

In SiFive's implementation of WFI, the WFI instruction is issued and the core goes into an internal clock gating state.

6.12.2 Supervisor-Mode Privileged Instructions

For S-mode enabled cores, one new supervisor-level instruction is provided in addition to SRET.

Supervisor Memory-Management Fence Instruction

The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution.

31–20	19–15	14–12	11–7	6–0
funct12	rs1	funct3	rd	opcode
SFENCE.VMA	vaddr	PRIV	0	SYSTEM

Figure 108: Supervisor Memory-Management Fence Instruction

Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to loads and stores in the instruction stream. Executing an SFENCE.VMA instruction guarantees that any stores in the instruction stream prior to the execution of SFENCE.VMA are ordered before all implicit references subsequent to the SFENCE.VMA. Furthermore, executing an SFENCE.VMA guarantees that any implicit writes caused by instructions prior to the SFENCE.VMA are ordered before all loads and stores subsequent to the SFENCE.VMA.

6.13 ABI - Register File Usage and Calling Conventions

RV64GC_Zba_Zbb_Sscofpmf has 32 `x` registers that are each 64 bits wide.

Register	**ABI Name**	**Description**	**Saver**
`x0`	`zero`	Hard-wired zero	-
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	-
`x4`	`tp`	Thread pointer	-
`x5`	`t0`	Temporary / alternate link register	Caller
`x6-7`	`t1-2`	Temporaries	Caller
`x8`	`s0/fp`	Saved-register / frame-pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-1`	Function arguments / return values	Caller
`x12-17`	`a2-7`	Function arguments	Caller
`x18-27`	`s2-11`	Saved registers	Callee
`x28-31`	`t3-6`	Temporaries	Caller
Floating-Point Registers			
`f0-7`	`ft0-7`	FP temporaries	Caller
`f8-9`	`fs0-1`	FP saved registers	Callee
`f10-11`	`fa0-1`	FP arguments / return values	Caller
`f12-17`	`fa2-7`	FP arguments	Caller
`f18-27`	`fa2-11`	FP saved registers	Callee
`f28-31`	`ft8-11`	FP temporaries	Caller

Table 95: RISC-V Registers

The programmer counter PC hold the address of the current instruction.

- `x1 / ra` - holds the return address for a call.
- `x2 / sp` - stack pointer, points to the current routine stack.
- `x8 / fp / s0` - frame pointer, points to the bottom of the top stack frame.
- `x3 / gp` - global pointer, points into the middle of the global data section.
  The common definition is: .data + 0x800. RISC-V immediate values are 12-bit signed values, which is +/- 2048 in decimal or +/- `0x800` in hex. So that global pointer relative accesses can reach their full extent, the global pointer point + `0x800` into the data section. The linker can then relax LUI+LW, LUI+SW into gp-relative LW or SW, i.e., shorter instruction sequences and access most global data using LW at gp +/- offset

```
LW t0 , 0x800(gp)
LW t1 , 0x7FF(gp)
```

- `x4 / tp` - thread pointer, point to thread-local storage (TLS-mostly used in Linux and RTOS).
  If you create a variable in TLS, every thread has its own copy of the variable, i.e., changes to the variable are local to the thread. This is a static area of memory that gets copied for each thread in a program. It is also used to create libraries that have thread-safe functions, because of the fact that each call to a function has its copy of the same global data, so it's safe.

6.13.1 RISC-V Assembly

RISC-V instructions have opcodes and operands.

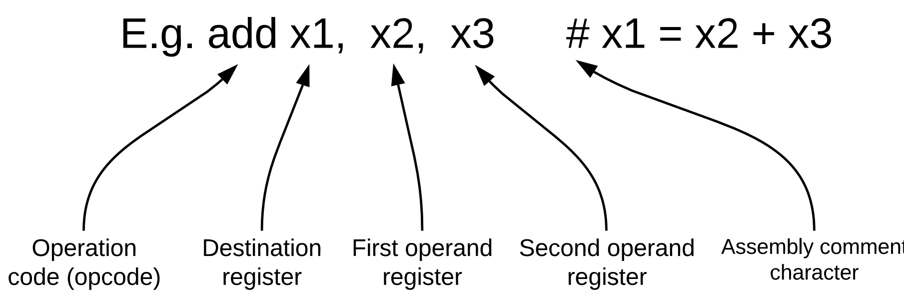

Figure 109: RISC-V Assembly Example

Assembly	C	Description
`add x1,x2,x3`	`a = b + c`	a=`x1`, b=`x2`, c=`x3`
`sub x3,x4,x5`	`d = e - f`	d=`x3`, e=`x4`, f=`x5`
`add x0,x0,x0`	NOP	Writes to `x0` are always ignored
`add x3,x4,x0`	`f = g`	f=`x3`, g=`x4`
`addi x3,x4,-10`	`f = g - 10`	f=`x3`, g=`x4`
`lw x10,12(x13) # 12 = 3x4` `add x11,x12,x10`	`int A[100];` `g = h + A[3];`	Reg `x10` gets A[3] g=`x11`, h=`x12`

`lw x10,12(x13) # 12 = 3x4`
`add x10,x12,x10`
`sw x10,40(x13) # 40 = 10x4`
	`int A[100];`
	`A[10] = h + A[3];`
		Reg `x10` gets A[3]
		h=`x12`
		Reg `x10` gets h + A[3]

`  bne x13,x14,done`
`  add x10,x11,x12`
`done:`
	`if (i == j)`
	`  f = g + h;`
		f=`x10`, g=`x11`, h=`x12`, i=`x13`, j=`x14`

`  bne x10,x14,else`
`  add x10,x11,x12`
`  j done`
`else: sub x10,x11,x12`
`done:`
	`if (i == j)`
	`  f = g + h;`
	`else`
	`  f = g - h;`
		f=`x10`, g=`x11`, h=`x12`, i=`x13`, j=`x14`

Table 96: RISC-V Assembly and C Examples

6.13.2 Assembler to Machine Code

The following flowchart describes how the assembler converts the RISC-V assembly code to machine code.

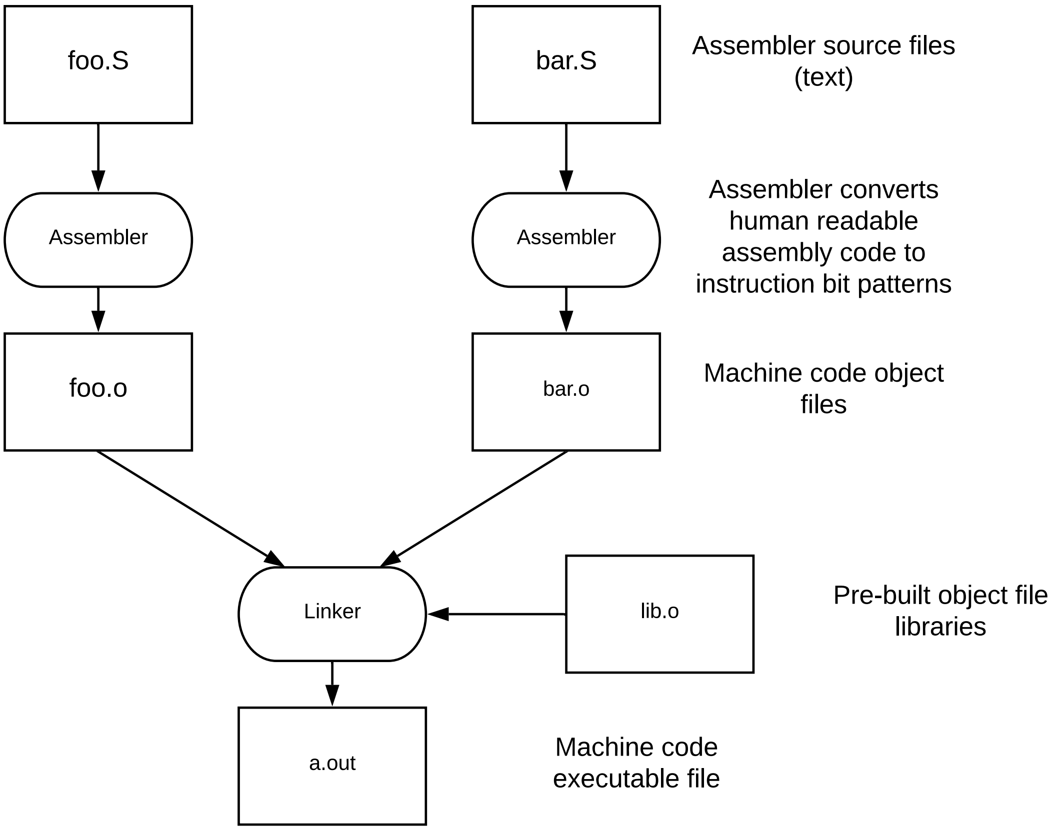

Figure 110: RISC-V Assembly to Machine Code

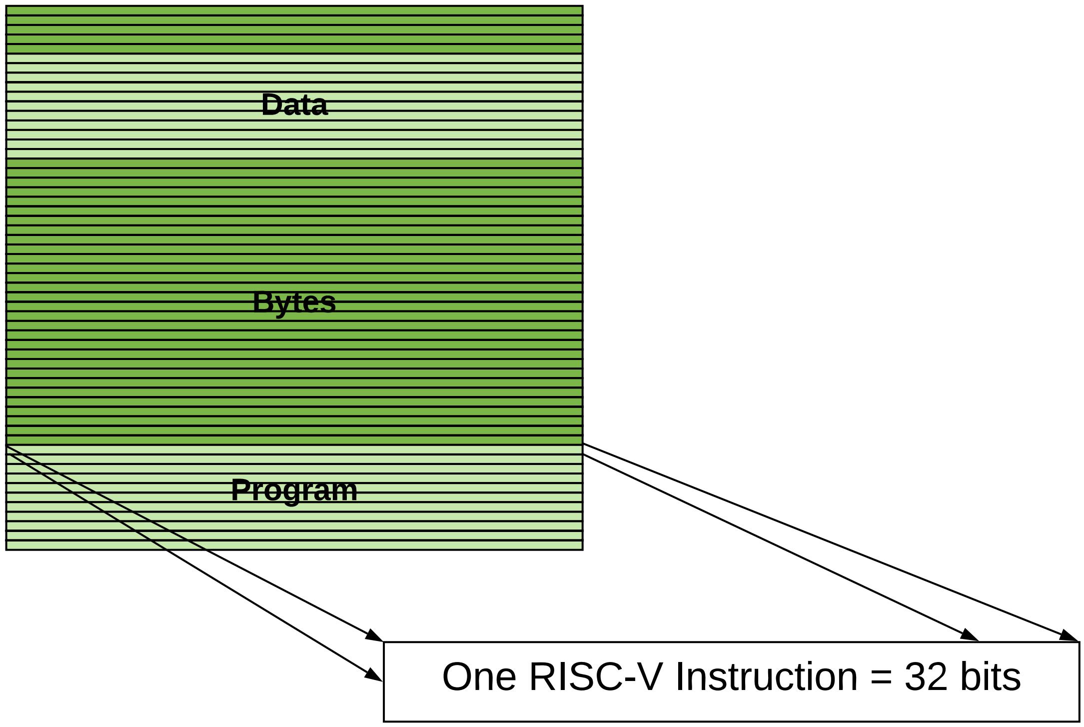

Figure 111: One RISC-V Instruction

6.13.3 Calling a Function (Calling Convention)

1. Put parameters in place where function can access them.
2. Transfer control to function.
3. Acquire local resources needed for function.
4. Perform function task.
5. Place result values where calling code can access and restore any registers might have used.
6. Return control to original caller.

Caller-saved The function invoked can do whatever it likes with the registers. Callee-saved If a function wants to use registers it needs to store and restore them.

Take, for example, the following function:

```
int leaf(int g, int h, int i, int j) {
      int f;
      f = (g+h) - (i+j);
      return f;
}
```

In this function above, arguments are passed in `a0`, `a1`, `a2` and `a3`. The return value is returned in `a0`.

```
addi sp, sp, -8   # adjust stack for 2 items
sw s1, 4(sp)      # save 1 for use afterwards
sw s0, 0(sp)      # save s0 for use afterwards

add s0,a0,a1      # s0 = g + h
add s1,a2,a3      # s1 = i + j
sub a0,s0,s1      # return value (g + h) - (i + j)

lw s0, 0(sp)      # restore register s0 for caller
lw s1, 4(sp)      # restore register s1 for caller
addi s1, 4(sp)    # adjust stack to delete 2 items
jr ra             # jump back to calling routine
```

In the assembly above, notice that the stack pointer was decremented by 8 to make room to save the registers. Also, `s1` and `s0` are saved and will be stored at the end.

Nested Functions

In the case of nested function calls, values held in `a0-7` and `ra` will be clobbered.

Take, for example, the following function:

```
int sumSquare(int x, int y) {
  return mult(x,x) + y;
}
```

In the function above, a function called `sumSquare` is calling `mult`. To execute the function, there's a value in `ra` that `sumSquare` wants to jump back to, but this value will be overwritten by the call to `mult`.

To avoid this, the `sumSquare` return address must be saved before the call to `mult`. To save the return address of `sumSquare`, the function can utilize stack memory. The user can use stack memory to preserve automatic (local) variables that don't fit within the registers.

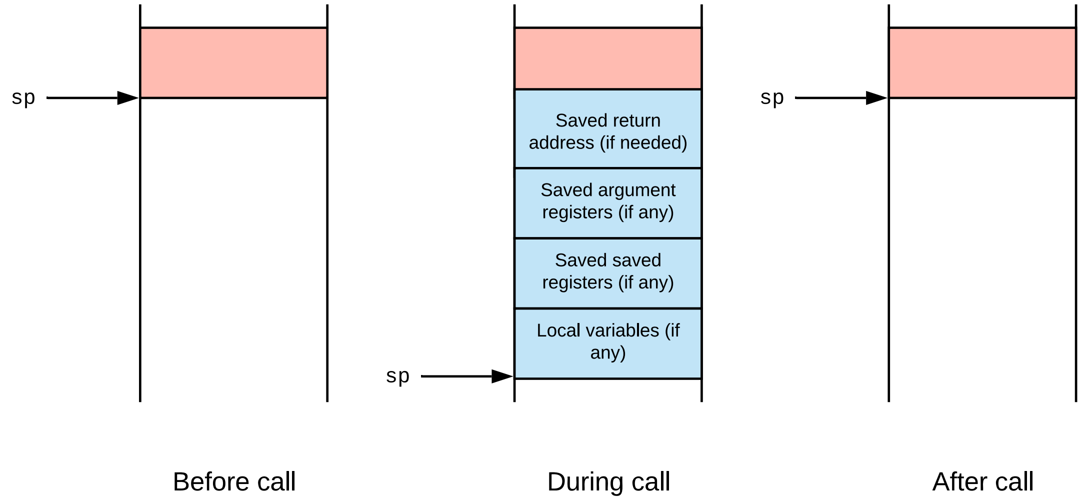

Figure 112: Stack Memory during Function Calls

Consider the assembly for `sumSquare` below:

```
sumSquare:
addi sp,sp,-8       # reserve space on stack
sw ra, 4(sp)        # save return address
sw a1, 0(sp)        # save y
mv a1,a0            # mult(x,x)
jal mult            # call mult
lw a1, 0(sp)        # restore y
add a0,a0,a1        # mult()+y
lw ra, 4(sp)        # get return address
addi sp,sp,8        # restore stack
mult:...
```

6.14 Memory Ordering - FENCE Instructions

In the RISC-V ISA, each thread, referred to as a hart, observes its own memory operations as if they executed sequentially in program order. RISC-V also has a relaxed memory model, which requires explicit FENCE instructions to guarantee the ordering of memory operations.

The FENCE instructions include `FENCE` and `FENCE.I`. The `FENCE` instruction simply ensures that the memory access instructions before the `FENCE` instruction get committed before the `FENCE` instruction is committed. It does not guarantee that those memory access instructions have actually completed. For example, a load instruction before a `FENCE` instruction can commit without waiting for its value to come back from the memory system. `FENCE.I` functions the same as `FENCE`, as well as flushes the instruction cache.

For example, without `FENCE` instructions:

Hart 1 executes:

```
Load X
Store Y
Store Z
```

Because of relaxed memory model, Hart 2 could see stores/loads arranged in any order:

```
Store Z
Load X
Store Y
```

With `FENCE` instructions:

Hart 1 executes:

```
Load X
Store Y
FENCE
Store Z
```

Hart 2 sees:

```
Store Y
Load X
Store Z
```

With `FENCE` instructions, Hart 2 is forced to see the Load X and the Store Y prior to the Store Z, but could arbitrarily see Store Y before Load X or Load X before Store Y. Functionally, `FENCE` instructions order the completion of older memory accesses prior to newer accesses. However, unnecessary `FENCE` instructions slow processes and can hide bugs, so it is essential to identify where and when FENCE should be used.

6.15 Boot Flow

This process is managed as part of the Freedom Metal source code. The freedom-metal boot code supports single core boot or multi-core boot, and contains all the necessary initialization code to enable every core in the system.

1. ENTRY POINT: File: `freedom-metal/src/entry.S`, label: `_enter`.
2. Write `mtvec` register with `early_trap_vector` as default exception handler.
3. Initialize global pointer `gp` register using the generated symbol `__global_pointer$`.
4. Clear feature disable CSR `0x7c1`.
5. Read `mhartid` into register `a0` and call `_start`, which exists in `crt0.S`.
6. We now transition to File: `freedom-metal/gloss/crt0.S`, label: `_start`.
7. Initialize stack pointer, `sp`, with `_sp` generated symbol. Harts with `mhartid` of one or larger are offset by (`_sp` + `__stack_size` × `mhartid`). The `__stack_size` field is generated in the linker file.

8. Check if `mhartid` == `__metal_boot_hart` and run the init code if they are equal. All other harts skip init and go to the Post-Init Flow, step #15.
9. Boot Hart Init Flow begins here.
10. Init data section to destination in defined RAM space.
11. Copy ITIM section, if ITIM code exists, to destination.
12. Zero out `bss` section.
13. Call `atexit` library function that registers the `libc` and `freedom-metal` destructors to run after main returns.
14. Call the `__libc_init_array` library function, which runs all functions marked with `__attribute__((constructor))`.
    a. For example, PLL, UART, hardware prefetcher, L2, L2 prefetcher, and/or L3, if they exist in the design. This method provides full early initialization prior to entering the main application. Prefetchers are initialized to a known value at startup, but not tuned for any specific workload. Refer to the prefetcher software example that demonstrates this tuning.
15. Post-Init Flow Begins Here.
16. Call the C routine `__metal_synchronize_harts`, where hart 0 will release all harts once their individual `msip` bits are set. The `msip` bit is typically used to assert a software interrupt on individual harts, however interrupts are not yet enabled, so `msip` in this case is used as a gatekeeping mechanism.
17. Check `misa` register to see if floating-point hardware is part of the design, and set up `mstatus` accordingly.
18. Single or multi-hart design redirection step.
    a. If design is a single hart only, or a multi-hart design without a C-implemented function `secondary_main`, ONLY the boot hart will continue to `main()`.
    b. For multi-hart designs, all other CPUs will enter sleep via WFI instruction via the weak `secondary_main` label in `crt0.S`, while boot hart runs the application program.
    c. In a multi-hart design which includes a C-defined `secondary_main` function, all harts will enter `secondary_main` as the primary C function.

6.16 Linker File

The linker file generates important symbols that are used in the boot code. The linker file options are found in the `freedom-e-sdk/bsp` path.

There are usually three different linker file options:

- `metal.default.lds` — Use flash and RAM sections

- `metal.ramrodata.lds` — Place read only data in RAM for better performance
- `metal.scratchpad.lds` — Places all code + data sections into available RAM location

Each linker option can be selected by specifying `LINK_TARGET` on the command line.

For example:

```
make PROGRAM=hello TARGET=design-rtl CONFIGURATION=release LINK_TARGET=scratchpad
software
```

The `metal.default.lds` linker file is selected by default when `LINK_TARGET` is not specified. If there is a scenario where a custom linker is required, one of the supplied linker files can be copied and renamed and used for the build. For example, if a new linker file named `metal.newmap.lds` was generated, this can be used at build time by specifying `LINK_TARGET=newmap` on the command line.

6.16.1 Linker File Symbols

The linker file generates symbols that are used by the startup code, so that software can use these symbols to assign the stack pointer, initialize or copy certain RAM sections, and provide the boot hart information. These symbols are made visible to software using the PROVIDE keyword.

For example:

```
__stack_size = DEFINED(__stack_size) ? __stack_size : 0x400;
PROVIDE(__stack_size = __stack_size);
```

Generated Linker Symbols

A description list of the generated linker symbols is shown below.

`__metal_boot_hart`

This is an integer number to describe which hart runs the main init flow. The `mhartid` CSR contains the integer value for each hart. For example, hart 0 has `mhartid==0`, hart 1 has `mhartid==1`, and so on. An assembly example is shown below, where `a0` already contains the `mhartid` value.

```
/* If we're not hart 0, skip the initialization work */
la t0, __metal_boot_hart
bne a0, t0, _skip_init
```

An example on how to use this symbol in C code is shown below.

```
extern int __metal_boot_hart;
int boot_hart = (int)&__metal_boot_hart;
```

Additional linker file generated symbols, along with descriptions are shown below.

`__metal_chicken_bit`
: Status bit to tell startup code to zero out the Feature Disable CSR. Details of this register are internal use only.

`__global_pointer$`
: Static value used to write the `gp` register at startup.

`_sp`
: Address of the end of stack for hart 0, used to initialize the beginning of the stack since the stack grows lower in memory. On a multi-hart system, the start address of the stack for each hart is calculated using (`_sp` + `__stack_size` × `mhartid`)

`metal_segment_bss_target_start`
`metal_segment_bss_target_end`
: Used to zero out global data mapped to `.bss` section.

- Only `__metal_boot_hart` runs this code.

`metal_segment_data_source_start`
`metal_segment_data_target_start`
`metal_segment_data_target_end`
: Used to copy data from image to its destination in RAM.

- Only `__metal_boot_hart` runs this code.

`metal_segment_itim_source_start`
`metal_segment_itim_target_start`
`metal_segment_itim_target_end`
: Code or data can be placed in itim sections using the `__attribute__((section(".itim")))`.

- When this attribute is applied to code or data, the `metal_segment_itim_source_start`, `metal_segment_itim_target_start`, and `metal_segment_itim_target_end` symbols get updated accordingly, and these symbols allow the startup code to copy code and data into the ITIM area.
    - Only `__metal_boot_hart` runs this code.

> **Note**
>
> At the time of this writing, the boot flow does not support C++ projects

6.17 RISC-V Compiler Flags

6.17.1 `arch`, `abi`, and `mtune`

RISC-V targets are described using three arguments:

1. `-march=ISA`: selects the architecture to target.
2. `-mabi=ABI`: selects the ABI to target.
3. `-mtune=CODENAME`: selects the microarchitecture to target.

`-march`

This argument controls which instructions and registers are available for the compiler, as defined by the RISC-V user-level ISA specification.

The RISC-V ISA with 32, 32-bit integer registers and the instructions for multiplication would be denoted as RV32IM. Users can control the set of instructions that GCC uses when generating assembly code by passing the lower-case ISA string to the -march GCC argument; for example, `-march=rv32im`. On RISC-V systems that don't support particular operations, emulation routines may be used to provide the missing functionality.

Example:

```
double dmul(double a, double b) {
  return a * b;
}
```

will compile directly to a FP multiplication instruction when compiled with the D extension:

```
$ riscv64-unknown-elf-gcc test.c -march=rv64imafdc -mabi=lp64d -o- -S -O3
    dmul:
      fmul.d  fa0,fa0,fa1
      ret
```

but will compile to an emulation routine without the D extension:

```
$ riscv64-unknown-elf-gcc test.c -march=rv64i -mabi=lp64 -o- -S -O3
    dmul:
      add     sp,sp,-16
      sd      ra,8(sp)
      call    __muldf3
      ld      ra,8(sp)
      add     sp,sp,16
      jr      ra
```

Similar emulation routines exist for the C intrinsics that are trivially implemented by the M and F extensions.

-mabi

`-mabi` selects the ABI to target. This controls the calling convention (which arguments are passed in which registers) and the layout of data in memory. The `-mabi` argument to GCC specifies both the integer and floating-point ABIs to which the generated code complies. Much like how the `-march` argument specifies which hardware generated code can run on, the `-mabi` argument specifies which software-generated code can link against. We use the standard naming scheme for integer ABIs (`ilp32` or `lp64`), with an argumental single letter appended to select the floating-point registers used by the ABI (`ilp32` vs. `ilp32f` vs. `ilp32d`). In order for objects to be linked together, they must follow the same ABI.

RISC-V defines two integer ABIs and three floating-point ABIs.

- `ilp32`: int, long, and pointers are all 32-bits long. long long is a 64-bit type, char is 8-bit, and short is 16-bit.
- `lp64`: long and pointers are 64-bits long, while int is a 32-bit type. The other types remain the same as `ilp32`.

The floating-point ABIs are a RISC-V specific addition:

- "" (the empty string): No floating-point arguments are passed in registers.
- `f`: 32-bit and smaller floating-point arguments are passed in registers. This ABI requires the F extension, as without F there are no floating-point registers.
- `d`: 64-bit and smaller floating-point arguments are passed in registers. This ABI requires the D extension.

`arch/abi` Combinations

- `march=rv32imafdc -mabi=ilp32d`: Hardware floating-point instructions can be generated and floating-point arguments are passed in registers. This is like the `-mfloat-abi=hard` argument for the Arm® architecture GCC.
- `march=rv32imac -mabi=ilp32`: No floating-point instructions can be generated and no floating-point arguments are passed in registers. This is like the `-mfloat-abi=soft` argument for the Arm architecture GCC.
- `march=rv32imafdc -mabi=ilp32`: Hardware floating-point instructions can be generated, but no floating-point arguments will be passed in registers. This is like the `-mfloat-abi=softfp` argument for the Arm architecture GCC, and is usually used when interfacing with soft-float binaries on a hard-float system.
- `march=rv32imac -mabi=ilp32d`: Illegal, as the ABI requires floating-point arguments are passed in registers but the ISA defines no floating-point registers to pass them in.

Example:

```
double dmul(double a, double b) {
  return b * a;
}
```

If neither the ABI nor ISA contains the concept of floating-point hardware then the C compiler cannot emit any floating-point-specific instructions. In this case, emulation routines are used to perform the computation and the arguments are passed in integer registers:

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imac -mabi=ilp32 -o- -S -O3
 dmul:
   mv      a4,a2
   mv      a5,a3
   add     sp,sp,-16
   mv      a2,a0
   mv      a3,a1
   mv      a0,a4
   mv      a1,a5
   sw      ra,12(sp)
   call    __muldf3
   lw      ra,12(sp)
   add     sp,sp,16
   jr      ra
```

The second case is the exact opposite of this one: everything is supported in hardware. In this case we can emit a single `fmul.d` instruction to perform the computation.

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32d -o- -S -O3
    dmul:
      fmul.d  fa0,fa1,fa0
      ret
```

The third combination is for users who may want to generate code that can be linked with code designed for systems that don’t subsume a particular extension while still taking advantage of the extra instructions present in a particular extension. This is a common problem when dealing with legacy libraries that need to be integrated into newer systems. For this purpose, the compiler arguments and multilib paths designed to cleanly integrate with this workflow. The generated code is essentially a mix between the two above outputs: the arguments are passed in the registers specified by the `ilp32` ABI (as opposed to the `ilp32d` ABI, which could pass these arguments in registers) but then once inside the function the compiler is free to use the full power of the RV32IMAFDC ISA to actually compute the result. While this is less efficient than the code the compiler could generate if it was allowed to take full advantage of the D-extension registers, it’s a lot more efficient than computing the floating-point multiplication without the D-extension instructions

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32 -o- -S -O3
    dmul:
      add     sp,sp,-16
      sw      a0,8(sp)
      sw      a1,12(sp)
      fld     fa5,8(sp)
      sw      a2,8(sp)
      sw      a3,12(sp)
      fld     fa4,8(sp)
      fmul.d  fa5,fa5,fa4
      fsd     fa5,8(sp)
      lw      a0,8(sp)
      lw      a1,12(sp)
      add     sp,sp,16
      jr      ra
```

6.18 Compilation Process

GCC driver script is actually running the preprocessor, then the compiler, then the assembler and finally the linker. If the user runs GCC with the `--save-temps` argument, several intermediate files will be generated.

```
$ riscv64-unknown-linux-gnu-gcc relocation.c -o relocation -O3 --save-temps
```

- `relocation.i`: The preprocessed source, which expands any preprocessor directives (things like #include or #ifdef).
- `relocation.s`: The output of the actual compiler, which is an assembly file (a text file in the RISC-V assembly format).
- `relocation.o`: The output of the assembler, which is an un-linked object file (an ELF file, but not an executable ELF).
- `relocation`: The output of the linker, which is a linked executable (an executable ELF file).

6.19 Large Code Model Workarounds

RISC-V software currently requires that linked symbols reside within a 32-bit range. There are two types of code models defined for RISC-V, **medlow** and **medany**. The medany code model generates auipc/ld pairs to refer to global symbols, which allows the code to be linked at any address, while medlow generates lui/ld pairs to refer to global symbols, which restricts the code to be linked around address zero. They both generate 32-bit signed offsets for referring to symbols, so they both restrict the generated code to being linked within a 2 GiB window. When building software, the code model parameter is passed into the RISC-V toolsuite and it defines a method to generate the necessary instruction combinations to access global symbols within the software program. This is done using `-mcmodel=medany/medlow`. For 32-bit architectures, we use the medlow code model, while medany is used for 64-bit architectures. This is controlled within the ‘setting.mk’ file in the `freedom-e-sdk/bsp` folder.

The real problem occurs when:

1. Total program size exceeds 2 GiB, which is rare
2. When global symbols within a single compiled image are required to reside in a region outside of the 32-bit space

Example for symbols within 32-bit address space:

```
MEMORY
{
ram (wxa!ri) : ORIGIN = 0x80000000, LENGTH = 0x4000
flash (rxai!w) : ORIGIN = 0x20400000, LENGTH = 0x1fc00000
}
```

Example for symbols outside 32-bit address space:

```
MEMORY
{
ram (wxa!ri) : ORIGIN = 0x100000000, LENGTH = 0x4000 /* Updated ORIGIN from
0x80000000 */
flash (rxai!w) : ORIGIN = 0x20400000, LENGTH = 0x1fc00000
}
```

If a software example uses the above memory map, and uses either medlow or medany code models, it will not link successfully. Generated errors will generally contain the following phrase:

```
relocation truncated to fit:
```

A workaround for the linker error "relocation truncated to fit:" is to use `LINK_TARGET=scratchpad` since both the code and data sections get placed into the ram section, as defined by the linker script. Note that this doesn't always solve the problem, since some designs do not have enough memory allocated to the ram section to fit the compiled software example. To solve these cases, SiFive provides support for the compact code model.

6.19.1 RISC-V Code Model Summary

	Medlow	**Medany**	**Compact**
Code	Small	Small	Small
Data	Small	Small	Small
Distance	< 4GB	< 2GB (PC to GOT)	No limitation
Address	4GB (absolute)	No limitation	No limitation

Table 97: RISC-V Code Model Table

As shown in the above table, the compact code model option has no limits on the base address, or the distance between, the code and data sections.

6.19.2 Enabling the Compact Code Model

To enable the large code model, follow the steps below:

1. Enable compact code model in settings.mk file to use `RISCV_CMODEL = compact` instead of `RISCV_CMODEL = medany`.
2. Update assembly files to use new instructions if `__riscv_cmodel_compact` is defined by the toolsuite.
3. Update linker alignment.
4. Use the latest GCC+LLVM toolsuite from SiFive, starting with 2021.06.x.

Makefile Update

To enable the toolsuite to generate the proper code sequences for the compact code model, first update settings.mk file, which can be found in the board support package (BSP) path, similar to `freedom-e-sdk/bsp/design-rtl`.

Change: `RISCV_CMODEL = medany` ⇒ `RISCV_CMODEL = compact`

The `RISCV_CMODEL` definition gets passed into the toolsuite using the `-mcmodel` switch.

> **Note**
>
> For 32-bit designs, which do not require the use of the compact code model, you will find `RISCV_CMODEL = medlow` in settings.mk.

This `-mcmodel=compact` option will enable the symbol `__riscv_cmodel_compact` to be visible within the code, and can be used to determine the correct code sequences to use within assembly files.

Assembly File Updates

Assembly files may need to be hand-edited to support the compact code model if they reference a global symbol. The following freedom-metal source files now support the compact code model option:

1. `freedom-metal/src/entry.S`
2. `freedom-metal/src/scrub.S`
3. `freedom-metal/gloss/crt0.S`

Linker Alignment

The global pointer alignment is required to be: `PROVIDE( __global_pointer$ = ALIGN . + 0x800 ), 16;`

See metal.default.lds, or the *.lds you plan to use.

6.20 Pipeline Hazards

The pipeline only interlocks on read-after-write and write-after-write hazards, so instructions may be scheduled to avoid stalls.

6.20.1 Read-After-Write Hazards

Read-after-Write (RAW) hazards occur when an instruction tries to read a register before a preceding instruction tries to write to it. This hazard describes a situation where an instruction refers to a result that has not been calculated or retrieved. This situation is possible because even though an instruction was executed after a prior instruction, the prior instruction may only have processed partly through the core pipeline.

Example:

- Instruction 1: `x1 + x3` is saved in `x2`
- Instruction 2: `x2 + x3` is saved in `x4`

The first instruction is calculating a value (`x1 + x3`) to be saved in `x2`. The second instruction is going to use the value of `x2` to compute a result to be saved in `x4`. However, in the core pipeline, when operations are fetched for the second operation, the results from the first operation have not yet been saved.

6.20.2 Write-After-Write Hazards

Write-after-write (WAW) hazards occur when an instruction tries to write an operand before it is written by a preceding instruction.

Example:

- Instruction 1: `x4 + x7` is saved in `x2`
- Instruction 2: `x1 + x3` is saved in `x2`

Write-back of instruction 2 must be delayed until instruction 1 finishes executing.

In general, MMIO accesses stall when there is a hazard on the result caused by either RAW or WAW. So, instructions may be scheduled to avoid stalls.

6.21 Reading CSRs

There are several methods for reading the CSRs that are implemented in the U74-MC Core Complex. A full list of the defined RISC-V CSRs are described in Section 6.10.2.

1. Inline assembly using `csrr` instruction and the register name. For example, reading the `misa` CSR:

```
int misa;
__asm__ volatile("csrr %0, misa" : "=r" (misa));
```

2. Using the Freedom Metal API `METAL_CPU_GET_CSR`. Again, reading the `misa` CSR:

```
int misa_value;
METAL_CPU_GET_CSR(misa,misa_value);
```

   In the second method, the first argument is the register name and the second is the variable to store the result in.

Both inline assembly and Freedom Metal API methods can receive the CSR number instead of its name. For example:

```
int mscratch;
METAL_CPU_GET_CSR(0x340, mscratch_value); // reading mscratch csr
```

> **Note**
>
> Accessing CSRs has to be according to the privilege level you are in. Attempting to access a CSR in a privilege level higher than the current level of operation will result in an exception.

To access a privileged CSR, the user must switch to the appropriate privilege level. This can be done using the following Freedom Metal API:

```
metal_privilege_drop_to_mode(METAL_PRIVILEGE_USER,
                             my_regfile,
                             user_mode_entry_point);
```

The Freedom Metal API routines and more examples located in freedom-e-sdk/software directory.

Chapter 7

Custom Instructions and CSRs

This chapter describes some of the custom instructions and CSRs configured in the U74-MC Core Complex.

7.1 SiFive Custom Instructions

These custom instructions use the `SYSTEM` instruction encoding space, which is the same as the custom CSR encoding space, but with `funct3=0`.

7.1.1 `CFLUSH.D.L1`

- Implemented as state machine in L1 data cache, for cores with data caches.
- Only available in M-mode. Execution from modes other than M-mode raises an illegal-instruction exception.
- Opcode `0xFC000073`, with optional `rs1` field in bits [19:15].
- When `rs1 = x0`, `CFLUSH.D.L1` writes back and invalidates all lines in the L1 data cache.
- When `rs1 ≠ x0`, an illegal-instruction exception is raised.
- If the effective privilege mode does not have write permissions to the address in `rs1`, then a store access or store page-fault exception is raised.
- If the address in `rs1` is in an uncacheable region with write permissions, the instruction has no effect but raises no exceptions.
- Note that if the PMP scheme write-protects only part of a cache line, then using a value for `rs1` in the write-protected region will cause an exception, whereas using a value for `rs1` in the write-permitted region will write back the entire cache line.

7.1.2 `CDISCARD.D.L1`

- Implemented as state machine in L1 data cache, for cores with data caches.
- Only available in M-mode. Execution from modes other than M-mode raises an illegal-instruction exception.
- Opcode `0xFC200073`, with optional `rs1` field in bits `[19:15]`.

- When `rs1 = x0`, `CDISCARD.D.L1` invalidates, but does not write back, all lines in the L1 data cache. Dirty data within the cache is lost.
- When `rs1 ≠ x0`, an illegal-instruction exception is raised.
- If the effective privilege mode does not have write permissions to the address in `rs1`, then a store access or store page-fault exception is raised.
- If the address in `rs1` is in an uncacheable region with write permissions, the instruction has no effect but raises no exceptions.
- Note that if the PMP scheme write-protects only part of a cache line, then using a value for `rs1` in the write-protected region will cause an exception, whereas using a value for `rs1` in the write-permitted region will invalidate and discard the entire cache line.

7.1.3 CEASE

- Privileged instruction only available in M-mode.
- Opcode `0x30500073`.
- After retiring `CEASE`, hart will not retire another instruction until reset.
- Instigates power-down sequence, which will eventually raise the `cease_from_tile_N` signal to the outside of the Core Complex, indicating that it is safe to power down.
- `CEASE` has no effect on System Bus Access.
- Debug `haltreq` will not work after a `CEASE` instruction has retired.

7.1.4 PAUSE

- Opcode `0x0100000F`, which is a `FENCE` instruction with predecessor set W and null successor set. Therefore, `PAUSE` is a `HINT` instruction that executes as a no-op on all RISC-V implementations.
- This instruction may be used for more efficient idling in spin-wait loops.
- This instruction causes a stall of up to 32 cycles or until a cache eviction occurs, whichever comes first.

7.2 SiFive Custom CSRs

These custom CSRs use the custom CSR encoding space. Refer to Appendix C for an exhaustive list of the custom CSRs available on the U74-MC Core Complex.

7.2.1 Branch Prediction Mode CSR

This SiFive custom extension adds an M-mode CSR to control the current branch prediction mode, `mbpm` at CSR `0x7C0`.

The U74-MC Core Complex's branch prediction system includes a Return Address Stack (RAS), a Branch Target Buffer (BTB), and a Branch History Table (BHT). While branch predictors are essential to achieve high performance in pipelined processors, they can also cause undesirable timing variability for hard real-time systems. The `mbpm` register provides a means to customize the branch predictor behavior to trade average performance for a more predictable execution time.

Branch Prediction Mode CSR				
CSR	`0x7C0`			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	`static`	RW	`0x0`	Branch-Direction Prediction
[7:1]	Reserved	RO	`0x0`	

Table 98: Branch Prediction Mode CSR

The `static` bit determines the value returned by the BHT component of the branch prediction system. A zero value indicates dynamic direction prediction, and a non-zero value indicates static-taken direction prediction. The BTB is cleared on any write to `static`, and the RAS is unaffected by writes to `static`.

7.2.2 SiFive Feature Disable CSR

The SiFive custom M-mode Feature Disable CSR is provided to enable or disable certain microarchitectural features. In the U74-MC Core Complex, CSR `0x7C1` has been allocated for this purpose. These features are described in Table 99.

> **Warning**
>
> The features that can be controlled by this CSR are subject to change or removal in future releases. It is not advised to depend on this CSR for development.

A feature is fully enabled when the associated bit is zero. If a particular core does not support the disabling of a feature, the corresponding bit is hardwired to zero.

On reset, all implemented bits are set to 1, disabling all features. The bootloader is responsible for turning on all required features and can simply write zero to turn on the maximal set of features. SiFive's Freedom Metal bootloader handles turning on these features; when using a custom bootloader, clearing the Feature Disable CSR must be implemented.

Note that arbitrary toggling of the Feature Disable CSR bits is neither recommended nor supported; they are only intended to be set from 1 to 0. A particular Feature Disable CSR bit is only to be used in a very limited number of situations, as detailed in the **Example Usage** entry in Table 100.

SiFive Feature Disable CSR				
CSR	`0x7C1`			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	`disableDCacheClockGate`	RW	`0x1`	Disable data cache clock gating
1	`disableICacheClockGate`	RW	`0x1`	Disable instruction cache clock gating
2	`disableCoreClockGate`	RW	`0x1`	Disable core clock gating
3	`disableSpeculativeICacheRefill`	RW	`0x1`	Disable speculative instruction cache refill
[6:4]	Reserved	RO	`0x0`	
7	`disableTileClockGate`	RW	`0x1`	Disable tile clock gating
8	Reserved	RO	`0x0`	
9	`suppressCorruptOnGrantData`	RW	`0x1`	Suppress corrupt signal on GrantData messages
[15:10]	Reserved	RO	`0x0`	
16	`branchpredicationdisable`	RW	`0x1`	Disable short forward branch optimization
17	`prefetchdisable`	RW	`0x1`	Disable instruction cache next-line prefetcher
[31:18]	Reserved	RO	`0x0`	

Table 99: SiFive Feature Disable CSR

Feature Disable CSR Usage	
Bit	**Description / Usage**
3	Disable speculative instruction cache refill **Example Usage** A particular integration might require that execution from the System Port range be disallowed. Startup code would first configure PMP to prevent execution from the System Port range, followed by clearing bit 3 of the Feature Disable CSR. This would enable speculative instruction cache refill accesses, without allowing those to access the System Port range because PMP would prohibit such accesses.
9	Suppress corrupt signal on GrantData messages **Example Usage 1** When running in debug mode on configurations having both ECC and a BEU, setting bit 9 of the Feature Disable CSR will suppress debug mode errors. **Example Usage 2** Startup code could scrub errors present in RAMs at power-on, followed by clearing bit 9 of the Feature Disable CSR to allow normal operation.

Table 100: SiFive Feature Disable CSR Usage

7.2.3 Power Dial CSR

The Power Dial CSR, at `0x7C8`, provides a method of scaling down dynamic power in a core in order to limit maximum power without frequency changes. This CSR is further described in Section 15.2.1.

7.3 Other Custom Instructions and CSRs

Other custom instructions and CSRs may be implemented, but their functionality is not documented further here, and they should not be used in this version of the U74-MC Core Complex.

Chapter 8

Interrupts and Exceptions

This chapter describes how interrupt and exception concepts in the RISC-V architecture apply to the U74-MC Core Complex.

8.1 Interrupt Concepts

Interrupts are *asynchronous* events that cause program execution to change to a specific location in the software application to handle the interrupting event. When processing of the interrupt is complete, program execution resumes back to the original program execution location. For example, a timer that triggers every 10 milliseconds will cause the CPU to branch to the interrupt handler, acknowledge the interrupt, and set the next 10 millisecond interval.

The U74-MC Core Complex supports machine mode and supervisor mode interrupts. By default, all interrupts are handled in machine mode. For harts that support supervisor mode, it is possible to selectively delegate interrupts to supervisor mode.

The Core Complex also has support for the following types of RISC-V interrupts: local and global. Local interrupts are signaled directly to an individual hart with a dedicated interrupt exception code and fixed priority. This allows for reduced interrupt latency as no arbitration is required to determine which hart will service a given request and no additional memory accesses are required to determine the cause of the interrupt. Software and timer interrupts are local interrupts generated by the Core-Local Interruptor (CLINT). The U74-MC Core Complex contains no other local interrupt sources.

Global interrupts are routed through a Platform-Level Interrupt Controller (PLIC), which can direct interrupts to any hart in the system via the external interrupt. Decoupling global interrupts from the harts allow the design of the PLIC to be tailored to the platform, permitting a broad range of attributes like the number of interrupts and the prioritization and routing schemes.

Chapter 9 describes the CLINT. Chapter 10 describes the global interrupt architecture and the PLIC design.

8.2 Exception Concepts

Exceptions are different from interrupts in that they typically occur *synchronously* to the instruction execution flow, and most often are the result of an unexpected event that results in the program to enter an exception handler. For example, if a hart is operating in supervisor mode and attempts to access a machine mode only Control and Status Register (CSR), it will immediately enter the exception handler and determine the next course of action. The exception code in the `mstatus` register will hold a value of 0x2, showing that an illegal instruction exception occurred. Based on the requirements of the system, the supervisor mode application may report an error and/or terminate the program entirely.

There are no specific enable bits to allow exceptions to occur since they are always enabled by default. However, early in the boot flow, software should set up `mtvec.BASE` to a defined value, which contains the base address of the default exception handler. All exceptions will trap to `mtvec.BASE`. Software must read the `mcause` CSR to determine the source of the exception, and take appropriate action.

Synchronous exceptions that occur from within an interrupt handler will immediately cause program execution to abort the interrupt handler and enter the exception handler. Exceptions within an interrupt handler are usually the result of a software bug and should generally be avoided since `mepc` and `mcause` CSRs will be overwritten from the values captured in the original interrupt context.

The RISC-V defined synchronous exceptions have a priority order which may need to be considered when multiple exceptions occur simultaneously from a single instruction. Table 101 describes the synchronous exception priority order.

Priority	Exception Code	Description
Highest	3	Instruction address breakpoint
	12	Instruction page fault
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8, 9, 11	Environment call
	3	Environment break
	3	Load/Store/AMO address breakpoint
	6	Store/AMO address misaligned
	4	Load address misaligned
	15	Store/AMO page fault
	13	Load page fault
Lowest	7	Store/AMO access fault
	5	Load access fault

Table 101: Exception Priority

Refer to Table 109 for the full list of exception codes.

Data address breakpoints (watchpoints), Instruction address breakpoints, and environment break exceptions (`EBREAK`) all have the same exception code (3), but different priority, as shown in the table above.

Instruction address misaligned exceptions (0) have lower priority than other instruction address exceptions because they are the result of control-flow instructions with misaligned targets, rather than from instruction fetch.

Some of the helpful CSRs for debugging exceptions and interrupts are described below:

CSR	Description
`exception`	SiFive Scope signal. Indicates the moment that an exception occurs in the write-back (commit) stage.
`mcause`	Contains the cause value of the exception/interrupt. See Section 8.7.5 for more description.
`mepc`	Contains the `pc` where the exception occurs.
`mtval`	If the cause is a load/store fault, this register has the value of the problematic address. If it is an invalid instruction, it provides the instruction that the core tried to execute.
`mstatus`	Contains the interrupt enables, privilege modes, and general status of execution. See Section 8.7.1 for more description.
`mtvec`	Contains the vector that the core will jump to when an exception occurs. If this is not a valid executable value, you may get a double exception when jumping to the exception handler, so it is important to look at all these registers when the exception FIRST occurs. See Section 8.7.2 for more description.

Table 102: Summary of Exception and Interrupt CSRs

8.3 Trap Concepts

The term trap describes the transfer of control in a software application, where trap handling typically executes in a more privileged environment. For example, a particular hart contains three privilege modes: machine, supervisor, and user. Each privilege mode has its own software execution environment including a dedicated stack area. Additionally, each privilege mode contains separate control and status registers (CSRs) for trap handling. While operating in user mode, a context switch is required to handle an event in supervisor mode. The software sets up the system for a context switch and then an ECALL instruction is executed, which synchronously switches control to the environment-call-from-user-mode exception handler.

The default mode out of reset is machine mode. Software begins execution at the highest privilege level, which allows all CSRs and system resources to be initialized before any privilege level changes. The steps below describe the required steps necessary to change privilege mode from machine to user mode, on a particular design that also includes supervisor mode.

1. Interrupts should first be disabled globally by writing `mstatus.MIE` to 0, which is the default reset value.
2. Write `mtvec` CSR with the base address of the machine mode exception handler. This is a required step in any boot flow.
3. Write `mstatus.MPP` to 0 to set the previous mode to user, which allows us to *return* to that mode.

4. Setup the Physical Memory Protection (PMP) regions to grant the required regions to user and supervisor mode, and optionally, revoke permissions from machine mode.
5. Write `stvec` CSR with the base address of the supervisor mode exception handler.
6. Write `medeleg` register to delegate exceptions to supervisor mode. Consider `ECALL` and page fault exceptions.
7. Write `mstatus.FS` to enable floating-point (if supported).
8. Store machine mode user registers to stack or to an application-specific frame pointer.
9. Write `mepc` with the entry point of user mode software
10. Execute `mret` instruction to enter user mode.

> **Note**
>
> There is only one set of user registers (`x1-x31`) that are used across all privilege levels, so application software is responsible for saving and restoring state when entering and exiting different levels.

8.4 Interrupt Block Diagram

The U74-MC Core Complex interrupt architecture is depicted in Figure 113.

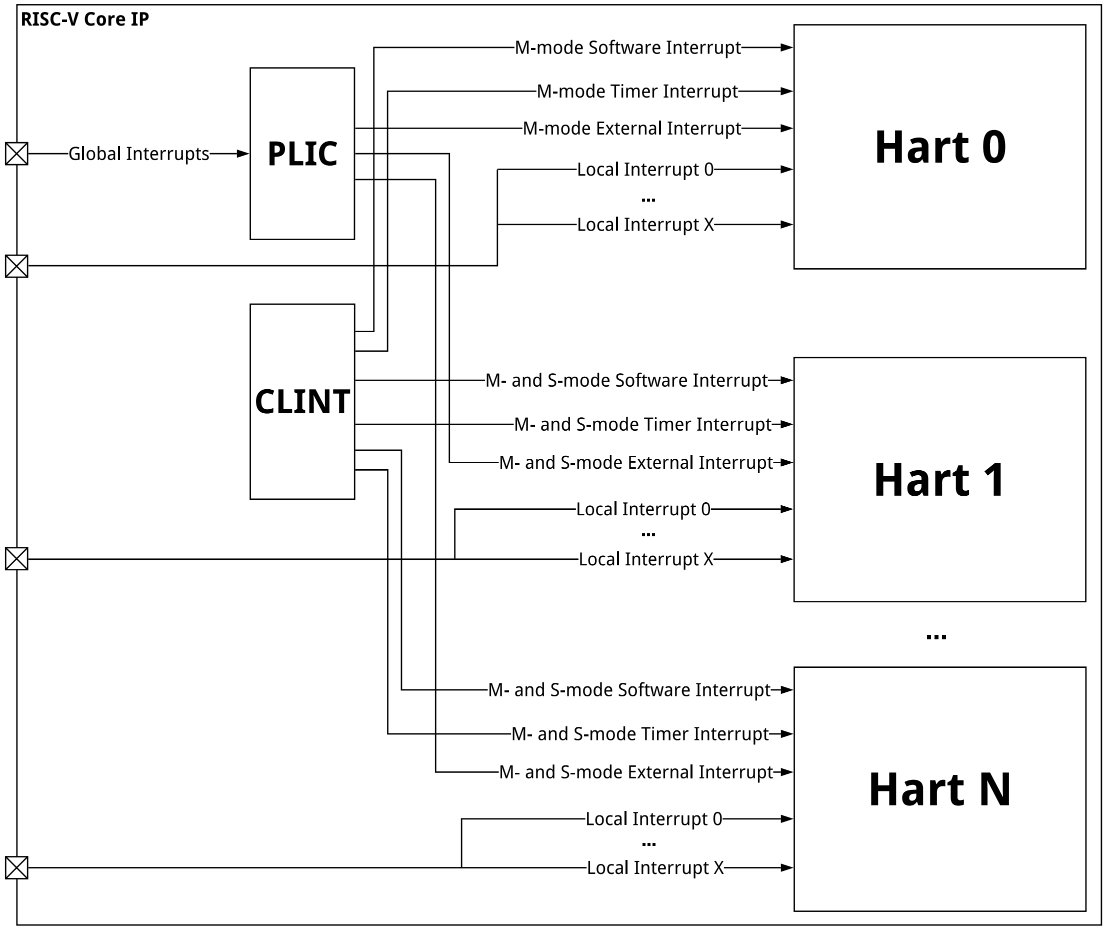

Figure 113: U74-MC Core Complex Interrupt Architecture Block Diagram

8.5 Local Interrupts

Software interrupts (Interrupt ID #3) are triggered by writing the memory-mapped interrupt pending register `msip` for a particular hart. Other harts are able to write `msip` to trigger a software interrupt on any other hart in the U74-MC Core Complex. This allows for efficient interprocessor communication. The `msip` register is described in Table 107.

Timer interrupts (Interrupt ID #7) are triggered when the memory-mapped register `mtime` is greater than or equal to the global timebase register `mtimecmp`, and both registers are part of the CLINT memory map. `mtimecmp` can be written by other harts to set up timer interrupts. The `mtime` and `mtimecmp` registers are generally only available in machine mode, unless the PMP grants user or supervisor mode access to the memory-mapped region in which they reside.

Global interrupts are usually first routed to the PLIC, then into the hart using external interrupts (Interrupt ID #11).

8.6 Interrupt Operation

Within a privilege mode *M*, if the associated global interrupt-enable {ie} is clear, then no interrupts will be taken in that privilege mode, but a pending-enabled interrupt in a higher privilege mode will preempt current execution. If {ie} is set, then pending-enabled interrupts at a higher interrupt level in the same privilege mode will preempt current execution and run the interrupt handler for the higher interrupt level.

When an interrupt or synchronous exception is taken, the privilege mode is modified to reflect the new privilege mode. The global interrupt-enable bit of the handler's privilege mode is cleared.

8.6.1 Interrupt Entry and Exit

When an interrupt occurs:

- The value of `mstatus.MIE` is copied into `mcause.MPIE`, and then `mstatus.MIE` is cleared, effectively disabling interrupts.
- The privilege mode prior to the interrupt is encoded in `mstatus.MPP`.
- The current `pc` is copied into the `mepc` register, and then `pc` is set to the value specified by `mtvec` as defined by the `mtvec.MODE` described in Table 105.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. When an `mret` instruction is executed, the following occurs:

- The privilege mode is set to the value encoded in `mstatus.MPP`.
- The global interrupt enable, `mstatus.MIE`, is set to the value of `mcause.MPIE`.
- The `pc` is set to the value of `mepc`.

At this point, control is handed over to software.

At the software level, interrupt attributes can be applied to interrupt processing functions, as described in Section 9.4.

The Control and Status Registers (CSRs) involved in handling RISC-V interrupts are described in Section 8.7.

8.7 Interrupt Control and Status Registers

The U74-MC Core Complex specific implementation of interrupt CSRs is described below. For a complete description of RISC-V interrupt behavior and how to access CSRs, please consult *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

8.7.1 Machine Status Register (`mstatus`)

The `mstatus` register keeps track of and controls the hart's current operating state, including whether or not interrupts are enabled. A summary of the `mstatus` fields related to interrupts in the U74-MC Core Complex is provided in Table 103. Note that this is not a complete description of `mstatus` as it contains fields unrelated to interrupts. For the full description of `mstatus`, please consult *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

Machine Status Register (`mstatus`)			
CSR	`0x300`		
Bits	**Field Name**	**Attr.**	**Description**
0	Reserved	WPRI	
1	`SIE`	RW	Supervisor Interrupt Enable
2	Reserved	WPRI	
3	`MIE`	RW	Machine Interrupt Enable
4	Reserved	WPRI	
5	`SPIE`	RW	Supervisor Previous Interrupt Enable
6	Reserved	WPRI	
7	`MPIE`	RW	Machine Previous Interrupt Enable
8	`SPP`	RW	Supervisor Previous Privilege Mode
[10:9]	Reserved	WPRI	
[12:11]	`MPP[1:0]`	RW	Machine Previous Privilege Mode

Table 103: Machine Status Register (partial)

Interrupts are enabled by setting the `MIE` bit in `mstatus`. Prior to writing `mstatus.MIE`=1, it is recommended to first enable interrupts in `mie`.

8.7.2 Machine Trap Vector (`mtvec`)

The `mtvec` register has two main functions: defining the base address of the trap vector, and setting the mode by which the U74-MC Core Complex will process interrupts. For Direct and Vectored modes, the interrupt processing mode is defined in the `MODE` field of the `mtvec` register. The `mtvec` register is described in Table 104, and the `mtvec.MODE` field is described in Table 105.

Machine Trap Vector Register (`mtvec`)			
CSR	`0x305`		
Bits	**Field Name**	**Attr.**	**Description**
[1:0]	`MODE`	WARL	`MODE` Sets the interrupt processing mode. The encoding for the U74-MC Core Complex supported modes is described in Table 105.
[63:2]	`BASE[63:2]`	WARL	Interrupt Vector Base Address. Operating in Direct Mode requires 4-byte alignment. Operating in Vectored Mode requires 256-byte alignment.

Table 104: Machine Trap Vector Register

MODE Field Encoding `mtvec.MODE`		
Value	**Mode**	**Description**
0x0	Direct	All asynchronous interrupts and synchronous exceptions set `pc` to `BASE`.
0x1	Vectored	Exceptions set `pc` to `BASE`, interrupts set `pc` to `BASE` + 4 × `mcause.EXCCODE`.
≥0x2	Reserved	

Table 105: Encoding of `mtvec.MODE`

Mode Direct

When operating in direct mode, all interrupts and exceptions trap to the `mtvec.BASE` address. Inside the trap handler, software must read the `mcause` register to determine what triggered the trap. The `mcause` register is described in Table 108.

When operating in Direct Mode, `BASE` must be 4-byte aligned.

Mode Vectored

While operating in vectored mode, interrupts set the `pc` to `mtvec.BASE` + 4 × exception code (`mcause.EXCCODE`). For example, if a machine timer interrupt is taken, the `pc` is set to `mtvec.BASE` + `0x1C`. Typically, the trap vector table is populated with jump instructions to transfer control to interrupt-specific trap handlers.

In vectored interrupt mode, `BASE` must be 256-byte aligned.

All machine external interrupts (global interrupts) are mapped to exception code 11. Thus, when interrupt vectoring is enabled, the `pc` is set to address `mtvec.BASE` + `0x2C` for any global interrupt.

8.7.3 Machine Interrupt Enable (`mie`)

Individual interrupts are enabled by setting the appropriate bit in the `mie` register. The `mie` register is described in Table 106.

Machine Interrupt Enable Register (`mie`)			
CSR	`0x304`		
Bits	**Field Name**	**Attr.**	**Description**
0	Reserved	WPRI	
1	`SSIE`	RW	Supervisor Software Interrupt Enable
2	Reserved	WPRI	
3	`MSIE`	RW	Machine Software Interrupt Enable
4	Reserved	WPRI	
5	`STIE`	RW	Supervisor Timer Interrupt Enable
6	Reserved	WPRI	
7	`MTIE`	RW	Machine Timer Interrupt Enable
8	Reserved	WPRI	
9	`SEIE`	RW	Supervisor External Interrupt Enable
10	Reserved	WPRI	
11	`MEIE`	RW	Machine External Interrupt Enable
[63:12]	Reserved	WPRI	

Table 106: Machine Interrupt Enable Register

8.7.4 Machine Interrupt Pending (`mip`)

The machine interrupt pending (`mip`) register indicates which interrupts are currently pending. The `mip` register is described in Table 107.

Machine Interrupt Pending Register (`mip`)			
CSR	`0x344`		
Bits	**Field Name**	**Attr.**	**Description**
0	Reserved	WIRI	
1	`SSIP`	RW	Supervisor Software Interrupt Pending
2	Reserved	WIRI	
3	`MSIP`	RO	Machine Software Interrupt Pending
4	Reserved	WIRI	
5	`STIP`	RW	Supervisor Timer Interrupt Pending
6	Reserved	WIRI	
7	`MTIP`	RO	Machine Timer Interrupt Pending
8	Reserved	WIRI	
9	`SEIP`	RW	Supervisor External Interrupt Pending
10	Reserved	WIRI	
11	`MEIP`	RO	Machine External Interrupt Pending
[63:12]	Reserved	WIRI	

Table 107: Machine Interrupt Pending Register

8.7.5 Machine Cause (`mcause`)

When a trap is taken in machine mode, `mcause` is written with a code indicating the event that caused the trap. When the event that caused the trap is an interrupt, the most-significant bit of `mcause` is set to 1, and the least-significant bits indicate the interrupt number, using the same encoding as the bit positions in `mip`. For example, a Machine Timer Interrupt causes `mcause` to be set to `0x8000_0000_0000_0007`. `mcause` is also used to indicate the cause of synchronous exceptions, in which case the most-significant bit of `mcause` is set to 0.

See Table 108 for more details about the `mcause` register. Refer to Table 109 for a list of synchronous exception codes.

Machine Cause Register (`mcause`)			
CSR	`0x342`		
Bits	**Field Name**	**Attr.**	**Description**
[9:0]	`EXCCODE`	WLRL	A code identifying the last exception.
[62:10]	Reserved	WLRL	
63	Interrupt	WARL	1, if the trap was caused by an interrupt; 0 otherwise.

Table 108: Machine Cause Register

Interrupt	Exception Code	Description
1	0	Reserved
1	1	Supervisor software interrupt
1	2	Reserved
1	3	Machine software interrupt
1	4	Reserved
1	5	Supervisor timer interrupt
1	6	Reserved
1	7	Machine timer interrupt
1	8	Reserved
1	9	Supervisor external interrupt
1	10	Reserved
1	11	Machine external interrupt
1	12–13	Reserved
1	14	Debug interrupt
1	≥15	Reserved
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Debug exception
0	15	Store/AMO page fault
0	≥16	Reserved

Table 109: `mcause` Exception Codes

Note that there are scenarios where a misaligned load or store will generate an access exception instead of an address-misaligned exception. The access exception is raised when the misaligned access should not be emulated in a trap handler, e.g., emulating an access in an I/O region, as such emulation could cause undesirable side-effects.

8.7.6 Minimum Interrupt Configuration

The minimum configuration needed to configure an interrupt is shown below.

- Write `mtvec` to configure the interrupt mode and the base address for the interrupt vector table.
- Enable interrupts in memory-mapped PLIC register space. The CLINT does not contain interrupt enable bits.
- Write `mie` CSR to enable the software, timer, and external interrupt enables for each privilege mode.
- Write `mstatus` to enable interrupts globally for each supported privilege mode.

8.8 Supervisor Mode Interrupts

The U74-MC Core Complex supports the ability to selectively direct interrupts and exceptions to supervisor mode, resulting in improved performance by eliminating the need for additional mode changes.

This capability is enabled by the interrupt and exception delegation CSRs; `mideleg` and `medeleg`, respectively. Supervisor interrupts and exceptions can be managed via supervisor versions of the interrupt CSRs, specifically: `stvec`, `sip`, `sie`, and `scause`.

Machine mode software can also directly write to the `sip` register, which effectively sends an interrupt to supervisor mode. This is especially useful for timer and software interrupts as it may be desired to handle these interrupts in both machine mode and supervisor mode.

The delegation and supervisor CSRs are described in the sections below. The definitive resource for information about RISC-V supervisor interrupts is *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

8.8.1 Delegation Registers (`mideleg` and `medeleg`)

By default, all traps are handled in machine mode. Machine mode software can selectively delegate interrupts and exceptions to supervisor mode by setting the corresponding bits in `mideleg` and `medeleg` CSRs. The exact mapping is provided in Table 110 and Table 111 and matches the `mcause` interrupt and exception codes defined in Table 109.

Note that local interrupts may be delegated to supervisor mode.

Machine Interrupt Delegation Register (`mideleg`)		
CSR	`0x303`	
Bits	**Attr.**	**Description**
0	WARL	Reserved
1	RW	Delegate Supervisor Software Interrupt
[4:2]	WARL	Reserved
5	RW	Delegate Supervisor Timer Interrupt
[8:6]	WARL	Reserved
9	RW	Delegate Supervisor External Interrupt
[63:10]	WARL	Reserved

Table 110: Machine Interrupt Delegation Register

Machine Exception Delegation Register (`medeleg`)		
CSR	`0x302`	
Bits	**Attr.**	**Description**
0	RW	Delegate Instruction Access Misaligned Exception
1	RW	Delegate Instruction Access Fault Exception
2	RW	Delegate Illegal Instruction Exception
3	RW	Delegate Breakpoint Exception
4	RW	Delegate Load Access Misaligned Exception
5	RW	Delegate Load Access Fault Exception
6	RW	Delegate Store/AMO Address Misaligned Exception
7	RW	Delegate Store/AMO Access Fault Exception
8	RW	Delegate Environment Call from U-Mode
9	RW	Delegate Environment Call from S-Mode
[11:10]	WARL	Reserved
12	RW	Delegate Instruction Page Fault
13	RW	Delegate Load Page Fault
14	WARL	Reserved
15	RW	Delegate Store/AMO Page Fault Exception
[63:16]	WARL	Reserved

Table 111: Machine Exception Delegation Register

8.8.2 Supervisor Status Register (`sstatus`)

Similar to machine mode, supervisor mode has a register dedicated to keeping track of the hart's current state called `sstatus`. `sstatus` is effectively a restricted view of `mstatus`, described in Section 8.7.1, in that changes made to `sstatus` are reflected in `mstatus` and vice-versa, with the exception of the machine mode fields, which are not visible in `sstatus`.

A summary of the `sstatus` fields related to interrupts in the U74-MC Core Complex is provided in Table 112. Note that this is not a complete description of `sstatus` as it also contains fields unrelated to interrupts. For the full description of `sstatus`, consult the *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

Supervisor Status Register (`sstatus`)			
CSR	`0x100`		
Bits	**Field Name**	**Attr.**	**Description**
0	Reserved	WPRI	
1	`SIE`	RW	Supervisor Interrupt Enable
[4:2]	Reserved	WPRI	
5	`SPIE`	RW	Supervisor Previous Interrupt Enable
[7:6]	Reserved	WPRI	
8	`SPP`	RW	Supervisor Previous Privilege Mode
[12:9]	Reserved	WPRI	

Table 112: Supervisor Status Register (partial)

Interrupts are enabled by setting the `SIE` bit in `sstatus` and by enabling the desired individual interrupt in the `sie` register, described in Section 8.8.3.

8.8.3 Supervisor Interrupt Enable Register (`sie`)

Supervisor interrupts are enabled by setting the appropriate bit in the `sie` register. The U74-MC Core Complex `sie` register is described in Table 113.

Supervisor Interrupt Enable Register (`sie`)			
CSR	`0x104`		
Bits	**Field Name**	**Attr.**	**Description**
0	Reserved	WPRI	
1	`SSIE`	RW	Supervisor Software Interrupt Enable
[4:2]	Reserved	WPRI	
5	`STIE`	RW	Supervisor Timer Interrupt Enable
[8:6]	Reserved	WPRI	
9	`SEIE`	RW	Supervisor External Interrupt Enable
[63:10]	Reserved	WPRI	

Table 113: Supervisor Interrupt Enable Register

8.8.4 Supervisor Interrupt Pending (`sip`)

The supervisor interrupt pending (`sip`) register indicates which interrupts are currently pending. The U74-MC Core Complex `sip` register is described in Table 114.

Supervisor Interrupt Pending Register (`sip`)			
CSR	`0x144`		
Bits	**Field Name**	**Attr.**	**Description**
0	Reserved	WIRI	
1	`SSIP`	RW	Supervisor Software Interrupt Pending
[4:2]	Reserved	WIRI	
5	`STIP`	RW	Supervisor Timer Interrupt Pending
[8:6]	Reserved	WIRI	
9	`SEIP`	RW	Supervisor External Interrupt Pending
[63:10]	Reserved	WPRI	

Table 114: Supervisor Interrupt Pending Register

8.8.5 Supervisor Cause Register (`scause`)

When a trap is taken in supervisor mode, `scause` is written with a code indicating the event that caused the trap. When the event that caused the trap is an interrupt, the most-significant bit of `scause` is set to 1, and the least-significant bits indicate the interrupt number, using the same encoding as the bit positions in `sip`. For example, a Supervisor Timer Interrupt causes `scause` to be set to `0x8000_0000_0000_0005`.

`scause` is also used to indicate the cause of synchronous exceptions, in which case the most-significant bit of `scause` is set to 0. Refer to Table 116 for a list of synchronous exception codes.

Supervisor Cause Register (`scause`)			
CSR	`0x142`		
Bits	**Field Name**	**Attr.**	**Description**
[62:0]	`EXCCODE`	WLRL	A code identifying the last exception.
63	Interrupt	WARL	1 if the trap was caused by an interrupt; 0 otherwise.

Table 115: Supervisor Cause Register

Interrupt	Exception Code	Description
1	0	Reserved
1	1	Supervisor software interrupt
1	2–4	Reserved
1	5	Supervisor timer interrupt
1	6–8	Reserved
1	9	Supervisor external interrupt
1	≥10	Reserved
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Reserved
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9–11	Reserved
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	≥16	Reserved

Table 116: `scause` Exception Codes

8.8.6 Supervisor Trap Vector (`stvec`)

By default, all interrupts trap to a single address defined in the `stvec` register. It is up to the interrupt handler to read `scause` and react accordingly. RISC-V and the U74-MC Core Complex also support the ability to optionally enable interrupt vectors. When vectoring is enabled, each interrupt defined in `sie` will trap to its own specific interrupt handler.

Vectored interrupts are enabled when the MODE field of the `stvec` register is set to 1.

Supervisor Trap Vector Register (stvec)			
CSR	0x105		
Bits	**Field Name**	**Attr.**	**Description**
[1:0]	MODE	WARL	MODE determines whether or not interrupt vectoring is enabled. The encoding for the MODE field is described in Table 118.
[63:2]	BASE[63:2]	WARL	Interrupt Vector Base Address. Must be aligned on a 128-byte boundary when MODE=1. Note, BASE[1:0] is not present in this register and is implicitly 0.

Table 117: Supervisor Trap Vector Register

MODE Field Encoding stvec.MODE		
Value	**Name**	**Description**
0x0	Direct	All exceptions and interrupts set pc to BASE
0x1	Vectored	Exceptions set pc to BASE, interrupts set pc to BASE + 4 × scause.EXCCODE
≥0x2	Reserved	

Table 118: Encoding of stvec.MODE

If vectored interrupts are disabled (stvec.MODE=0), all interrupts trap to the stvec.BASE address. If vectored interrupts are enabled (stvec.MODE=1), interrupts set the pc to stvec.BASE + 4 × exception code (scause.EXCCODE). For example, if a supervisor timer interrupt is taken, the pc is set to stvec.BASE + 0x14. Typically, the trap vector table is populated with jump instructions to transfer control to interrupt-specific trap handlers.

In vectored interrupt mode, BASE must be 128-byte aligned.

All supervisor external interrupts (global interrupts) are mapped to exception code of 9. Thus, when interrupt vectoring is enabled, the pc is set to address stvec.BASE + 0x24 for any global interrupt.

See Table 117 for a description of the stvec register. See Table 118 for a description of the stvec.MODE field. See Table 116 for the U74-MC Core Complex supervisor mode interrupt exception code values.

8.8.7 Delegated Interrupt Handling

Upon taking a delegated trap, the following occurs:

- The value of sstatus.SIE is copied into sstatus.SPIE, then sstatus.SIE is cleared, effectively disabling interrupts.

- The current `pc` is copied into the `sepc` register, and then `pc` is set to the value of `stvec`. In the case where vectored interrupts are enabled, `pc` is set to `stvec.BASE` + 4 × exception code (`scause.EXCCODE`).
- The privilege mode prior to the interrupt is encoded in `sstatus.SPP`.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. Interrupts can be re-enabled by explicitly setting `sstatus.SIE` or by executing an `SRET` instruction to exit the handler. When an `SRET` instruction is executed, the following occurs:

- The privilege mode is set to the value encoded in `sstatus.SPP`.
- The value of `sstatus.SPIE` is copied into `sstatus.SIE`.
- The `pc` is set to the value of `sepc`.

At this point, control is handed over to software.

8.9 Interrupt Priorities

Individual priorities of global interrupts are determined by the PLIC, as discussed in Chapter 10.

U74-MC Core Complex interrupts are prioritized as follows, in decreasing order of priority:

- Machine external interrupts
- Machine software interrupts
- Machine timer interrupts
- Supervisor external interrupts
- Supervisor software interrupts
- Supervisor timer interrupts

8.10 Interrupt Latency

Interrupt latency for the U74-MC Core Complex is four `external_source_for_core_N_clock` cycles, as counted by the number of cycles it takes from signaling of the interrupt to the hart to the first instruction fetch of the handler.

Global interrupts routed through the PLIC incur additional latency of three clock cycles, where the PLIC is clocked by `clock`. This means that the total latency, in cycles, for a global interrupt is: 4 + 3 × (`external_source_for_core_N_clock` Hz ÷ `clock` Hz). This is a best-case cycle count and assumes the handler is cached or located in ITIM. It does not take into account additional latency from a peripheral source.

8.11 Non-Maskable Interrupt

The `rnmi` (resumable non-maskable interrupt) interrupt signal is a level-sensitive input to the hart. Non-maskable interrupts have higher priority than any other interrupt or exception on the hart and cannot be disabled by software. Specifically, they are not disabled by clearing the `mstatus.mie` register.

8.11.1 Handler Addresses

The NMI has an associated exception trap handler address. This address is set by external input signals, described in the U74-MC Core Complex User Guide.

8.11.2 RNMI CSRs

These M-mode CSRs enable a resumable non-maskable interrupt (RNMI).

Number	Name	Description
`0x350`	`mnscratch`	Resumable Non-maskable scratch register
`0x351`	`mnepc`	Resumable Non-maskable EPC value
`0x352`	`mncause`	Resumable Non-maskable cause value
`0x353`	`mnstatus`	Resumable Non-maskable status

Table 119: RNMI CSRs

- The `mnscratch` CSR holds a 64-bit read-write register, which enables the NMI trap handler to save and restore the context that was interrupted.
- The `mnepc` CSR is a 64-bit read-write register, which, on entry to the NMI trap handler, holds the PC of the instruction that took the interrupt. The lowest bit of `mnepc` is hardwired to zero.
- The `mncause` CSR holds the reason for the NMI, with bit 63 set to 1, and the NMI cause encoded in the least-significant bits, or zero if NMI causes are not supported. The lower bits of `mncause`, defined as the `exception_code`, are as follows:

`mncause`	NMI Cause	Function
1	Reserved	Reserved
2	RNMI input pin	External `rnmi_N` input
3	Bus error	RNMI caused by BEU

Table 120: `mncause.exception_code` Fields

- The `mnstatus` CSR holds a two-bit field `mnpp` encoded in the same manner as `mstatus.mpp`, which, on entry to the trap handler, holds the privilege mode of the interrupted context.

- `mnstatus` also hold the one-bit field `mnie` indicating whether NMIs are currently enabled. This bit can only be cleared by hardware, but can be set by software to indicate a further NMI can be taken.

Bits	Field Name	Description
3	`mnie`	RMNI interrupt enable
7	`mnpv`	Hardwired to zero
12:11	`mnpp`	RMNI previous priority level

Table 121: `mnstatus` CSR Fields

8.11.3 MNRET Instruction

This M-mode only instruction uses the values in `mnepc` and `mnstatus` to return to the program counter and privileged mode of the interrupted context. This instruction also sets the internal `rnmie` state bits.

Encoding is same as MRET except with bit 30 set (i.e., `funct7=0111000`). For example:

```
.word 0x70200073      // opcode for MNRET (return from RNMI)
```

8.11.4 RNMI Operation

When an RNMI interrupt is detected, the interrupted PC is written to the `mnepc` CSR, the type of RNMI to the `mncause` CSR, and the privilege mode of the interrupted context to the `mnstatus` CSR. An internal microarchitectural state bit, `rnmie`, is cleared to indicate that the processor is in an RNMI handler and cannot take a new RNMI interrupt. When clear, the internal `rnmie` bit also disables all other interrupts.

> **Note**
>
> These interrupts are called non-maskable because software cannot mask the interrupts. However, for correct operation, other instances of the same interrupt must be held off until the handler is completed, hence the internal state bit.

The RNMI handler can resume original execution using the MNRET instruction (described in Section 8.11.3), which restores the PC from `mnepc`, the privilege mode from `mnstatus`, and also sets the internal `rnmie` state bit, which re-enables other interrupts.

If the hart encounters an exception while the `rnmie` bit is clear, the exception state is written to `mepc` and `mcause`, `mstatus.mpp` is set to M-mode, and the hart jumps to the RNMI exception handler address.

> **Note**
>
> Traps in the RNMI handler can only be resumed if they occur while the handler was servicing an interrupt that occurred outside of machine mode.

Chapter 9

Core-Local Interruptor (CLINT)

This chapter describes the operation of the Core-Local Interruptor (CLINT). The U74-MC Core Complex CLINT complies with *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

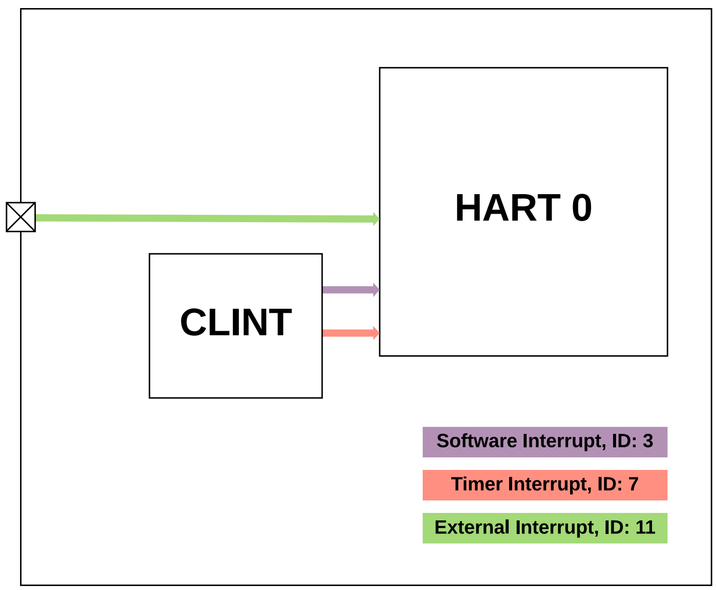

Figure 114: CLINT Block Diagram

The CLINT has a small footprint and provides software, timer, and external interrupts directly to the hart. The CLINT block also holds memory-mapped control and status registers associated with software and timer interrupts.

9.1 CLINT Priorities and Preemption

The CLINT has a fixed priority scheme, described in Section 8.9, and nested interrupts (preemption) within a given privilege level is not supported. Higher privilege levels may preempt lower privilege levels, however. The CLINT offers two modes of operation, Direct mode and Vectored mode.

In Direct mode, all interrupts and exceptions trap to `mtvec.BASE`. In Vectored mode, exceptions trap to `mtvec.BASE`, but interrupts will jump directly to their vector table index. See Section 8.7.2 for more information about `mtvec.BASE`.

9.2 CLINT Vector Table

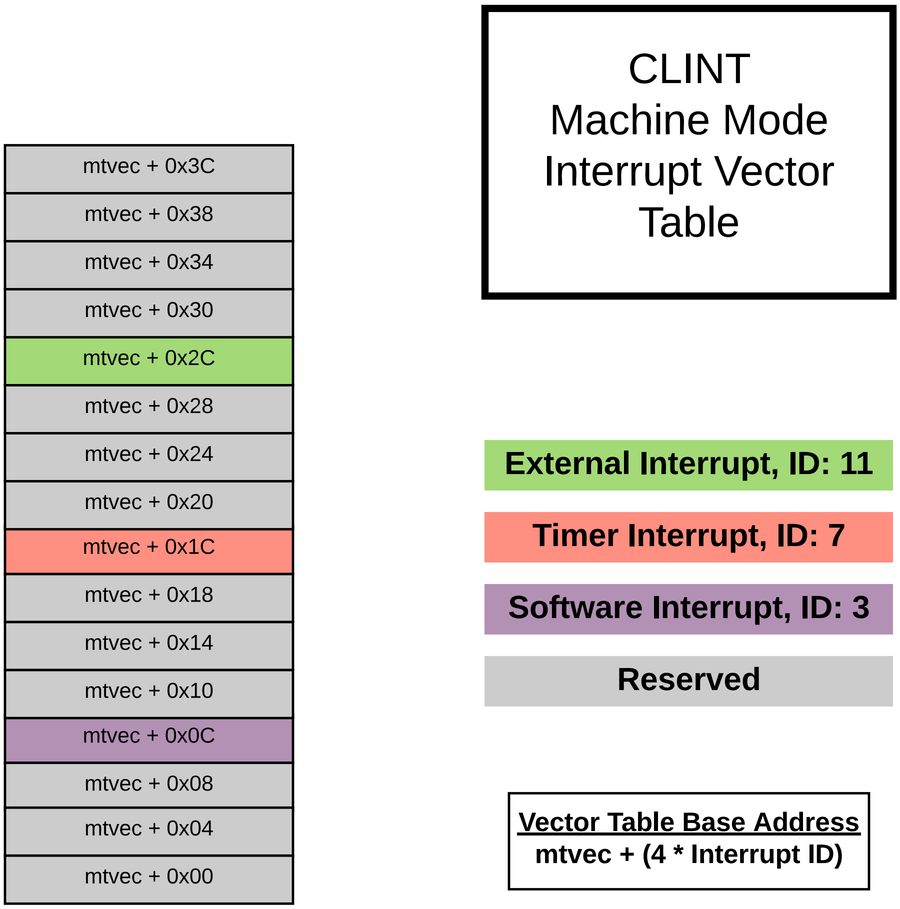

Figure 115: CLINT Interrupts and Vector Table

The CLINT vector table is populated with jump instructions, since hardware jumps to the index in the vector table first, then subsequently jumps to the handler. All exception types trap to the first entry in the table, which is `mtvec.BASE`.

An example CLINT vector table is shown below.

```
.weak default_exception_handler
.balign 4, 0
.global default_exception_handler

.weak software_handler
.balign 4, 0
.global software_handler

.weak timer_handler
.balign 4, 0
.global timer_handler

.weak external_handler
.balign 4, 0
.global external_handler

.option norvc
.weak __mtvec_clint_vector_table
#if __riscv_xlen == 32
.balign 128, 0
#else
.balign 256, 0
#endif
.global __mtvec_clint_vector_table
__mtvec_clint_vector_table:

IRQ_0:
        j default_exception_handler
IRQ_1:
        j default_vector_handler
IRQ_2:
        j default_vector_handler
IRQ_3:
        j software_handler
IRQ_4:
        j default_vector_handler
IRQ_5:
        j default_vector_handler
IRQ_6:
        j default_vector_handler
IRQ_7:
        j timer_handler
IRQ_8:
        j default_vector_handler
IRQ_9:
        j default_vector_handler
IRQ_10:
        j default_vector_handler
IRQ_11:
        j external_handler
IRQ_12:
        j default_vector_handler
IRQ_13:
        j default_vector_handler
IRQ_14:
        j default_vector_handler
IRQ_15:
        j default_vector_handler
```

Figure 116: CLINT Vector Table Example

9.3 CLINT Interrupt Sources

The U74-MC Core Complex supports the standard RISC-V software, timer, and external interrupts.

CLINT Interrupt IDs are provided in Table 122.

U74-MC Core Complex Interrupt IDs		
ID	**Interrupt**	**Notes**
0	Reserved	
1	ssip	Supervisor Software Interrupt
2	Reserved	
3	msip	Machine Software Interrupt
4	Reserved	
5	stip	Supervisor Timer Interrupt
6	Reserved	
7	mtip	Machine Timer Interrupt
8	Reserved	
9	seip	Supervisor External Interrupt
10	Reserved	
11	meip	Machine External Interrupt

Table 122: U74-MC Core Complex Interrupt IDs

9.4 CLINT Interrupt Attribute

To help with efficiency of save and restore context, interrupt attributes can be applied to functions used for interrupt handling.

```
void __attribute__((interrupt))
software_handler (void) {
  // handler code
}
```

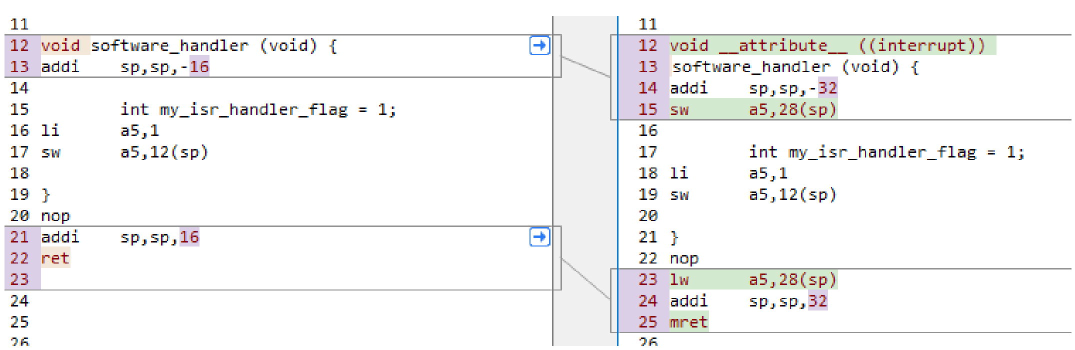

Figure 117: CLINT Interrupt Attribute Example

This attribute will save and restore registers that are used within the handler, and insert an `mret` instruction at the end of the handler.

9.5 CLINT Memory Map

Table 123 shows the memory map for CLINT on the U74-MC Core Complex. Note that there are no enable bits for specific interrupts within the CLINT memory map, as the enables for these interrupts reside in the `mie` CSR for each interrupt, and the `mstatus.mie` CSR bit, which enables all machine interrupts globally. See Section 8.7.3 for a description of the interrupt enable bits in the `mie` CSR, and Section 8.7.4 for a description of the interrupt pending bits in the `mip` CSR.

Address	Width	Attr.	Description	Notes
`0x0200_0000`	4B	RW	`msip` for hart 0	MSIP Registers (1-bit wide)
`0x0200_0004`	4B	RW	`msip` for hart 1	
`0x0200_0008`	4B	RW	`msip` for hart 2	
`0x0200_000C`	4B	RW	`msip` for hart 3	
`0x0200_0010`	4B	RW	`msip` for hart 4	
`0x0200_0014`			Reserved	
…				
`0x0200_3FFF`				
`0x0200_4000`	8B	RW	`mtimecmp` for hart 0	MTIMECMP Registers
`0x0200_4008`	8B	RW	`mtimecmp` for hart 1	
`0x0200_4010`	8B	RW	`mtimecmp` for hart 2	
`0x0200_4018`	8B	RW	`mtimecmp` for hart 3	
`0x0200_4020`	8B	RW	`mtimecmp` for hart 4	
`0x0200_4028`			Reserved	
…				
`0x0200_BFF7`				
`0x0200_BFF8`	8B	RW	`mtime`	Timer Register
`0x0200_C000`			Reserved	

Table 123: CLINT Memory Map

9.6 Register Descriptions

This section describes the functionality of the memory-mapped registers in the CLINT.

9.6.1 MSIP Register

Machine mode software interrupts are generated by writing to the memory-mapped control register `msip`. Each `msip` register is a 32-bit wide **WARL** register, where the upper 31 bits are tied to 0. The least-significant bit is reflected in the `MSIP` bit of the `mip` CSR. Other bits in the `msip` register are hardwired to zero. On reset, each `msip` register is cleared to zero.

Software interrupts are most useful for interprocessor communication in multi-hart systems, as harts may write each other's `msip` bits to effect interprocessor interrupts.

9.6.2 Timer Registers

`mtime` is a 64-bit read-write register that contains the number of cycles counted from the `rtc_toggle` signal, which is described in the U74-MC Core Complex User Guide. A timer interrupt is pending whenever `mtime` is greater than or equal to the value in the `mtimecmp` register. The timer interrupt is reflected in the `mtip` bit of the `mip` register, described in Chapter 8.

On reset, `mtime` is cleared to zero. The `mtimecmp` registers are not reset.

Note that `mtime` is volatile and may be masked in IP-XACT.

9.7 Supervisor Mode Delegation

By default, all interrupts trap to machine mode, including timer and software interrupts. In order for supervisor timer and software interrupts to trap directly to supervisor mode, supervisor timer and software interrupts must first be delegated to supervisor mode.

Please see Section 8.8 for more details on supervisor mode interrupts.

Chapter 10

Platform-Level Interrupt Controller (PLIC)

This chapter describes the operation of the Platform-Level Interrupt Controller (PLIC) on the U74-MC Core Complex. The PLIC complies with *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* and can support a maximum of 136 external interrupt sources with 7 priority levels.

The U74-MC Core Complex PLIC resides in the `clock` timing domain, allowing for relaxed timing requirements. The latency of global interrupts, as perceived by a hart, increases with the ratio of the `external_source_for_core_N_clock` frequency and the `clock` frequency.

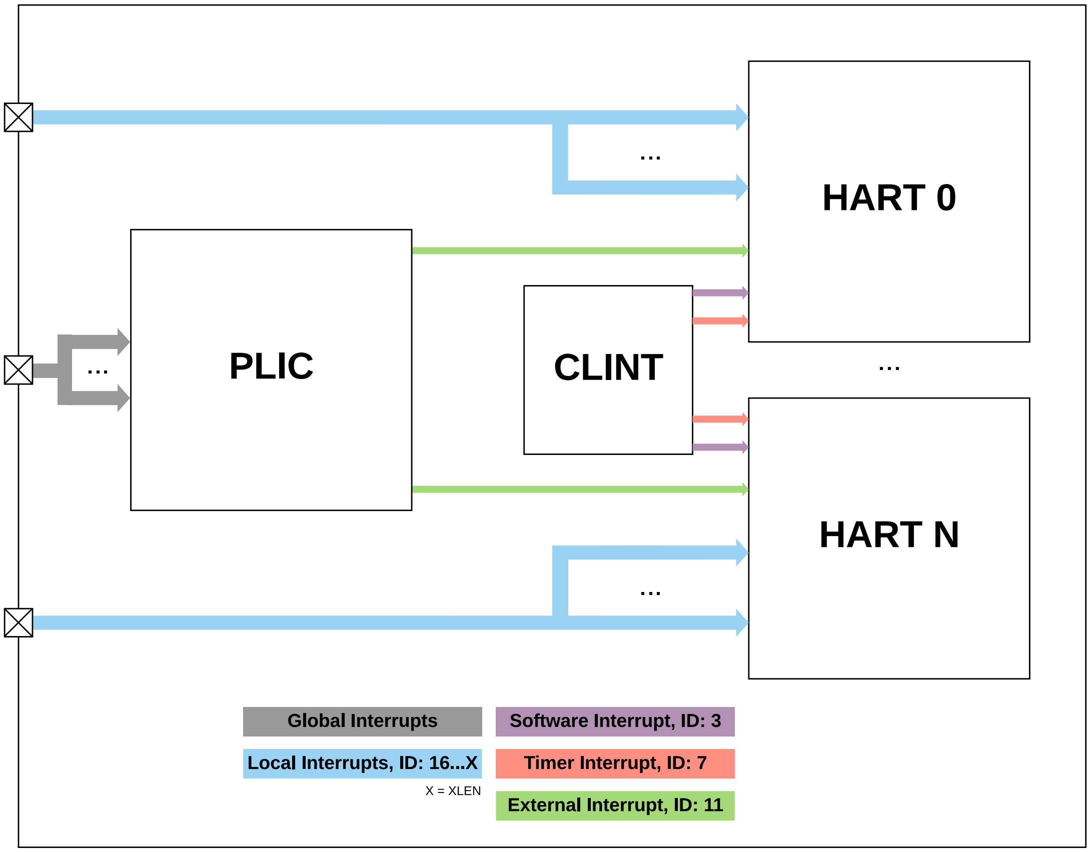

Figure 118: PLIC Multi-Core Block Diagram

10.1 Memory Map

The memory map for the U74-MC Core Complex PLIC control registers is shown in Table 124. The PLIC memory map only supports aligned 32-bit memory accesses.

Address	Width	Attr.	Description	Notes
0x0C00_0000			Reserved	
0x0C00_0004	4B	RW	Source 1 priority	See Section 10.3 for more information
…				
0x0C00_0220	4B	RW	Source 136 priority	
0x0C00_0224			Reserved	
…				
0x0C00_1000	4B	RO	Start of pending array	See Section 10.4 for more information
…				
0x0C00_1010	4B	RO	Last word of pending array	
0x0C00_1014			Reserved	
…				
0x0C00_2000	4B	RW	Start Hart 0 M-Mode interrupt enables	See Section 10.5 for more information
…				
0x0C00_2010	4B	RW	End Hart 0 M-Mode interrupt enables	
0x0C00_2014			Reserved	
…				
0x0C00_2080	4B	RW	Start Hart 1 M-Mode interrupt enables	See Section 10.5 for more information
…				
0x0C00_2090	4B	RW	End Hart 1 M-Mode interrupt enables	
0x0C00_2094			Reserved	
…				
0x0C00_2100	4B	RW	Start Hart 1 S-Mode interrupt enables	See Section 10.5 for more information
…				
0x0C00_2110	4B	RW	End Hart 1 S-Mode interrupt enables	
0x0C00_2114			Reserved	
…				
0x0C00_2180	4B	RW	Start Hart 2 M-Mode interrupt enables	See Section 10.5 for more information
…				
0x0C00_2190	4B	RW	End Hart 2 M-Mode interrupt enables	
0x0C00_2194			Reserved	
…				
0x0C00_2200	4B	RW	Start Hart 2 S-Mode interrupt enables	See Section 10.5 for more information
…				

Table 124: PLIC Memory Map

Address	Width	Attr.	Description	Notes
`0x0C00_2210`	4B	RW	End Hart 2 S-Mode interrupt enables	
`0x0C00_2214`			Reserved	
…				
`0x0C00_2280`	4B	RW	Start Hart 3 M-Mode interrupt enables	See Section 10.5 for more information
…				
`0x0C00_2290`	4B	RW	End Hart 3 M-Mode interrupt enables	
`0x0C00_2294`			Reserved	
…				
`0x0C00_2300`	4B	RW	Start Hart 3 S-Mode interrupt enables	See Section 10.5 for more information
…				
`0x0C00_2310`	4B	RW	End Hart 3 S-Mode interrupt enables	
`0x0C00_2314`			Reserved	
…				
`0x0C00_2380`	4B	RW	Start Hart 4 M-Mode interrupt enables	See Section 10.5 for more information
…				
`0x0C00_2390`	4B	RW	End Hart 4 M-Mode interrupt enables	
`0x0C00_2394`			Reserved	
…				
`0x0C00_2400`	4B	RW	Start Hart 4 S-Mode interrupt enables	See Section 10.5 for more information
…				
`0x0C00_2410`	4B	RW	End Hart 4 S-Mode interrupt enables	
`0x0C00_2414`			Reserved	
…				
`0x0C1F_F000`	1B	RW	PLIC global clock gating disable feature	See Section 10.6 for more information
`0x0C1F_F001`			Reserved	
…				
`0x0C20_0000`	4B	RW	Hart 0 M-Mode priority threshold	See Section 10.7 for more information
`0x0C20_0004`	4B	RW	Hart 0 M-Mode claim/complete	See Section 10.8 for more information
`0x0C20_0008`			Reserved	
…				

Table 124: PLIC Memory Map

Address	Width	Attr.	Description	Notes
0x0C20_1000	4B	RW	Hart 1 M-Mode priority threshold	See Section 10.7 for more information
0x0C20_1004	4B	RW	Hart 1 M-Mode claim/ complete	See Section 10.8 for more information
0x0C20_1008			Reserved	
…				
0x0C20_2000	4B	RW	Hart 1 S-Mode priority threshold	See Section 10.7 for more information
0x0C20_2004	4B	RW	Hart 1 S-Mode claim/ complete	See Section 10.8 for more information
0x0C20_2008			Reserved	
…				
0x0C20_3000	4B	RW	Hart 2 M-Mode priority threshold	See Section 10.7 for more information
0x0C20_3004	4B	RW	Hart 2 M-Mode claim/ complete	See Section 10.8 for more information
0x0C20_3008			Reserved	
…				
0x0C20_4000	4B	RW	Hart 2 S-Mode priority threshold	See Section 10.7 for more information
0x0C20_4004	4B	RW	Hart 2 S-Mode claim/ complete	See Section 10.8 for more information
0x0C20_4008			Reserved	
…				
0x0C20_5000	4B	RW	Hart 3 M-Mode priority threshold	See Section 10.7 for more information
0x0C20_5004	4B	RW	Hart 3 M-Mode claim/ complete	See Section 10.8 for more information
0x0C20_5008			Reserved	
…				
0x0C20_6000	4B	RW	Hart 3 S-Mode priority threshold	See Section 10.7 for more information
0x0C20_6004	4B	RW	Hart 3 S-Mode claim/ complete	See Section 10.8 for more information
0x0C20_6008			Reserved	
…				
0x0C20_7000	4B	RW	Hart 4 M-Mode priority threshold	See Section 10.7 for more information
0x0C20_7004	4B	RW	Hart 4 M-Mode claim/ complete	See Section 10.8 for more information
0x0C20_7008			Reserved	
…				

Table 124: PLIC Memory Map

Address	Width	Attr.	Description	Notes
0x0C20_8000	4B	RW	Hart 4 S-Mode priority threshold	See Section 10.7 for more information
0x0C20_8004	4B	RW	Hart 4 S-Mode claim/ complete	See Section 10.8 for more information
0x0C20_8008			Reserved	
…				
0x1000_0000			End of PLIC Memory Map	

Table 124: PLIC Memory Map

10.2 Interrupt Sources

The U74-MC Core Complex has a total of 136 global interrupt sources, in addition to the local interrupts described in Table 122. 127 of these are external global interrupts, and the remainder are driven by various on-chip devices as listed in Table 125.

Note

In the *RISC-V Platform-Level Interrupt Controller Specification*, interrupt source 0 (ID 0) is unused, so the first usable PLIC Interrupt ID has a value of 1.

Table 125 describes the global interrupt sources on the U74-MC Core Complex.

PLIC Interrupt ID	Source
0	*Unused*
1	L2 Cache `DirError`
2	L2 Cache `DirFail`
3	L2 Cache `DataError`
4	L2 Cache `DataFail`
5–131	External Global Interrupts
132	S7 Hart 0 Bus-Error Unit
133	U7 Hart 1 Bus-Error Unit
134	U7 Hart 2 Bus-Error Unit
135	U7 Hart 3 Bus-Error Unit
136	U7 Hart 4 Bus-Error Unit

Table 125: PLIC Interrupt Source Mapping

Table 126 describes the mapping of external global interrupts to its corresponding top-level `global_interrupts` signal bit. This signal is positive-level triggered and not configurable. See the U74-MC Core Complex User Guide for further description of `global_interrupts`.

global_interrupts Signal	PLIC Interrupt ID	PLIC Pending / Enable Register
global_interrupts[0]	5	pending1[5] / enable1[5]
global_interrupts[1]	6	pending1[6] / enable1[6]
global_interrupts[2]	7	pending1[7] / enable1[7]
…		
global_interrupts[126]	131	pending5[3] / enable5[3]

Table 126: Mapping of global_interrupts Signal Bits to PLIC Interrupt ID

10.3 Interrupt Priorities

Each PLIC interrupt source can be assigned a priority by writing to its 32-bit memory-mapped priority register. The U74-MC Core Complex supports 7 levels of priority. A priority value of 0 is reserved to mean "never interrupt" and effectively disables the interrupt. Priority 1 is the lowest active priority, and priority 7 is the highest. Ties between global interrupts of the same priority are broken by the Interrupt ID; interrupts with the lowest ID have the highest effective priority. See Table 127 for the detailed register description.

PLIC Interrupt Priority Register				
Base Address		0x0C00_0000 + 4 × Interrupt ID		
Bits	Field Name	Attr.	Rst.	Description
[2:0]	Priority	RW	X	Global interrupt priority
[31:3]	Reserved	RO	0x0	

Table 127: PLIC Interrupt Priority Register

10.4 Interrupt Pending Bits

The current status of the interrupt source pending bits in the PLIC core can be read from the pending array, organized as 5 words of 32 bits. The pending bit for interrupt ID *N* is stored in bit (*N* mod 32) of word (*N* / 32). As such, the U74-MC Core Complex has five interrupt pending registers. Bit 0 of word 0, which represents the non-existent interrupt source 0, is hardwired to zero.

A pending bit in the PLIC core can be cleared by setting the associated enable bit then performing a claim as described in Section 10.8.

PLIC Interrupt Pending Register 1				
Base Address		0x0C00_1000		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	Interrupt 0 Pending	RO	X	Non-existent global interrupt 0 is hardwired to zero
1	Interrupt 1 Pending	RO	X	Pending bit for global interrupt 1
2	Interrupt 2 Pending	RO	X	Pending bit for global interrupt 2
…				
31	Interrupt 31 Pending	RO	X	Pending bit for global interrupt 31

Table 128: PLIC Interrupt Pending Register 1

PLIC Interrupt Pending Register 5				
Base Address		0x0C00_1010		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	Interrupt 128 Pending	RO	X	Pending bit for global interrupt 128
…				
8	Interrupt 136 Pending	RO	X	Pending bit for global interrupt 136
[31:9]	Reserved	WIRI	X	

Table 129: PLIC Interrupt Pending Register 5

10.5 Interrupt Enables

Each global interrupt can be enabled by setting the corresponding bit in the `enable` registers. The `enable` registers are accessed as a contiguous array of 5 × 32-bit words, packed the same way as the `pending` bits. Bit 0 of enable word 0 represents the non-existent interrupt ID 0 and is hardwired to 0.

64-bit and 32-bit word accesses are supported by the `enables` array in SiFive RV64 systems.

PLIC Interrupt Enable Register 1 for Hart 0 M-Mode				
Base Address		0x0C00_2000		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	Interrupt 0 Enable	RO	0x0	Non-existent global interrupt 0 is hardwired to zero
1	Interrupt 1 Enable	RW	X	Enable bit for global interrupt 1
2	Interrupt 2 Enable	RW	X	Enable bit for global interrupt 2
…				
31	Interrupt 31 Enable	RW	X	Enable bit for global interrupt 31

Table 130: PLIC Interrupt Enable Register 1 for Hart 0 M-Mode

PLIC Interrupt Enable Register 5 for Hart 0 M-Mode				
Base Address		0x0C00_2010		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	Interrupt 128 Enable	RW	X	Enable bit for global interrupt 128
…				
8	Interrupt 136 Enable	RW	X	Enable bit for global interrupt 136
[31:9]	Reserved	RO	0x0	

Table 131: PLIC Interrupt Enable Register 5 for Hart 0 M-Mode

PLIC Interrupt Enable Register 1 for Hart 1 S-Mode				
Base Address		0x0C00_2100		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	Interrupt 0 Enable	RO	0x0	Non-existent global interrupt 0 is hardwired to zero
1	Interrupt 1 Enable	RW	X	Enable bit for global interrupt 1
2	Interrupt 2 Enable	RW	X	Enable bit for global interrupt 2
…				
31	Interrupt 31 Enable	RW	X	Enable bit for global interrupt 31

Table 132: PLIC Interrupt Enable Register 1 for Hart 1 S-Mode

PLIC Interrupt Enable Register 5 for Hart 1 S-Mode				
Base Address		`0x0C00_2110`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	Interrupt 128 Enable	RW	X	Enable bit for global interrupt 128
…				
8	Interrupt 136 Enable	RW	X	Enable bit for global interrupt 136
[31:9]	Reserved	RO	`0x0`	

Table 133: PLIC Interrupt Enable Register 5 for Hart 1 S-Mode

10.6 PLIC Clock Gate Disable

The PLIC implements a clock gating feature to gate the module clock node when not active. PLIC clock gating is disabled out of reset and should be enabled in startup code, unless otherwise specified by SiFive erratum. Once enabled, clock is only available when there is activity on the PLIC control bus or on any interrupt line when the corresponding interrupt is not inflight. Clock gating is further described in the U74-MC Core Complex User Guide.

PLIC Clock Gate Disable Register				
Base Address		`0x0C1F_F000`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	`disablePlicClockGateFeature`	RW	`0x1`	Used to enable/disable PLIC clock gating feature. Clear to enable.
[7:1]	Reserved	RO	`0x0`	

Table 134: PLIC Clock Gate Disable Register

10.7 Priority Thresholds

The U74-MC Core Complex supports setting of an interrupt priority threshold via the `threshold` register. The `threshold` is a **WARL** field, where the U74-MC Core Complex supports a maximum threshold of 7.

The U74-MC Core Complex masks all PLIC interrupts of a priority less than or equal to `threshold`. For example, a `threshold` value of zero permits all interrupts with non-zero priority, whereas a value of 7 masks all interrupts. If the `threshold` register contains a value of 5, all PLIC interrupt configured with priorities from 1 through 5 will not be allowed to propagate to the CPU.

PLIC Interrupt Priority Threshold Register for Hart 0 M-Mode				
Base Address		`0x0C20_0000`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[2:0]	Threshold	WARL	`X`	Sets the priority threshold
[31:3]	Reserved	RO	`0x0`	

Table 135: PLIC Interrupt Priority Threshold Register for Hart 0 M-Mode

PLIC Interrupt Priority Threshold Register for Hart 1 S-Mode				
Base Address		`0x0C20_2000`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[2:0]	Threshold	WARL	`X`	Sets the priority threshold
[31:3]	Reserved	RO	`0x0`	

Table 136: PLIC Interrupt Priority Threshold Register for Hart 1 S-Mode

10.8 Interrupt Claim Process

A U74-MC Core Complex hart can perform an interrupt claim by reading the `claim_complete` register (Table 137), which returns the ID of the highest-priority pending interrupt or zero if there is no pending interrupt. A successful claim also atomically clears the corresponding pending bit on the interrupt source.

A U74-MC Core Complex hart can perform a claim at any time, even if the `MEIP` bit in its `mip` (Table 107) register is not set.

The claim operation is not affected by the setting of the priority threshold register.

10.9 Interrupt Completion

A U74-MC Core Complex hart signals it has completed executing an interrupt handler by writing the interrupt ID it received from the claim to the `claim_complete` register (Table 137). The PLIC does not check whether the completion ID is the same as the last claim ID for that target. If the completion ID does not match an interrupt source that is currently enabled for the target, the completion is silently ignored.

PLIC Claim/Complete Register for Hart 0 M-Mode				
Base Address		`0x0C20_0004`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[31:0]	Interrupt Claim/ Complete for Hart 0 M-Mode	RW	X	A read of zero indicates that no interrupts are pending. A non-zero read contains the ID of the highest pending interrupt. A write to this register signals completion of the interrupt ID written. Consecutive reads will not return the same result. Note that this field is volatile and may be masked in IP-XACT.

Table 137: PLIC Claim/Complete Register for Hart 0 M-Mode

PLIC Claim/Complete Register for Hart 1 S-Mode				
Base Address		`0x0C20_2004`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[31:0]	Interrupt Claim/ Complete for Hart 1 S-Mode	RW	X	A read of zero indicates that no interrupts are pending. A non-zero read contains the ID of the highest pending interrupt. A write to this register signals completion of the interrupt ID written. Consecutive reads will not return the same result. Note that this field is volatile and may be masked in IP-XACT.

Table 138: PLIC Claim/Complete Register for Hart 1 S-Mode

The PLIC cannot forward a new interrupt to a hart that has claimed an interrupt, but has not yet finished the complete step of the interrupt handler. Thus, the PLIC does not support preemption of global interrupts to an individual hart.

Interrupt IDs for global interrupts routed through the PLIC are independent of the interrupt IDs for local interrupts. The PLIC handler may check for additional pending global interrupts once the initial claim/complete process has finished, prior to exiting the handler. This method could save additional PLIC save/restore context for global interrupts.

10.10 Example PLIC Interrupt Handler

Since the PLIC interfaces with the CPU through external interrupt #11, the external handler must contain an additional claim/complete step that is used to handshake with the PLIC logic.

```
void external_handler() {
  //get the highest priority pending PLIC interrupt
  uint32_t int_num = plic.claim_complete;

  //branch to handler
  plic_handler[int_num]();

  //complete interrupt by writing interrupt number back to PLIC
  plic.claim_complete = int_num;

  // Add additional checks for PLIC pending here, if desired
}
```

If a CPU reads `claim_complete` and it returns 0, the interrupt does not require processing, and thus write-back of the claim/complete is not necessary.

The `plic_handler[]()` routine shown above demonstrates one method to implement a software table where the offset of the function that resides within the table is determined by the PLIC interrupt ID. The PLIC interrupt ID is unique to the PLIC, in that it is completely independent of the interrupt IDs of local interrupts.

Chapter 11

TileLink Error Device

The Error Device is a TileLink slave that responds to all requests with a TileLink `denied` error and all reads with a `corrupt` error. It has no registers. The entire memory range discards writes and returns zeros on read. Both operation acknowledgements carry an error indication.

The Error Device serves a dual role. Internally, it is used as a landing pad for illegal off-chip requests. However, it is also useful for testing software handling of bus errors.

Chapter 12

Bus-Error Unit

This chapter describes the operation of the SiFive Bus-Error Unit (BEU).

12.1 Bus-Error Unit Overview

The Bus-Error Unit is a per-processor device that records erroneous events and reports them using platform-level and hart-local interrupts. Figure 119 shows the connections from the core to the BEU.

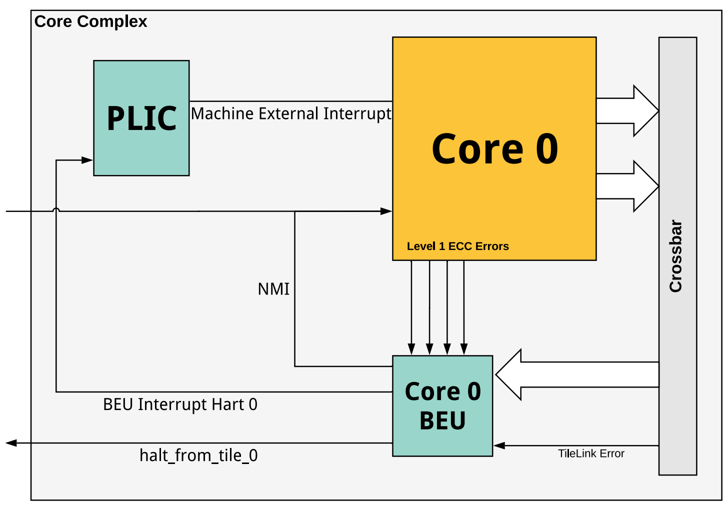

Figure 119: Bus-Error Unit Block Diagram

The BEU can be configured to generate interrupts on correctable memory errors, uncorrectable memory errors, and/or TileLink bus errors. When an error occurs, the BEU will hold the address of the error and a code to describe the error.

12.2 Memory Map

The Bus-Error Unit memory map is shown in Table 139.

Offset	Name	Description
`0x00`	`cause`	Cause of error event
`0x08`	`value`	Physical address of error event
`0x10`	`enable`	Event enable mask
`0x18`	`plic_interrupt`	Platform-level interrupt enable mask
`0x20`	`accrued`	Accrued event mask
`0x28`	`local_interrupt`	Hart-local interrupt-enable mask

Table 139: Bus-Error Unit Memory Map

Each bit in `enable`, `plic_interrupt`, `accrued`, and `local_interrupt` is associated with an error event, described in Table 140. The `cause` register represents which event occurred, and the `value` register contains the address of that error event. The `value` register will be updated when errors occur on memory that supports data cache. For example, the Memory Port has data cache support, where the System and Peripheral Ports do not.

12.3 Reportable Errors

Table 140 lists the events that the Bus-Error Unit may report.

Value	Description
0	*No error*
1	Instruction cache TileLink bus error
2	Instruction cache or ITIM correctable ECC error
3	ITIM uncorrectable ECC error
4	Reserved
5	Load/Store/PTW TileLink bus error
6	Data cache correctable ECC error
7	Data cache uncorrectable ECC error

Table 140: Bus-Error Unit Error Events

12.4 Functional Description

When one of the events listed in Table 140 occurs, the Bus-Error Unit can record information about that event and can generate an interrupt to a platform-level interrupt controller (i.e., the PLIC) or locally to the hart. The `enable` register contains a mask of which events the BEU can record. Each bit in `enable` corresponds to an event in Table 140. For example, if `enable[5]` is set, the BEU will record load, store, or PTW TileLink bus errors. If `plic_interrupt[5]` is also set, then a global interrupt will be asserted to the platform-level interrupt controller. Alternatively, if `local_interrupt[5]` is set, then a BEU error will result in a local interrupt to the hart. The `cause` register indicates the event the BEU has most recently recorded, e.g., a value of 5 indicates a load, store, or PTW TileLink bus error was recorded.

The `cause` value 0 is reserved to indicate "no error". `cause` is only written for events enabled in the `enable` register. Furthermore, `cause` is only written when its current value is 0; that is, if multiple events occur, only the first one is latched, until software clears the `cause` register.

The `value` register supplies the physical address that caused the event, or 0 if the address is unknown. The BEU writes the `value` register whenever it writes the `cause` register, such as when an event enabled in the `enable` register occurs or when `cause` contains 0.

The bit position in the `accrued` register indicates which events have occurred since the last time it was cleared by software. Its format is the same as the `enable` register. The BEU sets bits in the `accrued` register whether or not they are enabled in the `enable` register.

12.4.1 BEU Global Interrupt

The bit position in the `plic_interrupt` register indicates which accrued events should generate an interrupt into the PLIC. An interrupt is generated when the same bit is set in both `accrued` and `plic_interrupt`. The PLIC drives the machine external interrupt to the core, which has an interrupt ID of 11 (`0xB`). The exception code value, located in `mcause` (machine trap cause) CSR, will be 11 (`0xB`) when BEU interrupts are routed through the PLIC. This exception code value is independent of the PLIC interrupt number used to connect the BEU to the PLIC.

12.4.2 BEU Local Interrupt

The BEU output signal connects to the non-maskable interrupt (NMI) logic, described in Section 8.11. The `local_interrupt` register indicates which accrued events will cause a trap to the RNMI interrupt vector address. Within the NMI handler, the `nmcause` register can be read to determine if it was a BEU error, or an error triggered directly via the NMI `rnmi_N` input signal.

12.4.3 Global Interrupt Configuration

In addition to writing the BEU registers to enable interrupts, the `mstatus.MIE` CSR bit should be written to enable Machine level interrupts globally. The `mie.MEIE` CSR bit should also be configured to enable external interrupts when `plic_interrupt` is enabled to route the PLIC interrupt from the BEU interrupts to the core. Likewise, the corresponding PLIC Hart N M-Mode interrupt enable register should be enabled, and the PLIC Hart N interrupt priority threshold must be set. Additionally, the PLIC interrupt priority register must be set for the BEU's corresponding PLIC interrupt ID. Refer to the local interrupt chapter for more details regarding interrupt configurations.

Chapter 13

Level 2 Prefetcher

The SiFive L2 Prefetcher is a per-hart device that allows the L2 to perform accesses to memory based on the patterns of data accesses made by the harts in the Core Complex. For instance, if a hart is reading every 100th byte of a large array and accesses are missing the L1 data cache, the prefetcher will detect this. It will then allocate the appropriate memory addresses to the L2 Cache so subsequent accesses to the array will hit in the L2 Cache. This reduces overall access time to the array and improves performance of the application.

The U74-MC Core Complex contains four L2 Prefetcher instances. Their addresses are shown in Table 141.

Instance	Base Address
U7 Hart 1 L2 Prefetcher	`0x0203_0000`
U7 Hart 2 L2 Prefetcher	`0x0203_2000`
U7 Hart 3 L2 Prefetcher	`0x0203_4000`
U7 Hart 4 L2 Prefetcher	`0x0203_6000`

Table 141: L2 Prefetcher Instances

13.1 Operation

The L2 Prefetcher can monitor eight different data streams per hart, where a stream consists of a base address and a stride between memory addresses. Prefetches are automatically issued to the memory system when possible and a prefetch queue of 10 entries holds the accesses before they are issued. The range of a prefetch (or maximum stride length) in terms of cache lines is set by the `additionalCtrl.window` register. For instance, if this is set to `0x4`, then streams with a stride of 256 bytes or greater will be ignored (assuming each cache line is 64 bytes).

The initial number of prefetches (or prefetch distance) made per stream is set by `basicCtrl.initialDist`. The prefetching distance will adapt based on the success of the prefetching and the overall range of prefetching. If a hart continues to make direct accesses which match the stride of the stream (both in magnitude and sign), then additional prefetches will be made, and the prefetch distance can increase. It’s possible that the direct accesses made by a hart will be too fast for the prefetcher, and the hart will have to wait for the L2 to populate. When this occurs, the L2 Prefetcher will increase the prefetch distance (i.e., emit more prefetches) to minimize the chance of this occurring. The threshold to increase the prefetching distance is set by `additionalCtrl.hitMSHRThrd`.

The maximum number of prefetches emitted for a stream is limited by `basicCtrl.maxAllowedDist`. The prefetch distance will increase gradually, but at times this may not be fast enough. The speed of the ramp-up from the initial distance to the maximum is managed automatically by the prefetcher. The `basicCtrl.linToExpThrd` setting can be used to fine-tune this adaptive ramp-up. The smaller the value of this field, the faster the prefetcher will reach the maximum prefetch distance.

Prefetches are issued only if there are idle L2 Miss Status Holding Registers (MSHRs). The `additionalCtrl.qFullnessThrd` value is used to control when the prefetcher will stop issuing hints to the memory system. This 4-bit field allows the total number of MSHRs in the L2 Cache to be represented in fractions of 1/16th. For instance, a threshold value of `0xC` means that if 75% of the MSHRs are allocated, prefetches will be stalled until the number of MSHRs in use drops below 75% available. This setting allows the prefetcher to be tuned such that the direct accesses made by the Core Complex (from harts and Front Port masters) aren’t stalled by the prefetcher.

The prefetcher monitors both reads and writes to memory and the strides can either be incrementing or decrementing addresses. Prefetching is disabled at reset; scalar support can be enabled with the `basicCtrl.scalarLoadSupportEn` and `additionalCtrl.scalarStoreSupportEn` bits, and vector support can be enabled with the `additionalCtrl.vectorLoadSupportEn` and `additionalCtrl.vectorStoreSupportEn` bits.

13.2 Retiring Streams

It’s unlikely that a prefetch stream will continue indefinitely for the life of an application, so the L2 Prefetcher accommodates a method to retire old streams. This allows new streams to be tracked.

A stream can be retired by successfully hitting in the L2 Cache without the need for prefetching. If enough consecutive accesses hit in the L2 Cache, which were not prefetched, then there is no need for the prefetcher to continue monitoring the stream. The `additionalCtrl.hitCacheThrd` field sets this threshold for retiring the stream.

13.3 Page Boundaries

The L2 Prefetcher can be programmed to cross 4 KiB page boundaries of memory. This is useful in an application environment in which there is no OS page protection, or where OS page protection can be ignored. When `basicCtrl.crossPageEn` is set, prefetching can cross 4 KiB boundaries. Prefetching will resume if the hart continues to make accesses that match the stride in the new 4 KiB page of memory.

13.4 Memory Map

The memory map for the L2 Prefetcher control registers is shown in Table 142.

Offset	Name	Description
0x0	basicCtrl	L2 Prefetcher basic control register
0x4	additionalCtrl	L2 Prefetcher additional control register

Table 142: L2 Prefetcher Memory Map

13.5 Control Registers

The L2 Prefetcher control registers `basicCtrl` and `additionalCtrl` are described in the below tables.

Basic Control Register (`basicCtrl`)				
Register Address		L2 Prefetcher Base Address		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	scalarLoadSupportEn	RW	0x0	Enable hardware prefetcher support for scalar loads
1	Reserved	RW	0x0	
[7:2]	initialDist	RW	0x3	Initial prefetch distance
[13:8]	maxAllowedDist	RW	0xA	Maximum allowed prefetch distance
[19:14]	linToExpThrd	RW	0x5	Linear-to-exponential prefetch distance threshold
[27:20]	Reserved	RW	0x0	
28	crossPageEn	RW	0x0	Enable prefetches to cross-pages. When `crossPageEn == 1`, the prefetches will cross 4K boundary, if needed.
[30:29]	forgiveThrd	RW	0x0	Threshold for forgiving loads with mismatching strides when L2 Prefetcher is in trained state. Set to 0 to disable this feature (default).
31	Reserved	RO	0x0	

Table 143: Basic Control Register

Additional Control Register (`additionalCtrl`)				
Register Address		L2 Prefetcher Base Address + `0x4`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[3:0]	`qFullnessThrd`	RW	`0xE`	Threshold fraction/16 of MSHRs to stop sending hits
[8:4]	`hitCacheThrd`	RW	`0x5`	Threshold number of cache tag hits for evicting prefetch entry
[12:9]	`hitMSHRThrd`	RW	`0x2`	Threshold number of demand hits on hint MSHRs for increasing prefetch distance
[18:13]	`window`	RW	`0x6`	Size of the comparison window for address matching
19	`scalarStoreSupportEn`	RW	`0x0`	Enable hardware prefetcher support for scalar stores
20	`vectorLoadSupportEn`	RW	`0x0`	Enable hardware prefetcher support for vector loads
21	`vectorStoreSupportEn`	RW	`0x0`	Enable hardware prefetcher support for vector stores
[31:22]	Reserved			

Table 144: Additional Control Register

13.6 L2 Prefetcher Initialization

The L2 Prefetcher needs to be initialized in the boot software prior to use. Once initialized, the L2 Prefetcher can be tuned on a per-configuration basis as needed. Refer to the L2 prefetcher software example that demonstrates this tuning.

Chapter 14

Level 2 Cache Controller

This chapter describes the functionality of the Level 2 Cache Controller used in the U74-MC Core Complex.

14.1 Level 2 Cache Controller Overview

The SiFive Level 2 Cache Controller is used to provide access to fast copies of memory for masters in a Core Complex. The Level 2 Cache Controller also acts as a directory-based coherency manager.

The SiFive Level 2 Cache Controller offers extensive flexibility, as it allows for several features in addition to the Level 2 Cache functionality. These include memory-mapped access to L2 Cache RAM for disabled cache ways, scratchpad functionality, way masking and locking, ECC support with error tracking statistics, error injection, and interrupt signaling capabilities.

These features are described in Section 14.2.

14.2 Functional Description

The U74-MC Core Complex L2 Cache is a 2 MiB 16-way set-associative cache. It has a line size of 64 bytes and is read/write-allocate with a random replacement policy. The cache operates in write-back mode only. The L2 Cache is composed of 2 banks. This subdivision into banks helps facilitate increased available bandwidth between CPU masters and the L2 Cache, as each bank has its own dedicated 128-bit TL-C inner port. As such, multiple requests to different banks may proceed in parallel.

The outer port of the L2 Cache Controller is a 128-bit TL-C port shared among all banks and typically connected to a DDR controller. The outer Memory Port of the L2 Cache Controller is shared among all banks and typically connected to cacheable memory. The overall organization of the L2 Cache Controller is depicted in Figure 120.

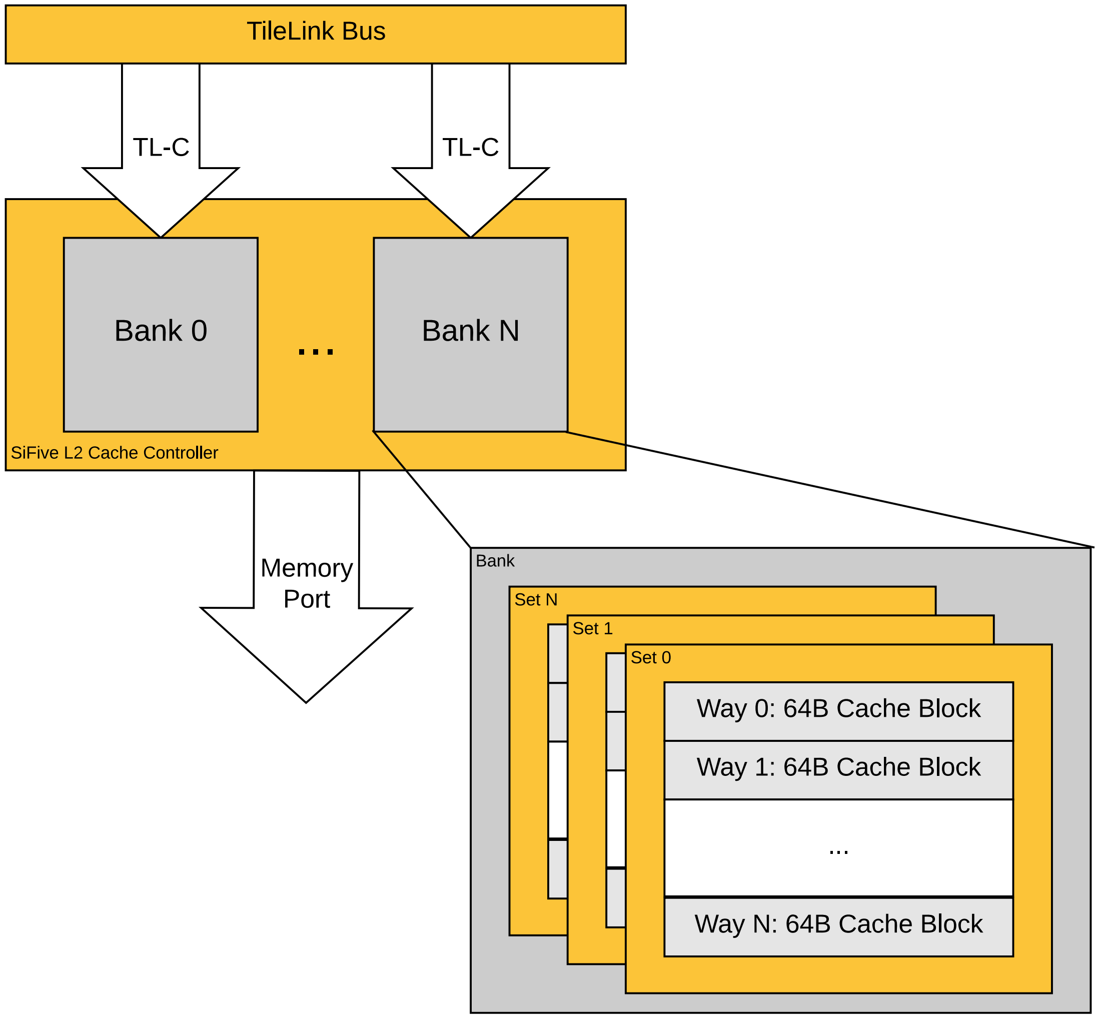

Figure 120: Organization of the SiFive L2 Cache Controller

14.2.1 Way Enable and the L2 Loosely-Integrated Memory (L2 LIM)

The SiFive Level 2 Cache Controller allows for its SRAMs to act either as direct-addressed memory in the Core Complex address space or as a cache that is controlled by the L2 Cache Controller, which can contain a copy of any cacheable address.

When cache ways are disabled, they are addressable in the L2 Loosely-Integrated Memory (L2 LIM) address space as described in the U74-MC Core Complex memory map in Section 5.2. The L2 LIM is an uncacheable port into unused L2 SRAM and provides deterministic access time. It is neither cached by the L1 data cache nor memory backed, as it is just a dedicated software-addressable, low latency, uncached memory. Fetching instructions or data from the L2 LIM provides deterministic behavior equivalent to an L2 Cache hit, with no possibility of a cache miss. Accesses to the L2 LIM are always given priority over cache way accesses, which target the same L2 Cache bank.

Out of reset, all ways, except for way 0, are disabled. Cache ways can be enabled by writing to the `WayEnable` register described in Section 14.4.2. Once a cache way is enabled, it cannot be disabled unless the U74-MC Core Complex is reset. The highest numbered L2 cache way is mapped to the lowest L2 LIM address space, and way 1 occupies the highest L2 LIM address range. As L2 cache ways are enabled, the size of the L2 LIM address space shrinks. The mapping of L2 cache ways to L2 LIM address space is shown in Figure 121, where N is the number of L2 cache ways, each of size 128 KiB (`0x0002_0000`).

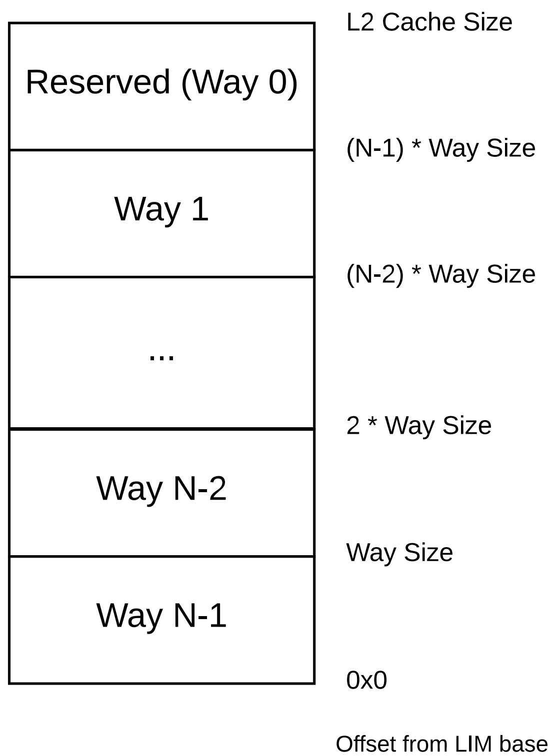

Figure 121: Mapping of L2 Cache Ways to L2 LIM Addresses

14.2.2 Way Masking and Locking

The SiFive L2 Cache Controller can control the amount of cache memory a CPU master is able to allocate into by using the `WayMaskN` register described in Section 14.4.10. Note that `WayMaskN` registers only affect allocations, and reads can still occur to ways that are masked. As such, it becomes possible to lock down specific cache ways by masking them in all `WayMaskN` registers. In this scenario, all masters can still read data in the locked cache ways but cannot evict data.

The following example shows how to lock the L2 cache ways:

1. For the following example, assume the data to be locked is not present in any of the L2 ways.
2. Select an L2 way where the data is to be locked to.

3. From Table 155, select the Master ID(s) to be used to load the data that is to be locked into the L2.
4. Clear bit M of the `WayMaskN register` for the other masters, besides the master(s) selected in the step above, to enable allocation by the selected master(s) into locked way M. The `WayMask0_M` - `WayMaskN_M` registers are described in Table 150.
    a. Use Table 150 to locate the `WayMaskN` register(s).
5. Enable the WayMask registers for the selected master(s).
    a. Set bit M of the `WayMaskN` register to 1, and clear other bits in that register, to allow master N to be able to allocate data only into way M of the L2.
6. Load the data to be locked into the L2 way M. Note that becaue the L2 is an inclusive cache, any load access by the master N will allocated the cache line for that access into the L2 cache.
7. Set the `WayMaskN_M` bit for this way to zero. At this point, no other master can evict data from this way.
8. If unlocking the data from way M is desired at some point in the future, set bit M in some or all of the `WayMaskN` registers.

14.2.3 L2 Zero Device

The SiFive L2 Cache Controller has a dedicated scratchpad address region that allows for allocation into the cache using an address range that is not memory backed. This address region is denoted as the L2 Zero Device in the Section 5.2 memory map. Writes to the scratchpad region allocate into cache ways that are enabled and not masked.

A Zero Device ignores write data and always returns zero on reads. The U74-MC Core Complex provides a Zero Device behind the L2 Cache, similar to the Memory Port. When combined with locked L2 cache ways, which prevent eviction, locations within a Zero Device's address range appear to retain their value. This provides a mechanism to create L1 cacheable memory that is essentially backed by L2 SRAM until the way is released (and the value resets to zero). The L2 Zero Device is cacheable like the Memory Port. However, if dirty data is evicted and a write-back to the L2 Zero Device occurs, the Zero Device will discard the write data. Therefore, care must be taken with the scratchpad, as there is no memory backing this address space. Cache evictions from addresses in the scratchpad result in data loss.

The main advantage of the L2 Zero Device over the L2 LIM is that it is a cacheable region allowing for data stored to the scratchpad to also be cached in a master's L1 data cache, which results in faster access.

To understand the difference between the L2 LIM and the L2 Zero Device, consider Figure 122. Notice that the L2 LIM accesses the same blocks of memory as the main path into the L2 Cache, whereas the L2 Zero Device sits behind L2 Cache much like the Memory Port:

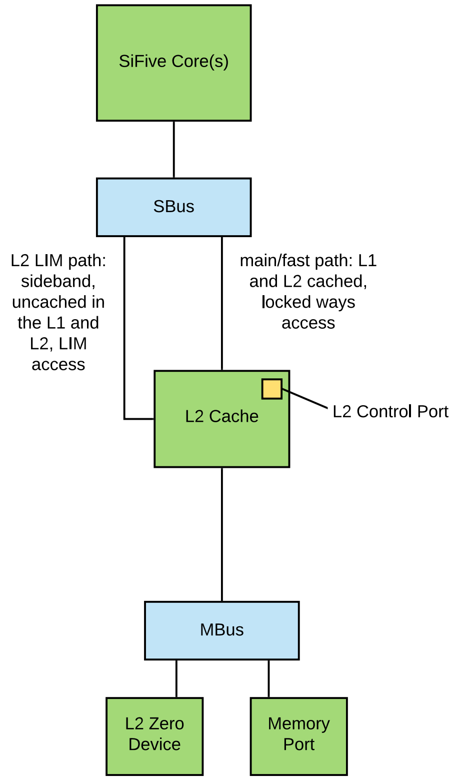

Figure 122: Difference between L2 LIM and L2 Zero Device

The recommended procedure for using the L2 Zero Device is as follows:

1. Use the `WayEnable` register to enable the desired cache ways
2. Designate a single master that will allocate into the scratchpad. For this procedure, we designate this master as Master S. All other masters (CPU and non-CPU) are denoted as Masters N.
3. Masters N: Write to the `WayMaskN` register to mask the ways that are to be used for the scratchpad. This prevents Masters N from evicting cache lines in the designated scratchpad ways.
4. Master S: Write to the `WayMaskN` register to mask all ways *except* the ways that are to be used for the scratchpad. At this point, Master S should only be able to allocate into the cache ways meant to be used as a scratchpad.
5. Master S: Write scratchpad data into the L2 Zero Device address range
6. Master S: Repeat steps 4 and 5 for each way to be used as scratchpad
7. Master S: Use the `WayMaskN` register to mask the scratchpad ways for Master S so that it is no longer able to evict cache lines from the designated scratchpad ways
8. At this point, the scratchpad ways should contain the scratchpad data, with all masters able to read, write, and execute from this address space, and no masters able to evict the scratchpad contents

14.2.4 L2 Features Access Summary

Table 145 describes the L2 features as a function of Way Enable and Way Mask.

Way Enable	Way Mask	Access Base Address	L2 Feature
0	X	`0x0800_0000`	LIM
1	0	`0x8000_0000`	Way Masking and Locking — Fast Read access
1	0	`0x0A00_0000`	Zero Device — Fast Read access from scratchpad
1	1	`0x8000_0000`	Way Masking and Locking — Write data mode
1	1	`0x0A00_0000`	Zero Device — Write data to scratchpad

Table 145: L2 Features Access Summary

14.2.5 L2 Performance Monitor

Similar to the hardware performance monitor (HPM) for the core, the L2 Cache also has an L2 performance monitoring (L2PM) facility. It consists of a set of event-programmable counters and their event selector registers. The registers are available to control the behavior of the counters. The performance event selector and other control registers are configured in machine mode, and the event-programmable counters can be read in user mode.

L2PM Event Selector Registers (`pmEventSelectX`)

The L2PM event selector CSRs (`pmEventSelect0-5`) follow the definition of `mhpmeventX` as defined in the RISC-V ISA. They are 64-bit bit **WARL** registers. To control the event type to count, these CSRs are used to program the corresponding event counters.

Offset	Bits	Access	Description
`0x2000`	64	RW	`pmEventSelect0`
`0x2008`	64	RW	`pmEventSelect1`
`0x2010`	64	RW	`pmEventSelect2`
`0x2018`	64	RW	`pmEventSelect3`
`0x2020`	64	RW	`pmEventSelect4`
`0x2028`	64	RW	`pmEventSelect5`

Table 146: L2 Performance Monitor Event Selectors

The event selectors are partitioned into two fields: the lower 8 bits select an event class, and the upper bits form a mask of events in that class.

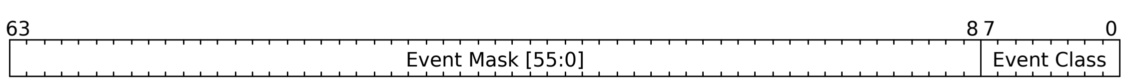

Figure 123: Event Selector Fields

The counter increments if the event corresponding to any set mask bit occurs.

L2PM Counter Client Filter CSR (`pmClientFilterX`)

The L2PM counter client filter register (`pmClientFilter0`) is a 64-bit **WARL** register that controls which client's performance events are excluded from incrementing the performance monitor counters. However, some events might not be attributed to a particular client in a specific implementation (for example, the L2 release event in SiFive's L2 Cache design).

Offset	Bits	Access	Description
`0x2800`	64	RW	`pmClientFilter0`

Table 147: L2 Performance Monitor Counter Client Filter

L2 Performance Monitor Counters (`pmEventCounterX`)

The L2 Performance Monitor counters (`pmEventCounter0-5`), follow the definition of `mhpmcounterX` as defined in the RISC-V ISA. They are 64-bit **WARL** registers.

Offset	Bits	Access	Description
`0x3000`	64	RW	`pmEventCounter0`
`0x3008`	64	RW	`pmEventCounter1`
`0x3010`	64	RW	`pmEventCounter2`
`0x3018`	64	RW	`pmEventCounter3`
`0x3020`	64	RW	`pmEventCounter4`
`0x3028`	64	RW	`pmEventCounter5`

Table 148: L2 Performance Monitor Counters

> **Note**
>
> If a counter is not implemented, such as when there is no L2PM, both the counter and its corresponding event selector are hard-wired to 0, meaning that the counter always returns 0.

Event Selector Encodings

Table 149 describes the event selector encodings available. Events are categorized into two classes based on the Event Class field encoded in `pmEventSelectX[7:0]`. One or more events can be programmed by setting the respective Event Mask bit for a given event class. An event selector encoding of 0 means "count nothing". Multiple events will cause the counter to increment any time any of the selected events occur.

L2 Performance Monitor Event Register		
Transaction Events, `pmEventSelectX[7:0]=0x1`		
Bits	**Description**	**Can use `ClientFilter?`**
0	`PutFullData` request from inner cache	Yes
1	`PutPartialData` request from inner cache	Yes
2	`AtomicData` request from inner cache	Yes
3	`Get` request from inner cache	Yes
4	`PrefetchRead` request from inner cache	Yes
5	`PrefetchWrite` request from inner cache	Yes
6	`AcquireBlock.NtoB` request from inner cache	Yes
7	`AcquireBlock.NtoT` request from inner cache	Yes
8	`AcquireBlock.BtoT` request from inner cache	Yes
9	`AcquirePerm.NtoT` request from inner cache	Yes
10	`AcquirePerm.BtoT` request from inner cache	Yes
11	`Release.TtoB` request from inner cache	Yes
12	`Release.TtoN` request from inner cache	Yes
13	`Release.BtoN` request from inner cache	Yes
14	`ReleaseData.TtoB` request from inner cache	Yes
15	`ReleaseData.TtoN` request from inner cache	Yes
16	`ReleaseData.BtoN` request from inner cache	Yes
17	`ProbeBlock.toT` request from outer cache	No
18	`ProbeBlock.toB` request from outer cache	No
19	`ProbeBlock.toN` request from outer cache	No
L2 Query Result Events, `pmEventSelectX[7:0]=0x2`		
Bits	**Description**	**Can use `ClientFilter?`**
0	`PutFullData` request from inner cache hits a valid line in L2	Yes
1	`PutPartialData` request from inner cache hits a valid line in L2	Yes
2	`AtomicData` request from inner cache hits a valid line in L2	Yes
3	`Get` request from inner cache hits a valid line in L2	Yes
4	`Prefetch` request from inner cache hits a valid line in L2	Yes
5	`AcquireBlock` request from inner cache hits a valid line in L2	Yes
6	`AcquirePerm` request from inner cache hits a valid line in L2	Yes
7	`Release` request from inner cache hits a valid line in L2	Yes
8	`ReleaseData` request from inner cache hits a valid line in L2	Yes
9	`Probe` request from outer cache hits a valid line in L2	No
10	`PutFullData` request from inner cache hits a shared line in L2	Yes

Table 149: L2PM `pmEventSelect` Register

11	`PutPartialData` request from inner cache hits a shared line in L2	yes
12	`AtomicData` request from inner cache hits a shared line in L2	Yes
13	`Get` request from inner cache hits a shared line in L2	Yes
14	`Prefetch` request from inner cache hits a shared line in L2	Yes
15	`AcquireBlock` request from inner cache hits a shared line in L2	Yes
16	`AcquirePerm` request from inner cache hits a shared line in L2	Yes
17	`Probe` request from outer cache hits a shared line in L2	No
18	`Probe` request from outer cache hits a dirty line in L2	No
L2 Request Events, `pmEventSelectX[7:0]=0x3`		
Bits	**Description**	**Can use `ClientFilter`?**
0	`AcquireBlock.NtoB` request to outer cache, miss	No
1	`AcquireBlock.NtoT` request to outer cache, miss	No
2	`AcquireBlock.BtoT` request to outer cache, miss	No
3	`AcquirePerm.NtoT` request to outer cache, miss	No
4	`AcquirePerm.BtoT` request to outer cache, miss	No
5	`Release.TtoB` request to outer cache, eviction	No
6	`Release.TtoN` request to outer cache, eviction	No
7	`Release.BtoN` request to outer cache, eviction	No
8	`ReleaseData.TtoB` request to outer cache, not applicable	No
9	`ReleaseData.TtoN` request to outer cache, dirty eviction	No
10	`ReleaseData.BtoN` request to outer cache, not applicable	No
11	`ProbeBlock.toT` request to inner cache, code miss hits other harts	No
12	`ProbeBlock.toB` request to inner cache, load miss hits other harts	No
13	`ProbeBlock.toN` request to inner cache, store miss hits other harts	No
Other Events, `pmEventSelectX[7:0]=0x4`		
Bits	**Description**	**Can use `ClientFilter`?**
0	`Hint` request from inner cache hits an inflight miss request	Yes
Note: "Inner cache" refers to the inner side of the L2 cache, i.e., transactions from the core and Front Port. "Outer cache" refers to the other side of the L2 Cache, i.e., read/write transactions from the L2 and probe requests from outside.		

Table 149: L2PM `pmEventSelect` Register

Setting up the `pmClientFilterX` Register

Table 155 shows the mapping of L2 Cache masters and their client ID.

Note that the default value of `pmClientFilterX` is `0`, meaning all clients' events are counted. To disable event counting for a particular client, set its corresponding bit in `pmClientFilterX`.

Programming the `pmEventSelect` registers

The following example shows the use of 6 performance counters:

```
// directory lookup events: L1 miss
*pmEventSelect0 = 0x01            |  //   Event Set 1
                 ( (0x01  << 6 )  |  //   innerAcquireBlockNtoB
                   (0x01  << 7 )  |  //   innerAcquireBlockNtoT
                   (0x01  << 8 )  |  //   innerAcquireBlockBtoT
                   (0x01  << 9 )  |  //   innerAcquirePermNtoT
                   (0x01  << 10)     //   innerAcquirePermB2T
                 ) << 8;
// directory lookup results: L1 miss hit L2
*pmEventSelect1 = 0x02            | //   Event Set 2
                 ( (0x01 << 5)    | //   innerAcquireBlock_Hit
                   (0x01 << 6)      //   innerAcquirePerm_Hit
                 ) << 8;
// directory lookup events: prefetch
*pmEventSelect2 = 0x01            |  //   Event Set 1
                 ( (0x01  << 4 )  |  //   innerPrefetchRead
                   (0x01  << 5 )     //   innerPrefetchWrite
                 ) << 8;
// prefetch hits L2
*pmEventSelect3 = 0x02            | //   Event Set 2
                 ( (0x01 << 4)      //   innerPrefetch_Hit
                 ) << 8;

// L1 request misses L2
*pmEventSelect4 = 0x03            | //   Event Set3
                 ( (0x01 << 0)    | //   outerAcquireBlockNtoB
                   (0x01 << 1)    | //   outerAcquireBlockNtoT
                   (0x01 << 2)    | //   outerAcquireBlockBtoT
                   (0x01 << 3)    | //   outerAcquirePermNtoT
                   (0x01 << 4)      //   outerAcquirePermBtoT
                 ) << 8;
```

14.2.6 Error Correction Codes (ECC)

The SiFive Level 2 Cache Controller supports ECC. ECC is applied to both categories of SRAM used, the data SRAMs and the metadata SRAMs (index, tag, and directory information). The data SRAMs use Single-Error Correcting, Double-Error Detecting (SECDED). The metadata SRAMs use Single-Error Correcting, Double-Error Detecting (SECDED).

Whenever a correctable error is detected, the cache immediately repairs the corrupted bit and writes it back to SRAM. This corrective procedure is completely invisible to application software. However, to support diagnostics, the cache records the address of the most recently corrected metadata and data errors. Whenever a new error is corrected, a counter is increased, and an interrupt is raised. There are independent addresses, counters, and interrupts for correctable metadata and data errors.

`DirError`, `DirFail`, `DataError`, and `DataFail` signals are used to indicate that an L2 metadata, uncorrectable L2 metadata, L2 data, or uncorrectable L2 data error has occurred, respectively. These signals are connected to the PLIC, as described in Chapter 10, and are cleared upon reading their respective count registers.

14.2.7 Coherence

The SiFive L2 Cache is partially inclusive of the L1 instruction cache and is inclusive of the L1 data cache. When a block of data is allocated to the L1 cache, it is also allocated to the L2 Cache. When a block is evicted from the L1, the corresponding block in the L2 is then updated and marked dirty.

To understand how coherence is managed differently in the L2 Cache with respect to the L1 instruction and data caches, consider the following rules:

1. Only an instruction cache allocation from the Memory Port will land in the L2 Cache
2. An eviction from the L2 Cache does not cause an eviction from the instruction cache
3. An eviction from the instruction cache does not cause L2 Cache eviction either
4. A discard from the data cache does not invalidate the L2 Cache
5. Following a flush in the L2 Cache, the L2 Cache will back probe lines in L1 data cache

14.3 Memory Map

The L2 Cache Controller memory map is shown in Table 150.

Offset	Name	Description
0x0000	Config	Information about the Cache Configuration
0x0008	WayEnable	The index of the largest way which has been enabled. May only be increased.
0x0040	ECCInjectError	Inject an ECC Error
0x0100	DirECCFixLow	The low 32-bits of the most recent address to fail ECC
0x0104	DirECCFixHigh	The high 32-bits of the most recent address to fail ECC
0x0108	DirECCFixCount	Reports the number of times an ECC error occurred
0x0120	DirECCFailLow	The low 32-bits of the most recent address to fail ECC
0x0124	DirECCFailHigh	The high 32-bits of the most recent address to fail ECC
0x0128	DirECCFailCount	Reports the number of times an ECC error occurred
0x0140	DatECCFixLow	The low 32-bits of the most recent address to fail ECC
0x0144	DatECCFixHigh	The high 32-bits of the most recent address to fail ECC
0x0148	DatECCFixCount	Reports the number of times an ECC error occurred
0x0160	DatECCFailLow	The low 32-bits of the most recent address to fail ECC
0x0164	DatECCFailHigh	The high 32-bits of the most recent address to fail ECC
0x0168	DatECCFailCount	Reports the number of times an ECC error occurred
0x0200	Flush64	Flush the physical address equal to the 64-bit written data from the cache
0x0240	Flush32	Flush the physical address equal to the 32-bit written data << 4 from the cache
0x0800	WayMask0	Master 0 way enable mask register
0x0808	WayMask1	Master 1 way enable mask register
0x0810	WayMask2	Master 2 way enable mask register
0x0818	WayMask3	Master 3 way enable mask register
0x0820	WayMask4	Master 4 way enable mask register
0x0828	WayMask5	Master 5 way enable mask register
0x0830	WayMask6	Master 6 way enable mask register
0x0838	WayMask7	Master 7 way enable mask register
0x0840	WayMask8	Master 8 way enable mask register
0x0848	WayMask9	Master 9 way enable mask register
0x0850	WayMask10	Master 10 way enable mask register
0x0858	WayMask11	Master 11 way enable mask register
0x0860	WayMask12	Master 12 way enable mask register
0x0868	WayMask13	Master 13 way enable mask register
0x0870	WayMask14	Master 14 way enable mask register
0x0878	WayMask15	Master 15 way enable mask register
0x0880	WayMask16	Master 16 way enable mask register
0x0888	WayMask17	Master 17 way enable mask register
0x0890	WayMask18	Master 18 way enable mask register
0x0898	WayMask19	Master 19 way enable mask register
0x08A0	WayMask20	Master 20 way enable mask register

Table 150: L2 Cache Controller Memory Map

Offset	Name	Description
`0x08A8`	`WayMask21`	Master 21 way enable mask register
`0x08B0`	`WayMask22`	Master 22 way enable mask register
`0x08B8`	`WayMask23`	Master 23 way enable mask register
`0x08C0`	`WayMask24`	Master 24 way enable mask register
`0x08C8`	`WayMask25`	Master 25 way enable mask register
`0x08D0`	`WayMask26`	Master 26 way enable mask register
`0x08D8`	`WayMask27`	Master 27 way enable mask register
`0x08E0`	`WayMask28`	Master 28 way enable mask register
`0x08E8`	`WayMask29`	Master 29 way enable mask register
`0x08F0`	`WayMask30`	Master 30 way enable mask register
`0x1000`	`FeatureDisable`	Composable cache
`0x2000`	`pmEventSelect0`	Performance monitor event select 0
`0x2008`	`pmEventSelect1`	Performance monitor event select 1
`0x2010`	`pmEventSelect2`	Performance monitor event select 2
`0x2018`	`pmEventSelect3`	Performance monitor event select 3
`0x2020`	`pmEventSelect4`	Performance monitor event select 4
`0x2028`	`pmEventSelect5`	Performance monitor event select 5
`0x2800`	`pmClientFilter0`	Performance counter client disable mask 0
`0x3000`	`pmEventCounter0`	Performance monitor event counter 0
`0x3008`	`pmEventCounter1`	Performance monitor event counter 1
`0x3010`	`pmEventCounter2`	Performance monitor event counter 2
`0x3018`	`pmEventCounter3`	Performance monitor event counter 3
`0x3020`	`pmEventCounter4`	Performance monitor event counter 4
`0x3028`	`pmEventCounter5`	Performance monitor event counter 5

Table 150: L2 Cache Controller Memory Map

14.4 Register Descriptions

This section describes the functionality of the memory-mapped registers in the Level 2 Cache Controller.

14.4.1 L2 Cache Configuration Register (`Config`)

The `Config` Register can be used to programmatically determine information regarding the cache size and organization.

L2 Cache Configuration Register (`Config`)				
Register Offset		`0x0`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	`Banks`	RO	`0x2`	Number of banks in the cache
[15:8]	`Ways`	RO	`0x10`	Number of ways per bank
[23:16]	`lgSets`	RO	`0xA`	Base-2 logarithm of the sets per bank
[31:24]	`lgBlockBytes`	RO	`0x6`	Base-2 logarithm of the bytes per cache block

Table 151: L2 Cache Configuration Register

14.4.2 Way Enable Register (`WayEnable`)

The `WayEnable` register determines which ways of the Level 2 Cache Controller are enabled as cache. Cache ways that are not enabled are mapped into the U74-MC Core Complex's L2 LIM (Loosely-Integrated Memory) as described in the memory map in Section 5.2.

This register is initialized to 0 on reset and may only be increased. This means that, out of reset, only a single L2 cache way is enabled, as one cache way must always remain enabled. Once a cache way is enabled, the only way to map it back into the L2 LIM address space is by a reset.

Way Enable Register (`WayEnable`)				
Register Offset		`0x8`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	`WayEnable`	WARL	`0x0`	The index of the largest way which has been enabled. May only be increased.

Table 152: Way Enable Register

14.4.3 ECC Error Injection Register (`ECCInjectError`)

The `ECCInjectError` register can be used to insert an ECC error into either the backing data or metadata SRAM. This function can be used to test error correction logic, measurement, and recovery.

ECC Error Injection Register (`ECCInjectError`)				
Register Offset		`0x40`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	`ECCToggleBit`	RW	`0x0`	Toggle (corrupt) this bit index on the next cache operation
[15:8]	Reserved			
16	`ECCToggleType`	RW	`0x0`	Toggle (corrupt) a bit in 0=data or 1=directory
[31:17]	Reserved			

Table 153: ECC Error Injection Register

14.4.4 ECC Directory Fix Registers (`DirECCFix*`)

The `DirECCFixHigh` and `DirECCFixLow` registers are read-only registers that contain the address of the most recently corrected L2 metadata error. This field supplies only the portions of the address that correspond to the affected set, since all ways are corrected together.

The `DirECCFixCount` register is a read-only register that contains the number of corrected L2 metadata errors. Reading this register clears the `DirError` interrupt signal described in Section 14.2.6.

Note that these registers are volatile and may be masked in IP-XACT.

14.4.5 ECC Directory Fail Registers (`DirECCFail*`)

The `DirECCFailLow` and `DirECCFailHigh` registers are read-only registers that contains the address of the most recent uncorrected L2 metadata error.

The `DirECCFailCount` register is a read-only register that contains the number of uncorrected L2 metadata errors. Reading this register clears the `DirFail` interrupt signal described in Section 14.2.6.

Note that these registers are volatile and may be masked in IP-XACT.

14.4.6 ECC Data Fix Registers (`DatECCFix*`)

The `DatECCFixLow` and `DatECCFixHigh` registers are read-only registers that contain the address of the most recently corrected L2 data error.

The `DataECCFixCount` register is a read-only register that contains the number of corrected L2 data errors. Reading this register clears the `DataError` interrupt signal described in Section 14.2.6.

Note that these registers are volatile and may be masked in IP-XACT.

14.4.7 ECC Data Fail Registers (`DatECCFail*`)

The `DatECCFailLow` and `DatECCFailHigh` registers are a read-only registers that contain the address of the most recent uncorrected L2 data error.

The `DatECCFailCount` register is a read-only register that contains the number of uncorrected L2 data errors. Reading this register clears the `DataFail` interrupt signal described in Section 14.2.6.

Note that these registers are volatile and may be masked in IP-XACT.

14.4.8 L2 Cache ECC Error Injection and Correction

Writes to the `ECCInjectError` register, described in Section 14.4.3, specifies a bit position within the combined data+ECC (or directoryEntry+ECC) granule to toggle or corrupt exactly 1 bit the next time an L2 entry is written.

The `ECCToggleBit` register specifies a hex value corresponding to which bit to toggle. For example, if you write a value of `0x47`, bit 71 will be toggled (consequently, for a write to an L2 data RAM that has 64 bits of data and 8 bits of ECC, writing a value > `0x47` will have no effect).

The error will be injected (i.e., the bit will be toggled/corrupted) on the next write. When there are multiple data+ECC granules comprising a cache line, error injection will always be applied to the first of these granules. For instance, when you have a 64+8-bit granule and a 64-byte line, it takes eight granules to make up a full cache line, and errors will be injected into the specified bit in the first of these eight granules. For metadata, this is a non-issue since there is only one directoryEntry+ECC granule for each cache line.

The error will not be reported until the corrupted entry is subsequently read, and the error is detected/corrected; that is, this will be at an arbitrary time after the `ECCInjectError` Register is written, unless a test specifically ensures that it reads the particular set and way that was corrupted due to that write.

The address reported in the fix/fail address registers (`DirECCFix` and `DirECCFail`, respectively) will always be the cache-line-size-aligned address of the start of the cache line in which the error was detected.

For error injection on the data, if the L1-L2 interface (and consequently, the width of the L2 data R/W interface) is larger than the ECC granule, the designated bit will actually be toggled on multiple granules; the number of granules being as many granules as make up the data interface. For example, if the L2 data interface is 128-bits wide and the data+ECC granule is 72-bits, two bits will get toggled—one per granule. This is, however, invisible to software since, when these bits are read, only a single corrected error will be reported, and the address will be the start of the cache line as stated above.

For the reported fix address (`DirECCFix`) on corrected errors in the metadata, only the address bits corresponding to the set (i.e., the index) are captured. In contrast, address bits corresponding to the tag are not captured and they will read out as `0`.

14.4.9 Cache Flush Registers (`Flush*`)

The U74-MC Core Complex L2 Cache Controller provides two registers that can be used for flushing specific cache blocks.

`Flush64` is a 64-bit write-only register that flushes the cache block containing the address written. `Flush32` is a 32-bit write-only register that flushes a cache block containing the written address left shifted by 4 bytes. In both registers, all bits must be written in a single access for the flush to take effect.

The flush operation performs a write-back and invalidate, meaning the contents are written to memory and L2 and L1 cache lines are then invalidated.

14.4.10 Way Mask Registers (`WayMask*`)

The `WayMaskN` register allows a master connected to the L2 Cache Controller to specify which L2 Cache ways can be evicted by Master N. Masters can still access memory cached in masked ways. The mapping between masters and their L2 master IDs is shown in Table 155.

At least one cache way must be enabled. It is recommended to set/clear bits in this register using atomic operations.

Way Mask 0 Register (`WayMask0`)				
Register Offset		`0x800`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	`WayMask0_0`	RW	`0x1`	Enable way 0 for Master 0
1	`WayMask0_1`	RW	`0x1`	Enable way 1 for Master 0
2	`WayMask0_2`	RW	`0x1`	Enable way 2 for Master 0
3	`WayMask0_3`	RW	`0x1`	Enable way 3 for Master 0
4	`WayMask0_4`	RW	`0x1`	Enable way 4 for Master 0
5	`WayMask0_5`	RW	`0x1`	Enable way 5 for Master 0
6	`WayMask0_6`	RW	`0x1`	Enable way 6 for Master 0
7	`WayMask0_7`	RW	`0x1`	Enable way 7 for Master 0
8	`WayMask0_8`	RW	`0x1`	Enable way 8 for Master 0
9	`WayMask0_9`	RW	`0x1`	Enable way 9 for Master 0
10	`WayMask0_10`	RW	`0x1`	Enable way 10 for Master 0
11	`WayMask0_11`	RW	`0x1`	Enable way 11 for Master 0
12	`WayMask0_12`	RW	`0x1`	Enable way 12 for Master 0
13	`WayMask0_13`	RW	`0x1`	Enable way 13 for Master 0
14	`WayMask0_14`	RW	`0x1`	Enable way 14 for Master 0
15	`WayMask0_15`	RW	`0x1`	Enable way 15 for Master 0

Table 154: Way Mask 0 Register

Master ID	Description
0	Debug
1	Hart 0 Fetch Unit
2	Hart 0 D-Cache MMIO
3	Hart 1 Fetch Unit
4	Hart 1 D-Cache
5	Hart 1 D-Cache
6	Hart 1 D-Cache MMIO
7	Hart 1 L2 Prefetcher
8	Hart 2 Fetch Unit
9	Hart 2 D-Cache
10	Hart 2 D-Cache
11	Hart 2 D-Cache MMIO
12	Hart 2 L2 Prefetcher
13	Hart 3 Fetch Unit
14	Hart 3 D-Cache
15	Hart 3 D-Cache
16	Hart 3 D-Cache MMIO
17	Hart 3 L2 Prefetcher
18	Hart 4 Fetch Unit
19	Hart 4 D-Cache
20	Hart 4 D-Cache
21	Hart 4 D-Cache MMIO
22	Hart 4 L2 Prefetcher
23	AXI4 Front Port ID#0 [W]
24	AXI4 Front Port ID#1 [W]
25	AXI4 Front Port ID#2 [W]
26	AXI4 Front Port ID#3 [W]
27	AXI4 Front Port ID#0 [R]
28	AXI4 Front Port ID#1 [R]
29	AXI4 Front Port ID#2 [R]
30	AXI4 Front Port ID#3 [R]

Table 155: Master IDs in the L2 Cache Controller

14.5 Procedure to Flush the L2 Cache

This section describes how to flush the L2 Cache using the Zero Device scratchpad. As the scratchpad resides in the L2 Cache, allocations made to the scratchpad can result in an eviction of a cache line that was allocated from the Memory Port. By controlling the `WayMaskN` register and targeted scratchpad address, it is possible to perform different flush operations.

As the scratchpad region is cacheable, it is necessary to perform a flush of the scratchpad region used to perform the L2 flush. Otherwise, if the targeted scratchpad line already exists in the L2 Cache, either through a direct access or speculation, it will prevent the desired L2 cache line from being evicted from the Memory Port when it is accessed. The steps outlined below include this necessary flush of the scratchpad region.

For flushing ranges of data, it is recommended to use the flush-by-address function in the L2 Controller space, described in Section 14.4.9.

14.5.1 Flushing a Single Index+Way

1. Write `WayMaskN` to allow evictions from only the desired way
2. Issue a FENCE instruction
3. Determine address A in the L2 scratchpad region that corresponds to the desired index to be flushed. For example, consider a 2 MiB 16-way cache with the Zero Device located at `0x0A00_000`. To target index 8, the valid addresss would be {`0x0A00_0200`, `0x0A02_0200`, `0x0A04_0200`, …, `0x0A1E_0200`}.
4. Write address A to the L2 Controller `Flush64` register
5. Issue a load from address A (or a store to address A)
6. (Optional) To force observation of the flush, write address A to the L2 `Flush64` register and issue a FENCE instruction
7. Restore the `WayMaskN` register to the original value
8. Issue a FENCE instruction

14.5.2 Flushing the Entire L2 Cache

1. Write `WayMaskN` to allow evictions from only way `0`
2. Issue a FENCE instruction
3. Flush the first way-size of L2 scratchpad memory using a series of `Flush64` operations. For example, consider a 2 MiB 16-way cache with the Zero Device located at `0x0A00_000`. 2048 lines (128 KiB) in the region from `0xA000_0000-0xA001_FFC0` must be targeted with a `Flush64` operation.
4. Issue a FENCE instruction
5. Access the first way-size of L2 scratchpad memory using a series of load or store operations that correspond to each index of the L2 Cache. Considering the same cache example (2 MiB, 16-way), all 2048 lines in the region from `0x0A00_0000-0xA001_FFC0` must be accessed. Only one access per 64 B cache line is required. Each scratchpad access will cause an eviction if the corresponding cache index is dirty.
6. Issue a FENCE instruction
7. Repeat steps 1-6 for the next way to be flushed until all ways have been flushed. The region flushed in step 3 needs to advance by the cache way size, as does the region accessed in step 5. Considering the same example, address range `0xA002_0000-0xA003_FFC0` will be targeted for way 1 operations.

8. (Optional) To force observation of the complete flush, repeat steps 3 and 4
9. Restore the `WayMaskN` register to the original value
10. Issue a FENCE instruction Memory accesses from harts not performing the flush can interfere with the flush operation. It is recommended to put other harts into WFI or a holding loop until the L2 flush is completed.

Chapter 15

Power Management

The following chapter describes power modes and establishes flows for powering up, powering down, and resetting the hardware of the U74-MC Core Complex.

15.1 Power Modes

Power modes include normal run mode with the Power Dial option and wait-for-interrupt clock gating mode using the `WFI` instruction. Additionally, there is a full power down mode supported via the `CEASE` instruction. These modes are covered in detail below.

15.2 Run Mode

The hart is fully operational in run mode, and SiFive designs include the option to include coarse-grained architectural clock gating. When this feature is enabled in the hart, configured instruction cache, data cache, integer pipeline, Debug Logic, and Floating-Point Unit (FPU) modules each contain their own clock gate. The clock gating feature will enable automatic clock gating of functional units when they are inactive and allow the hart to gate its own clock(s) based on activity.

15.2.1 Power Control

Core Clock

To further reduce power while in run mode, users may choose to reduce `external_source_for_core_N_clock`, which is required to be changed synchronously to the rest of the clocks in the system. It is important to note that the clock relationships with the rest of the system must still be maintained if `external_source_for_core_N_clock` is reduced. `external_source_for_core_N_clock` and all other clock signals are described in the U74-MC Core Complex User Guide.

Power Dial

To limit maximum power without frequency changes, Power Dial provides a method of scaling down dynamic power in a core. Power is reduced by restricting cycles allowed to advance instructions into the execution pipeline. This feature can typically be used for throttling high CPU usage applications.

Power Dial CSR				
CSR	0x7C8			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[3:0]	dutycycle	RW	0x0	(1 - value/16) portion of the peak instruction throughput (i.e., value=0 is no reduction)
[7:4]	Reserved	RO	0x0	

Table 156: Power Dial CSR

The Power Dial Register may only be programmed in M-mode. When written with a non-zero value, the register restricts peak instruction throughput to the indicated rate. A value of 0 has no effect on instruction throughput. The rate is calculated per 256-cycle period, so a Power Dial value of 1 restricts instruction throughput at a common point in the pipeline to 240 cycles of each 256-cycle period. Reducing the peak rate reduces the worst-case power while minimizing the impact on performance.

15.3 WFI Clock Gate Mode

WFI clock gating mode can be entered by executing the `WFI` instruction. The assembly-level instruction is simply `wfi` and executing the C method using the GCC compiler can be accomplished with `asm("WFI")`.

15.3.1 WFI Wake Up

Wake up from a WFI occurs when the hart receives any interrupt. Depending on the software configuration, the hart will either immediately enter the interrupt handler, or resume execution on the instruction immediately after the WFI.

If interrupts are enabled and `mstatus.MIE=1`, then the hart will wake when an interrupt is enabled and becomes pending, and immediately enter the interrupt handler. Upon exit from the interrupt handler, program execution will resume at the instruction following the `WFI`.

If interrupts are enabled but `mstatus.MIE=0`, then the hart will wake when an interrupt is enabled and becomes pending but will not enter the interrupt handler. It will simply resume at the instruction immediately after the `WFI` in this case.

To prevent an interrupt source from waking a hart, the enable bit for that interrupt must be written to 0 prior to executing the `WFI` instruction. If any interrupts are pending upon executing a `WFI` instruction, then the `WFI` is effectively treated as a `NOP` instruction.

Refer to Chapter 8 for more detail on interrupt configuration.

15.4 CEASE Instruction for Power Down

To fully power down, follow the steps described in Section 15.11, where the last step is to execute a `CEASE` instruction. Once the `CEASE` instruction is executed, the core will not retire another instruction until reset. The `CEASE` opcode is `0x30500073` and can be implemented in either assembly or C. To create an assembly-level function using GCC, consider the following example.

```
.global _cease
.type         _cease, @function
_cease:
    .word 0x30500073
    ret
```

The next example demonstrates how to implement the `CEASE` instruction within a function in C.

```
static inline void cease()
{
    __asm__ __volatile__ (".word 0x30500073" : : : "memory"); // CEASE
}
```

15.5 Subsystem LowPower Controller (SLPC)

The Subsystem LowPower Controller (SLPC) is a unit inside the uncore that contains power management-related logic.

It includes the `CorePowerState` register that indicates the individual `WFI` and `CEASE` status for up to 16 Tiles in the core-complex.

It is used by core-complex power gating, but can be present when core-complex power gating is not enabled or supported.

Below are the memory map and bitfield definitions for the SLPC:

Location	Description
`base + 0x000`	CorePower State Register
`base + 0x008`	Reserved

Table 157: CorePower State Memory Map

CorePower State Register		
Bits	**Field**	**Description**
[15:0]	wfi_<x>	`WFI` indication from Tile-X, where `x` is the Tile Index. `WFI` status for up to 16 Tiles can be captured here.
[31:16]	cease_<x>	`CEASE` indication from Tile-X, where `x` is the Tile Index. `CEASE` status for up to 16 Tiles can be captured here.

Table 158: CorePower State Register

15.6 Composable Cache Clock Gating

The Composable Cache implements two levels of architectural clock gating to gate clock nodes when not active, reducing its clock tree and dynamic power.

1. The first level is a trunk clock gate on the entire Composable Cache Wrapper; that is, the clock to the entire module and all its flops is gated. This feature is disabled (i.e., the clock is always enabled) out of reset.

   `DisableClockGate` register bit `0` (`DisableWFICCTrunkClockGate`) enables the feature during start-up code. The clock gate only becomes active when all the cores in the Core IP subsystem are in `WFI` mode and the Composable Cache is idle. This saves internal clock tree and Flop's dynamic power in `WFI` mode. When any of the core is out of `WFI` mode, the clock is always enabled.

   If the feature is enabled, in `WFI` mode, the clock is only available under the following conditions:

   - Activity on the ingress memory bus initiated by any bus master
   - Activity on the LIM bus
   - Activity on the Composable Cache TileLink control bus

     The clock remains available until all the above inflight transactions are finished at the Composable Cache Wrapper boundary.

2. The second level a regional clock gate that gates the clock of some the Composable Cache's major units. This feature is disabled (i.e., the clock is always enabled) out of reset.

   `DisableClockGate` register bit `1` (`DisableCCRegionalClockGate`) enables the feature during start-up code.

Below are the memory map and bitfield definitions for the `DisableClockGate` register:

Location	**Description**
`base + 0x1000`	DisableClockGate Register

Table 159: DisableClockGate Memory Map

DisableClockGate Register				
Bits	**Field**	**Attr**	**Reset State**	**Description**
2	`DisableCCRegionalClockGateSlow`	RW	`0x1`	When set, disables regional clock gating "slow" feature. When clear, if regional clock gating is also enabled, the aggressive clock gating feature is enabled. This clock gater imposes a 1-cycle exit penalty from the gated state (on the first TL channel-A request), but saves more power than the no-penalty regional clock gate option.
1	`DisableCCRegionalClockGate`	RW	`0x1`	When set, disables all regional clock gating. When clear, non-aggressive clock gating is enabled. This clock gater has no penalty for exit, but less power savings than regional "slow" clock gating.
0	`DisableWFICCTrunkClockGate`	RW	`0x1`	When set, disables trunk clock gating feature. When clear, dynamic gating of the Ccache clock trunk is enabled. This clock gater incurs a 2-cycle penalty when exiting the `WFI` clock gated state and saves the most power. All cores in a core-complex must be in `WFI` before this state is entered.

Table 160: DisableClockGate Register

> **Note**
>
> To enable the regional clock gating "slow" feature, both bits `1` and `2` need to be cleared.

15.7 Hardware Reset

The following list summarizes the hardware reset values required by *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* and applies to all SiFive designs.

1. Privilege mode is set to machine mode.

2. `mstatus.MIE` and `mstatus.MPRV` are required to be 0.
3. The `misa` register holds the full set of supported extensions for that implementation, and `misa.MXL` defaults to the widest supported ISA available, referred to as MXLEN.
4. The `pc` is set to the implementation specific reset vector.
5. The `mcause` register is set to 0x0 at reset.
6. The PMP configuration fields for address matching mode (`A`) and Lock (`L`) are set to 0, which defaults to no protection for any privilege level.

The internal state of the rest of the system should be completed by software early in the boot flow.

15.8 Early Boot Flow

For the early stages of boot, some of the first things software must consider are listed below:

- The global pointer (`gp` or `x3`) user register should be initialized to the `__global_pointer$` linker generated symbol and not changed at any point in the application program.
- The stack pointer (`sp` or `x2`) user register should be also set up as a standard part of the boot flow.
- All other user registers (`x1`, `x4` - `x31`) can be written to 0 upon initial power-on.
- The `mtvec` register holds the default exception handler base address, so it is important to set up this register early in the boot flow, so it points to a properly aligned, valid exception handler location.
- Zero out the `bss` section and copy data sections into RAM areas as needed.

15.9 Interrupt State During Early Boot

Since `mstatus.MIE` defaults to 0, all interrupts are disabled globally out of reset. Prior to enabling interrupts globally through `mstatus.MIE`, consider the following:

- Ensure no timer interrupts are pending by checking the `mip.MTIP` bit. The `mtime` register is 0 out of reset and starts running immediately. However, the `mtimecmp` register does not have a reset value.

  If no timer interrupt is required, leave `mie.MTIE` equal to 0 prior to enabling global interrupt with `mstatus.MIE`.

  If the application requires a timer interrupt, write `mtimecmp` to a value in the future for the next timer interrupt before enabling `mstatus.MIE`.
- Write the remaining bits in the `mie` CSR to the desired value to enable interrupts based on the requirements of the system. This register is not defined to have a reset value.

- Each `msip` register in the Core-Local Interruptor (CLINT) or Core-Local Interrupt Controller (CLIC) address space is reset to 0, so no specific initialization is required for local software interrupts.

  Since `msip` is memory-mapped, any hart in the system may trigger a software interrupt on another hart, so this should be considered during the boot flow on a multi-hart system.
- If a Platform-Level Interrupt Controller (PLIC) exists, check the PLIC pending status. The PLIC memory mapped pending bits are read-only, so the pending status should be cleared at the source if they reset to a non-zero status. Then, enable the PLIC interrupts as required by the system prior to enabling interrupts in the system via `mstatus.MIE`.

  If an L2 Cache or Bus-Error Unit (BEU) is present, these interrupt IDs begin at 128, so the enable bits may lie in a different region of the memory map than other PLIC enable bits in the design.
- Wipe down memory if enabled with ECC. This can be done by writing `0x0` to memory with either store instructions issued by the CPU or using a DMA controller. ECC errors are reported via the Bus-Error Unit (BEU).
- Check BEU registers to ensure no errors are reported and that the enable bits reflect the requirements of the application.

15.10 Other Boot Time Considerations

- Write `0` to enable the appropriate bits in the Feature Disable CSR as described in Section 7.2.2.
- Ensure the remaining bits in the `mstatus` CSR are written to the desired application specific configuration at boot time.
- If a design includes user and supervisor privilege levels, initialize `medeleg` and `mideleg` registers to 0 until supervisor-level trap handling is set up correctly using `stvec`.
- The `mcause`, `mepc`, and `mtval` registers hold important information in the event of a synchronous exception. If the synchronous exception handler forces reset in the application, the contents of these registers can be checked to understand root cause.
- The PMP address and configuration CSRs are required to be initialized if user or supervisor privilege levels are part of the design. By default, user and supervisor modes have no permissions to the memory map unless explicitly granted by the PMP.
- The `mcycle` CSR is a 64-bit counter on both RV32 and RV64 systems, and it counts the number of cycles executed by the hart. It has an arbitrary value after reset and can be written as needed by the application.
- Instructions retired can be counted by the `minstret` register, and this also has an arbitrary value after reset. This can be written to any given value.
- The `mhpmeventX` CSR selects which hardware events to count, where the count is reflected in `mhpmcounterX`. At any point, the `mhpmcounterX` registers can be directly written to reset their value when the `mhpmeventX` register has the proper event selected.

- For cores with an MMU, ensure the `satp` register holds the correct configuration for address translation.
- There is no requirement for boot time initialization to any of the registers within the Debug Module, unless there is an application specific reason to do so.
- All other CSRs during boot time initialization should be considered based on system and application requirements.

15.11 Power-Down Flow

Designate one core as "primary" and all others as "secondary". For SiFive Core IP, coordination with an "External Agent" is required.

1. External Agent: Wait for communication from the primary core to initiate the following steps:
    a. Stop sending inbound traffic (both transactions and interrupts) into the Core Complex.
    b. Wait until all outstanding requests to the Core Complex are completed, then
    c. Wait until `cease_from_tile_N` is high for the primary core and all secondary cores.
    d. Once `cease_from_tile_N` is high for the primary core and all secondary cores, apply reset to the entire Core Complex.
2. Primary core:
    a. The following sequence should be executed in machine mode and NOT out of a remote ITIM/DTIM.
    b. Communicate with external agent to initiate cease power-down sequence.
    c. Poll external agent until steps 1.a and 1.b are completed.
    d. Disable all interrupts except those related to bus errors/memory corruption, and IPIs (if using enabled IPI to coordinate power-down sequence among cores).
        i. Copy contents of any TIMs/LIMs into external memory.
        ii. If there is an L2 cache, flush it.
        iii. If there is no L2 cache, but there is a data cache, flush it.
    e. Inform all secondary cores to proceed.
    f. Wait until `cease_from_tile_N` is high for all secondary cores. Examples of how this can be accomplished:
        i. Have an off-core-complex memory-mapped register that tracks the state of the `cease_from_tile_N` signals. Primary core polls this register.

   ii. Wire the `cease_from_tile_N` signals back into interrupt wires. Corresponding interrupts can be disabled along with all others in the first step. Primary core polls the interrupt-pending bits for those interrupts.

   g. Disable all interrupts.

   h. Execute CEASE instruction.

3. Secondary cores:

   a. The following sequence should be executed in machine mode and NOT out of a remote ITIM/DTIM.

   b. Execute in an idle loop for notification sent in step 2.e (could be via polling on an MMIO-accessible mailbox, polling on an IPI (disabled), or waiting on an IPI (enabled)).

   c. Disable all interrupts except those related to bus errors/memory corruption, and IPIs (if using enabled IPI to coordinate power-down sequence among cores).

   d. Copy contents of any TIMs/LIMs into external memory.

   e. If there is no L2 cache but there is a data cache, flush it using full-cache variant of `CFLUSH.D.L1` if available, or per-line variant if not.

   f. Disable all interrupts.

   g. Execute CEASE instruction.

Chapter 16

Debug

This chapter describes the operation of SiFive debug hardware, which follows *The RISC-V Debug Specification, Version 1.0*. Currently, only interactive debug and hardware breakpoints are supported.

16.1 Debug Module

The Debug Module (DM) handles nearly all of the functions related to debugging. It is a slave to both the Debug Module Interface (DMI) coming from the probe, and a TileLink bus coming from the cores. From the perspective of the core, the DM appears as a 4KiB block in the memory map. The DM memory map as seen from the perspective of the core is shown in Table 162, and the register map from the perspective of the DMI is shown in Table 161.

Most of the DM is clocked by `debug_clock`. The `dmcontrol` register is accessible when `debug_clock` is not running, mainly to be able to write to `haltreq` while the core is in reset due to `ndreset`. Doing so generates a debug interrupt and will interrupt the selected core immediately once it is out of reset or during a WFI instruction.

DMI Address	Name	Description
`0x04-0x0F`	`data0-data11`	Read/Write DATA registers. 32-bit SiFive cores have 1 data register, 64-bit cores have 2. Note that these registers are volatile and may be masked in IP-XACT.
`0x10`	`dmcontrol`	Debug Module Control. See Table 173 for more information.
`0x11`	`dmstatus`	Debug Module Status. See Table 172 for more information.
`0x12`	`hartinfo`	Hart Information. See Table 174 for more information.
`0x13`	`haltsum1`	Read-only. Halt Summary 1. Only present on systems with >32 harts. **Not used by SiFive**.
`0x14`	`hawindowsel`	Read/Write. Select which window of up to 32 harts is visible in `hawindow`. **Not used by SiFive** since all SiFive systems have less than 32 harts.
`0x15`	`hawindow`	Read/Write. Window of 32 harts to be selected, in addition to the one selected by `hartsel`. Bit 0 corresponds to hart 0. A 1 will select the corresponding hart.
`0x16`	`abstractcs`	Abstract Control and Status. See Table 175 for more information.
`0x17`	`command`	Initiate abstract command. See Table 176 for more information.
`0x18`	`abstractauto`	Selects whether access to particular DATA or PROGBUF locations will re-execute the last command. Used for block transfers or other repeating commands. See Table 177 for more information.
`0x20-0x2F`	`progbuf0-progbuf15`	Read/Write PROGBUF registers.
`0x32`	`dmcs2`	Fields to set up and read back Halt Group or Resume Group configuration. Present by default on systems with more than 1 hart or with any external triggers. See Table 178 for more information.
`0x38`	`sbcs`	System Bus Access Control and Status
`0x39`	`sbaddress0`	System Bus Address 31:0
`0x3A`	`sbaddress1`	System Bus Address 63:32.
`0x3C`	`sbdata0`	System Bus Data 31:0
`0x3D`	`sbdata1`	System Bus Data 63:32
`0x40`	`haltsum0`	Read-only. Halt Summary 0. Bit n reads 1 if hart n is halted. Note that this register is volatile and may be masked in IP-XACT.

Table 161: Debug Module Memory Map Seen from the Debug Module Interface

From the point of view of the core, the DM appears as a 4KiB block of memory. It is mapped into low memory so that memory references can use addresses relative to the `$zero` register.

TL Address	Name	Attr.	Description
`0x100`	`HALTED`	WO	Written with `hartid` by ROM code when hart gets a debug interrupt or reenters ROM due to EBREAK. Sets `halted[hartid]`. If an abstract command was running, writing this also clears busy.
`0x104`	`GOING`	WO	Written by ROM code when it begins executing a command started by `FLAGS[hartid].go`. Clears `FLAGS[hartid].go`.
`0x108`	`RESUMING`	WO	Written with `hartid` by hart when it is about to resume. Sets `resumeack[hartid]` and clears `halted[hartid]` and `FLAGS[hartid].resume`.
`0x10C`	`EXCEPTION`	WO	Written by hart when it encounters an exception in debug mode. Sets `cmderr` to "exception".
`0x300`	`WHERETO`	RO	JAL to ABSTRACT. This opcode is constructed by DM hardware and is needed because ABSTRACT is not a fixed address (depends on number of PROGBUF words selected in the configuration). Note that ABSTRACT is volatile and may be masked in IP-XACT.
`contiguous`	`ABSTRACT`	RO	2 words constructed by DM hardware based on abstract command written from DTM. • 0 - If `transfer` set, construct instruction to load/store specific register to/from `DATA[0]` (32 bits) or `DATA[1:0]` (64 bits), else NOP. • 4 - If `postexec` set, then NOP to fall thru and execute PROGBUF, else EBREAK to return to ROM park loop. Note that this register is volatile and may be masked in IP-XACT.
`contiguous`	`PROGBUF`	RW	Configurable number (typically 16, max 16) of R/W words to be filled in by debugger and executed by hart. Note that this register is volatile and may be masked in IP-XACT.
`0x380-0x3BF`	`DATA`	RW	Configurable number (1 for 32-bit or 2 for 64-bit, max 12) of R/W words intended for use for data transfer between debugger and hart. Since it is contiguous with PROGBUF, the debugger may use DATA as an extension of PROGBUF. Note that these registers are volatile and may be masked in IP-XACT.
`0x400-0x7FF`	`FLAGS`	RO	One byte flag per hart. • Bit 0 (`go`): Set by writing an abstract command, cleared by `ROM` write to `GOING`. `ROM` will jump to `WHERETO`.

Table 162: Debug Module Memory Map from the Perspective of the Core

TL Address	Name	Attr.	Description
			• Bit 1 (`resume`): Set by writing 1 to `resumereq[hartid]`. Cleared by ROM write of `hartid` to `RESUMING`. `ROM` restores `s0` then executes `dret`. Note that these registers are volatile and may be masked in IP-XACT.
`0x800-0xFFF`	`ROM`	RO	Debug interrupt or EBREAK enters at `0x800`, saves `s0`, writes `hartid` to HALTED, then busy-waits for `FLAGS[hartid]` > 0. If `FLAGS[hartid].go`, write 0 to `GOING`, then jump to `WHERETO`. Else write `hartid` to `RESUMING`, then execute `dret` to return to user program. ROM Source Code: https://github.com/chipsalliance/rocket-chip/blob/master/scripts/debug_rom/debug_rom.S

Table 162: Debug Module Memory Map from the Perspective of the Core

16.2 Debug and Trigger Registers

This section describes the per hart debug and trigger registers, which are mapped into the CSR space as follows:

CSR	Name	Allowed Access Mode	Description
`0x7B0`	`dcsr`	Debug	Debug Control and Status Register
`0x7B1`	`dpc`	Debug	Debug PC. Stores execution address just before debug exception and to return to at `dret`.
`0x7B2`	`dscratch0`	Debug	Debug Scratch Register 0
`0x7A0`	`tselect`	Debug, Machine	Trigger Select. Most configs implement 2, 4, or 8 triggers. Triggers are all type 6 (address/data).
`0x7A1`	`tdata1`	Debug, Machine	Trigger Data 1, `mcontrol6`
`0x7A2`	`tdata2`	Debug, Machine	Trigger Data 2, the address for comparison
`0x7A3`	`tdata3`	Debug, Machine	Trigger Data 3

Table 163: Debug and Trigger Registers

16.2.1 Debug Control and Status Register (`dcsr`)

This register gives information about debug capabilities and status. Its detailed functionality is described in *The RISC-V Debug Specification, Version 1.0*.

Debug Control and Status Register (`dcsr`)			
CSR	`0x7B0`		
Bits	**Field Name**	**Attr.**	**Description**
[1:0]	`prv`	RW	Privilege level of processor prior to debug exception and to return to at `dret`.
2	`step`	RW	Set to 0x1 to single-step.
3	`nmip`	RO	Non-maskable interrupt pending. Not used by SiFive.
4	`mprven`	WARL	Not used by SiFive.
5	Reserved		
[8:6]	`cause`	RO	Indicates cause of most recent debug exception.
9	`stoptime`	WARL	0x1 will stop timers in debug mode. Not used by SiFive (timers continue).
10	`stopcount`	WARL	0x1 will stop counters in debug mode. Not used by SiFive (counters continue).
11	`stepie`	WARL	Enable interrupts when stepping. Not used by SiFive (interrupts disabled).
12	`ebreaku`	RW	EBREAK instructions in U-mode enter debug mode (vs. breakpoint exception).
13	`ebreaks`	RW	EBREAK instructions in S-mode enter debug mode.
14	Reserved		
15	`ebreakm`	RW	EBREAK instructions in M-mode enter debug mode.
[27:16]	Reserved		
[31:28]	`xdebugver`	RO	Version

Table 164: Debug Control and Status Register

16.2.2 Debug PC (`dpc`)

When entering debug mode, the current PC is copied here. When leaving debug mode, execution resumes at this PC.

16.2.3 Debug Scratch (`dscratch`)

This register is generally reserved for use by Debug ROM in order to save registers needed by the code in Debug ROM. The debugger may use it as described in *The RISC-V Debug Specification, Version 1.0*.

16.2.4 Trigger Select Register (tselect)

To support a large and variable number of triggers for tracing and breakpoints, they are accessed through one level of indirection where the `tselect` register selects which bank of three `tdata1-3` registers are accessed via the other three addresses.

The `tselect` register has the format shown below:

Trigger Select Register (tselect)			
CSR	`0x7A0`		
Bits	**Field Name**	**Attr.**	**Description**
[63:0]	`index`	WARL	Selection index of triggers

Table 165: Trigger Select Register

The `index` field is a **WARL** field that does not hold indices of unimplemented triggers. Even if `index` can hold a trigger index, it does not guarantee the trigger exists. The `type` field of `tdata1` must be inspected to determine whether the trigger exists.

16.2.5 Trigger Data Registers (tdata1-3)

The `tdata1-3` registers are 64-bit read/write registers selected from a larger underlying bank of triggers by the `tselect` register.

Trigger Data Register 1 (tdata1)			
CSR	`0x7A1`		
Bits	**Field Name**	**Attr.**	**Description**
[58:0]	Trigger-Specific Data		
59	`dmode`	WARL	Selects between debug mode (`dmode`=1) and machine mode (`dmode`=0) views of the registers, where only debug mode code can access the debug mode view of the triggers
[63:60]	`type`	WARL	The type of trigger selected by `tselect` • 0x0 - No such trigger • 0x1-0x5 - Reserved • 0x6 - Address/Data Match Trigger • ≥0x7 - Reserved

Table 166: Trigger Data Register 1

Trigger Data Registers 2 and 3 (`tdata2/3`)			
CSR	`0x7A2 - 0x7A3`		
Bits	**Field Name**	**Attr.**	**Description**
[63:0]	Trigger-Specific Data		

Table 167: Trigger Data Registers 2 and 3

Any attempt to read/write the `tdata1-3` registers in machine mode when `TSELECT.dmode`=1 raises an illegal-instruction exception.

16.3 Breakpoints

The U74-MC Core Complex supports two hardware breakpoint registers per hart, which can be flexibly shared between debug mode and machine mode.

When a breakpoint register is selected with `tselect`, the other CSRs access the following information for the selected breakpoint:

CSR Name	Breakpoint Alias	Description
`tselect`	`tselect`	Breakpoint selection index
`tdata1`	`mcontrol6`	Breakpoint match control
`tdata2`	`maddress`	Breakpoint match address
`tdata3`	N/A	Reserved

Table 168: Trigger CSRs When Used as Breakpoints

16.3.1 Breakpoint Match Control Register (`mcontrol6`)

Each breakpoint control register is a read/write register laid out in Table 169. This register is accessible as `tdata1` when `type` is 0x2.

Breakpoint Match Control Register (`mcontrol6`)				
CSR	0x7A1			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	R	WARL	0x0	Address match on load
1	W	WARL	0x0	Address match on store
2	X	WARL	0x0	Address match on instruction fetch
3	U	WARL	X	Address match on user mode
4	S	WARL	X	Address match on supervisor mode
5	Reserved	WPRI	X	
6	M	WARL	X	Address match on machine mode
[10:7]	match	WARL	X	Breakpoint Address Match type • 0x0 - Single address • 0x1 - Power-of-2 range, limited to 64 bytes in SiFive implementations • 0x2 - ≥ address • 0x3 - < address • Others not supported by SiFive
11	chain	WARL	0x0	Chain adjacent conditions. When set, this trigger and the next must match at the same time to fire. Typically used for a range breakpoint using 2 triggers, one with `match`=0x2 and one with `match`=0x3. This is not a sequential trigger.
[15:12]	action	WARL	0x0	Breakpoint action to take
[19:16]	size	WARL	0x0	Size of the breakpoint. Fixed at 0, meaning accesses of any size that cover any part of the trigger address range will fire.
20	timing	WARL	0x0	Timing of the breakpoint. Fixed at 0, meaning breaks happen just before the event.
21	select	WARL	0x0	Perform match on address or data. Fixed at 0, meaning all triggers compare addresses only (no data value).
[58:22]	Reserved			
59	dmode	RW	0x0	Debug-only access mode
[63:60]	type	RO	0x6	Address/Data match type, always 0x6

Table 169: Breakpoint Match Control Register

The `type` field is a 4-bit read-only field holding the value 0x2 to indicate this is a breakpoint containing address match logic.

The `action` field is a 4-bit read-write **WARL** field that specifies the available actions when the address match is successful. The value 0 generates a breakpoint exception. The value 1 enters debug mode. Other actions are not implemented.

The `R`/`W`/`X` bits are individual **WARL** fields, and if set, indicate an address match should only be successful for loads, stores, and instruction fetches, respectively. All combinations of implemented bits must be supported.

The `M`/`S`/`U` bits are individual **WARL** fields, and if set, indicate that an address match should only be successful in the machine, supervisor and user modes, respectively. All combinations of implemented bits must be supported.

The `match` field is a 4-bit read-write **WARL** field that encodes the type of address range for breakpoint address matching. Three different `match` settings are currently supported: exact, NAPOT, and arbitrary range. A single breakpoint register supports both exact address matches and matches with address ranges that are naturally aligned powers-of-two (NAPOT) in size. Breakpoint registers can be paired to specify arbitrary exact ranges, with the lower-numbered breakpoint register giving the byte address at the bottom of the range and the higher-numbered breakpoint register giving the address 1 byte above the breakpoint range and using the `chain` bit to indicate both must match for the action to be taken.

NAPOT ranges make use of low-order bits of the associated breakpoint address register to encode the size of the range as follows:

`maddress`	**Match Type and Size**
`a…aaaaaa`	Exactly 1 byte
`a…aaaaa0`	2-byte NAPOT range
`a…aaaa01`	4-byte NAPOT range
`a…aaa011`	8-byte NAPOT range
`a…aa0111`	16-byte NAPOT range
`a…a01111`	32-byte NAPOT range
…	
`a01…1111`	2^{31}-byte NAPOT range

Table 170: NAPOT Size Encoding

`maskmax6` determines the largest supported NAPOT range. The value of `maskmax6` is the logarithm base 2 of the number of bytes in the largest supported NAPOT range. A value of 0 indicates that only exact address matches are supported (1-byte range). A value of 31 corresponds to the maximum NAPOT range, which is 2^{31} bytes in size. The largest range is encoded in `maddress` with the 30 least-significant bits set to 1, bit 30 set to 0, and bit 31 holding the only address bit considered in the address comparison.

The value of `maskmax6` is not directly observable, but can be determined via the following sequence:

1. Set `match` to 1 to select NAPOT mode
2. Read `match`. If the returned value is not 1, then NAPOT matching is not supported.
3. Write all ones to `tdata2`
4. Read `tdata2`. The value of `maskmax6` is one more than the index of the most-significant zero bit. For example, if the read value is `0xFFFF_FFF7`, bit 3 is zero, so `maskmax6` is 4.

To provide breakpoints on an exact range, two neighboring breakpoints can be combined with the `chain` bit. The first breakpoint can be set to match on an address using `action` of 2 (greater than or equal). The second breakpoint can be set to match on address using `action` of 3 (less than). Setting the `chain` bit on the first breakpoint prevents the second breakpoint from firing unless they both match.

> **Note**
>
> When a chain includes both instruction trigger and data address trigger, the breakpoint does not fire. To work around this limitation, set a data trigger on any access to the data item. Then, in the GDB breakpoint command script, check whether the PC is the one you want and restart if not.

16.3.2 Breakpoint Match Address Register (`maddress`)

Each breakpoint match address register is a 64-bit read/write register used to hold significant address bits for address matching and also the unary-encoded address masking information for NAPOT ranges.

16.3.3 Breakpoint Execution

Breakpoint traps are taken precisely. Implementations that emulate misaligned accesses in software will generate a breakpoint trap when either half of the emulated access falls within the address range. Implementations that support misaligned accesses in hardware must trap if any byte of an access falls within the matching range.

Debug mode breakpoint traps jump to the debug trap vector without altering machine mode registers.

Machine mode breakpoint traps jump to the exception vector with "Breakpoint" set in the `mcause` register and with `badaddr` holding the instruction or data address that caused the trap.

16.3.4 Sharing Breakpoints Between Debug and Machine Mode

When debug mode uses a breakpoint register, it is no longer visible to machine mode (that is, the `tdrtype` will be 0). Typically, a debugger will leave the breakpoints alone until it needs them, either because a user explicitly requested one or because the user is debugging code in ROM.

16.4 Debug Memory Map

This section describes the Debug Module's memory map when accessed via the regular system interconnect. The Debug Module is only accessible to debug code running in debug mode on a hart (or via a Debug Transport Module). The following addresses are offsets from the base address of the Debug Module. Note that the PMP must allow M-mode access to the Debug Module address range for debugging to be possible.

16.4.1 Debug RAM and Program Buffer (`0x300–0x3FF`)

The U74-MC Core Complex has 16 32-bit words of program buffer for the debugger to direct a hart to execute arbitrary RISC-V code. Its location in memory can be determined by executing `aiupc` instructions and storing the result into the program buffer.

The U74-MC Core Complex has two 32-bit words of debug data RAM. Its location can be determined by reading the `DM.hartinfo` register, as described in *The RISC-V Debug Specification, Version 1.0*. This RAM space is used to pass data for the Access Register abstract command, as described in *The RISC-V Debug Specification, Version 1.0*. The U74-MC Core Complex supports only general-purpose register access when harts are halted. All other commands must be implemented by executing from the debug program buffer.

In the U74-MC Core Complex, both the program buffer and debug data RAM are general-purpose RAM and are mapped contiguously in the Core Complex memory space. Therefore, additional data can be passed in the program buffer, and additional instructions can be stored in the debug data RAM.

Debuggers must not execute program buffer programs that access any Debug Module memory except defined program buffer and debug data addresses.

16.4.2 Debug ROM (`0x800–0xFFF`)

This ROM region holds the debug routines on SiFive systems. The actual total size may vary between implementations.

16.4.3 Debug Flags (`0x100–0x110`, `0x400–0x7FF`)

The flag registers in the Debug Module are used for the Debug Module to communicate with each hart. These flags are set and read used by the debug ROM and should not be accessed by any program buffer code. The specific behavior of the flags is not further documented here.

16.4.4 Safe Address

In the U74-MC Core Complex, the Debug Module contains the Debug Module address range in the memory map. Memory accesses to these addresses raise access exceptions, unless the hart is in debug mode. This property allows a "safe" location for unprogrammed parts, as the default `mtvec` location is `0x0`.

16.5 Debug Module Interface

The SiFive Debug Module (DM) conforms to *The RISC-V Debug Specification, Version 1.0*. A debug probe or agent connects to the Debug Module through the Debug Module Interface (DMI). The following sections describe notable spec options used in the implementation and should be read in conjunction with *The RISC-V Debug Specification, Version 1.0*.

DMI is a simple read/write bus whose master is the DTM (if it exists, otherwise DMI passes through to customer logic) and whose slave is the Debug Module. The master sends a request to the slave and the slave responds with a response. A request is considered sent if `req_ready`=1 indicating the master is sending a request and `req_valid`=1 indicating the slave is accepting the request on this cycle. Similarly, the response is sent when both `resp_valid`=1 indicating the slave is sending a response and `resp_ready`=1 indicating the master is accepting it.

> **Note**
>
> It is the responsibility of the debugger to simulate virtual address accesses by accessing the page tables directly, then sending the translated physical address to hardware when doing the access.

> **Note**
>
> The Debug Module registers are not directly accessible from the core.

Group	Signal	Source	Description
System	`clock`	system	All signals timed to this clock. With JTAG DTM, this clock is the JTAG TCK.
	`reset`	system	Synchronous reset. Generated by power-on reset circuit.
Request Bus	`req_ready`	slave	Slave ready to receive request.
	`req_valid`	master	Master's request valid.
	`req_addr`	master	Configurable width address bus. 0x7 for SiFive.
	`req_data`	master	32-bit write data bus.
	`req_op`	master	• 0x0 = None • 0x1 = Read • 0x2 = Write • 0x3 = Reserved
Response Bus	`resp_ready`	master	Master is ready to receive response.
	`resp_valid`	slave	Slave response is valid.
	`resp_data`	slave	32-bit read data bus.
	`resp_op`	slave	• 0x0 = Success • 0x1 = Failure • 0x2 = Not used • 0x3 = Reserved

Table 171: Debug Module Interface Signals

16.5.1 Debug Module Status Register (`dmstatus`)

`dmstatus` holds the DM version number and other implementation information. Most importantly, it contains status bits that indicate the current state of the selected hart(s).

Debug Module Status Register (`dmstatus`)				
DMI Address		`0x11`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[3:0]	`version`	RO	`0x3`	Implementation version number
4	Reserved	RO	`0x0`	
5	`hasresethaltreq`	RO	`0x1`	1 if `resethaltreq` exists
[7:6]	Reserved	RO	`0x0`	
8	`anyhalted`	RO	`0x0`	Any currently selected hart is halted. Note that this field is volatile and may be masked in IP-XACT.
9	`allhalted`	RO	`0x0`	All currently selected harts are halted. Note that this field is volatile and may be masked in IP-XACT.
10	`anyrunning`	RO	`0x1`	Any currently selected hart is running. Note that this field is volatile and may be masked in IP-XACT.
11	`allrunning`	RO	`0x1`	All currently selected harts are running. Note that this field is volatile and may be masked in IP-XACT.
12	`anyunavail`	RO	`0x0`	Any currently selected hart is not available (i.e., is powered down). DM supports it, but not currently used by SiFive cores Note that this field is volatile and may be masked in IP-XACT.
13	`allunavail`	RO	`0x0`	All currently selected harts are not available (i.e., is powered down). DM supports it, but not currently used by SiFive cores. Note that this field is volatile and may be masked in IP-XACT.
14	`anynonexistent`	RO	`0x0`	Any currently selected hart does not exist in the system. Note that this field is volatile and may be masked in IP-XACT.
15	`allnonexistent`	RO	`0x0`	All currently selected harts do not exist in the system. Note that this field is volatile and may be masked in IP-XACT.
16	`anyresumeack`	RO	`0x1`	Any currently selected hart has resumed execution. Note that this field is volatile and may be masked in IP-XACT.
17	`allresumeack`	RO	`0x1`	All currently selected harts have resumed execution. Note that this field

Table 172: Debug Module Status Register

				is volatile and may be masked in IP-XACT.
18	`anyhavereset`	RO	`0x0`	Any currently selected hart has been reset, but reset has not been acknowledged. Note that this field is volatile and may be masked in IP-XACT.
19	`allhavereset`	RO	`0x0`	All currently selected harts have been reset, but reset has not been acknowledged. Note that this field is volatile and may be masked in IP-XACT.
[21:20]	Reserved	RO	`0x0`	
22	`impebreak`	RO	`0x0`	1 if `PROGBUF` is followed by implicit `EBREAK`. Generally, 1 for E2 cores, 0 otherwise.
[31:23]	Reserved	RO	`0x0`	

Table 172: Debug Module Status Register

16.5.2 Debug Module Control Register (`dmcontrol`)

A debugger performs most hart controls through the `dmcontrol` register.

Debug Module Control Register (`dmcontrol`)				
DMI Address		`0x10`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	`dmactive`	RW	`0x0`	0 disables the DM and sets DMI registers to their reset state, 1 puts the DM in operational mode. Drives `dmactive` output that could be used by a system power controller to maintain power to the DM while it is being used. When 1, `dmcontrol` should be read back until `dmactive`=1, which indicates that the Debug Module is fully operational. When 0, the DM TileLink clock is gated off to save power.
1	`ndmreset`	RW	`0x0`	Write 1 to reset system (assert `ndreset` output). Write 0 to operate normally.
2	`clrresethaltreq`	WO	`0x0`	Write 1 to clear the reset-halt-request bit
3	`setresethaltreq`	WO	`0x0`	When written to 1, the core will halt upon the next deassertion of its reset
[15:4]	Reserved	RO	`0x0`	
[25:16]	`hartsel`	RW	`0x0`	Selects the hart to operate on
26	`hasel`	RW	`0x0`	Selects hart(s) in the hart-array mask register (`hawindow`)
27	Reserved	RO	`0x0`	
28	`ackhavereset`	WO	`0x0`	Write 1 to acknowledge that a reset occurred on the selected hart
29	Reserved	RO	`0x0`	
30	`resumereq`	WO	`0x0`	Write 1 to request selected hart to resume, cleared to 0 automatically when hart resumes
31	`haltreq`	RW	`0x0`	Write 1 to request selected hart to halt. Generates debug interrupt to the core. Write 0 once `halted` has been set by the DM.

Table 173: Debug Module Control Register

16.5.3 Hart Info Register (`hartinfo`)

`hartinfo` contains information about the currently selected hart.

Hart Info Register (`hartinfo`)				
DMI Address		`0x12`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[11:0]	`dataaddr`	RO	`0x380`	Address of DATA registers in hart memory map. 0x380 for SiFive.
[15:12]	`datasize`	RO	`0x2`	Number of DATA registers. 0x1 for 32-bit, 0x2 for 64-bit SiFive cores.
16	`dataaccess`	RO	`0x1`	DATA registers are shadowed in the hart memory map. 1 for SiFive.
[19:17]	Reserved	RO	`0x0`	
[23:20]	`nscratch`	RO	`0x1`	Number of `dscratch` registers available for debugger. 1 for SiFive.
[31:24]	Reserved	RO	`0x0`	

Table 174: Hart Info Register

16.5.4 Hart Array Window Register (`hawindow`)

This register contains a bitmap where bit 0 corresponds to hart 0, bit 1 to hart 1, etc. Any bits set in this register select the corresponding hart in addition to the hart selected by `dmcontrol.hartsel`.

16.5.5 Abstract Control and Status Register (`abstractcs`)

Abstract Control and Status Register (`abstractcs`)				
DMI Address		`0x16`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[3:0]	`datacount`	RO	`0x2`	Number of DATA registers. 0x1 for 32-bit, 0x2 for 64-bit SiFive cores.
[7:4]	Reserved	RO	`0x0`	
[10:8]	`cmderr`	RW1C	`0x0`	Non-zero value indicates an abstract command error. Remains set until cleared by writing all ones. If set, no abstract commands are accepted. • 0x0 - No error • 0x1 - Busy. Abstract command or register was accessed while command was running. • 0x2 - Not supported. Abstract command type not supported by hardware was attempted. • 0x3 - Exception. An exception occurred during execution of an abstract command. • 0x4 - Halt/resume. Abstract command attempted while hart was running or unavailable. • 0x5 - Bus. Bus error occurred during abstract command. Not used by SiFive. • 0x7 - Other. Abstract command failed for another reason. Not used by SiFive. Note that this field is volatile and may be masked in IP-XACT.
11	Reserved	RO	`0x0`	
12	`busy`	RO	`0x0`	Reads as 1 while Abstract command is running, 0 if not. Note that this field is volatile and may be masked in IP-XACT.
[23:13]	Reserved	RO	`0x0`	
[28:24]	`progbufsize`	RO	`0x10`	Number of 32-bit words in `PROGBUF`. U74-MC Core Complex has 16 words.
[31:29]	Reserved	RO	`0x0`	

Table 175: Abstract Control and Status Register

16.5.6 Abstract Command Register (`command`)

Abstract Command Register (`command`)			
DMI Address		`0x17`	
Bits	**Field Name**	**Attr.**	**Description**
[15:0]	`regno`	RW	Select which register to read/write. SiFive only supports GPRs: `0x1000-0x101F`.
16	`write`	RW	1=write register, 0=read register. Only done if `transfer`=1.
17	`transfer`	RW	1=do the register read/write, 0=don't.
18	`postexec`	RW	1=execute `PROGBUF` after the command, 0=don't.
19	`aarpostincrement`	RW	Not supported by SiFive.
[22:20]	`aarsize`	RW	0x2, 0x3, 0x4 select 32, 64, 128 bits, respectively.
23	Reserved	RO	
[31:24]	`cmdtype`	RW	0=Access Register is the only type supported by SiFive.

Table 176: Abstract Command Register

Note that this register is volatile and may be masked in IP-XACT.

16.5.7 Abstract Command Autoexec Register (`abstractauto`)

Abstract Command Autoexec Register (`abstractauto`)				
DMI Address		`0x18`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[11:0]	`autoexecdata`	RW	`0x0`	Bitmap of `DATA` registers [11:0]. 1 indicates `DATA` access initiates command.
[15:12]	Reserved	RO	`0x0`	
[31:16]	`autoexecprogbuf`	RW	`0x0`	Bitmap of `PROGBUF` words [15:0]. 1 indicates `PROGBUF` access initiates command.

Table 177: Abstract Command Autoexec Register

16.5.8 Debug Module Control and Status 2 Register (`dmcs2`)

Table 178 describes the Debug Module Control and Status 2 Register `dmcs2`. If halt/resume groups are not implemented, then `group` will always read back as 0. The Debug Module external triggers may be allocated as needed between halt and resume groups.

Debug Module Control and Status 2 Register (`dmcs2`)				
DMI Address		`0x32`		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	`hgselect`	RW	`0x0`	0=operate on harts, 1=operate on external triggers.
1	`hgwrite`	WO	`X`	When written with 1, the selected harts or external trigger is assigned to group `group`.
[6:2]	`group`	RW	`0x0`	Specify the halt group or resume group number that the selected harts or external triggers will be assigned to. Note that this field is volatile and may be masked in IP-XACT.
[10:7]	Reserved	RO	`0x0`	
11	`grouptype`	RW	`0x0`	0=operate on Halt Group configuration, 1=operate on Resume Group configuration.
[31:12]	Reserved	RO	`0x0`	

Table 178: Debug Module Control and Status 2 Register

16.5.9 Abstract Commands

Abstract commands provide a debugger with a path to read and write processor state and are used for extracting and modifying processor state such as registers and memory. Register `s0` is saved by the ROM and is available for use by the abstract command code. An abstract command is started by the debugger writing to `command`. In `command`, the debugger selects whether to load/store a register, execute `PROGBUF`, or both. Only GPR register transfers are supported currently. Many aspects of Abstract Commands are optional in *The RISC-V Debug Specification, Version 1.0* and are implemented as described below.

`cmdtype`	Feature	Support
Access Register	GPR registers	Access Register command, register number `0x1000` - `0x101F`
	CSR registers	Not supported. CSRs are accessed using the Program Buffer.
	FPU registers	Not supported. FPU registers are accessed using the Program Buffer.
	Autoexec	Both `autoexecprogbuf` and `autoexecdata` are supported.
	Post-increment	Not supported.
	Core Register Access	Not supported.
Quick Access		Not supported.
Access Memory		Not supported. Memory access is accomplished using the Program Buffer.

Table 179: Debug Abstract Commands

The use of abstract commands is outlined in the following example, describing how to read a word of target memory:

1. The debugger writes opcodes to `PROGBUF` to accomplish the desired function.
2. The debugger writes the desired memory address to `DATA[0]`.
3. The debugger requests an abstract command specifying to load `s0` from `DATA[0]`, then execute `PROGBUF`. Writing to `command` while hart n is selected has the side effect of setting `FLAGS[n].go`. Writing to `command` also sets busy which is readable from the debugger, and indicates that an abstract command is in progress.
4. The ROM busy-wait loop being executed by hart n sees `FLAGS[n].go` set.
5. ROM code writes 0 to `GOING` which has the effect of clearing `FLAGS[n].go`.
6. ROM code jumps to `WHERETO`, then `ABSTRACT` which contains the opcode `lw s0, 0(DATA)` to load `s0` from `DATA[0]`. Opcodes in `ABSTRACT` are constructed by DM hardware from command. If `command.transfer`=0, no register transfer is done and instead `ABSTRACT[0]` reads as NOP.
7. If a register read/write is all that is needed, the debugger would set `command.postexec` to 0. `ABSTRACT[1]` would then read as `EBREAK`.
8. If `command.postexec`=1, `ABSTRACT[1]` reads as NOP and execution falls through to `PROGBUF` which will have been previously written by the debugger with the opcodes `lw s0, 0(s0)`, then `sw s0, DATA(zero)`, then `EBREAK`.
9. `EBREAK` reenters ROM at address `0x800`. ROM writes `hartid` to `HALTED` which has the side effect of clearing busy, telling the debugger that the abstract command is finished.
10. The debugger reads the result from `DATA[0]`.

The autoexec feature of Abstract Commands is supported by SiFive hardware (and is used by OpenOCD for memory block read and write). Once an abstract command has been completed, the debugger can read or write a particular `DATA` or `PROGBUF` location to run the command again. For example, fast download can be accomplished by setting up PROGBUF for memory write, then repeatedly writing words to `DATA[0]`. Each write re-executes the register transfer and `PROGBUF` to store the word into memory. For a 32-bit block write, the abstract command would be set up like this:

ABSTRACT	`regno=s1`, `write=1`, `transfer=1`, `postexec=1`. DM constructs the instructions
	`lw s1,0(DATA)    // load s1 from debugger`
	`NOP              // fall thru to PROGBUF`

PROGBUF	`sw   s1, 0(s0)   // store s1 to memory`
	`addi s0, s0, 4   // increment memory pointer`
	`ebreak           // done`

Table 180: Abstract Command Example for 32-bit Block Write

16.5.10 Multi-core Synchronization

The DM is configured with one Halt Group that may be programmed to synchronize execution between harts, or between hart(s) and external logic, such as a cross-trigger matrix. The Halt Group is configured using the `dmcs2` register.

Hart Array

The Hart Array is an internal bitmap that selects a subset of the harts in a system. Debug operations such as resume and halt are automatically applied to all the selected harts simultaneously.

To configure the Hart Array:

1. Set the `hasel` bit in the Debug Module `dmcontrol` register.
2. Set bits in the `hawindow` register corresponding to the harts to be selected. Bit 0 = hart 0, Bit 1 = hart 1, etc.

The Hart Array covers debug operations initiated by the debugger but does not cover the case when harts halt due to other causes, such as breakpoints. This is handled with a Halt Group.

16.5.11 System Bus Access

System Bus Access (SBA) provides an alternative method to access memory. SBA operation conforms to *The RISC-V Debug Specification, Version 1.0* and its description is not duplicated here. It implements a bus master that connects with the bus crossbar to allow access to the device's physical address space without involving a hart to perform accesses. SBA is controlled from the DMI using registers in the range `0x37` - `0x3F`. By default, the maximum bus width supported by SBA is 64. Comparing Program Buffer memory access and SBA:

Program Buffer Memory Access	**SBA Memory Access**
Physical Address	Physical Address
Subject to Physical Memory Protection (PMP)	Not subject to PMP
Cache coherent	Cache coherent
Hart must be halted	Hart may be halted or running

Table 181: System Bus vs. Program Buffer Comparison

16.6 Debug Module Operational Sequences

The sections below describe the flow for entering into and exiting from debug mode. The user can halt and resume more than one hart at a time using the hart array mask.

16.6.1 Entering Debug Mode

To use debug mode, the DM must be enabled by writing `0x0000_0001` to `dmcontrol`.

The debugger can request a halt by writing `0x8000_0001` to `dmcontrol` to set `haltreq`. This generates a debug interrupt to the core.

The core enters debug mode and jumps to the debug interrupt handler located at `0x800` and serviced from the DM.

ROM code at `0x800` writes `hartid` into the `HALTED` register which has the effect of setting the halted bit for this hart. Halted bits are readable from the debugger and generally will be continually polled to check for breakpoints when a hart is running.

ROM code then busy-waits checking its hart-specific `FLAGS` register.

16.6.2 Exiting Debug Mode

The debugger writes 1 to `resumereq` in the `dmcontrol` register to restart execution. This clears `resumeack` and sets bit 1 of the `FLAGS` register for the selected hart.

The ROM busy-wait loop being executed by hart n sees `FLAGS[n].resume` set.

ROM code writes `hartid` to RESUMING, which has the effect of clearing `FLAGS[n].resume`, setting `resumeack`, and clearing `halted` for the hart.

ROM code then executes `dret` which returns to user code at the address currently in `dpc`.

The debugger sees `resumeack` and knows the resume was successful.

Chapter 17

Error Correction Codes (ECC)

Error correction codes (ECC) are implemented on various memories within the U74-MC Core Complex, allowing for the detection and, in some cases, correction of memory errors. The following SRAM blocks on the U74-MC Core Complex support ECC: data cache, instruction cache, DTIM, ITIM, and L2 cache.

The minimal case of an ECC error is a single-bit error that is detected, reported via interrupt handler, and corrected automatically by hardware without any software intervention. More difficult scenarios that involve double-bit errors (fatal and non-recoverable, potentially causing data loss) are still reported and tracked in hardware but are not correctable. The ECC hardware includes logic for detection and correction, in addition to 7 redundant bits per 32-bit codeword or 8 redundant bits per 64-bit codeword.

Name	Protection Type
Branch Predictor	None
ITIM	SECDED ECC (64+8b)
L1 D-Cache Data	SECDED ECC (32+7b)
L1 D-Cache Tag	SECDED ECC (32+2b)
L1 I-Cache Data	Parity-only (64+1b)
L1 I-Cache Tag	Parity-only (25+1b)
L1 I-Cache Tag	Parity-only (26+1b)
L2 Cache Data	SECDED ECC (64+8b)
L2 Directory Tag	SECDED ECC (32+7b)
UTLB	Parity-only (49+1b)

Table 182: Memory Protection Summary

17.1 ECC Configuration

All blocks with ECC support are enabled globally through the Bus-Error Unit (BEU) configuration registers. The BEU is used to configure ECC reporting and enable interrupt handling via the global or local interrupt controller. The global interrupt controller is the Platform-Level Interrupt Controller (PLIC). The local interrupt controller is the Core-Local Interruptor (CLINT). The BEU registers `plic_interrupt` and `local_interrupt` are used to route the errors to the respective interrupt controller. Additionally, the BEU can be used for TileLink bus errors.

17.1.1 ECC Initialization

Any SRAM block containing ECC functionality needs to be initialized prior to use. This does not include cache memory, since an internal state machine initializes data cache valid bits, and instruction cache valid bits are flops with reset. ECC will correct defective bits based on memory contents, so if memory is not first initialized to a known state, then the ECC will not operate as expected. It is recommended to use a DMA, if available, to write the entire SRAM or cache to zeros prior to enabling ECC reporting. If no DMA is present, use store instructions issued from the processor. Initializing memory with ECC from an external bus is not recommended. After initialization, ECC-related registers can be written to zero, and then ECC reporting can be enabled. 64-bit aligned writes are recommended.

The startup code in the `freedom-e-sdk/freedom-metal` directory of the IP deliverables provides a method to automatically initialize memory with ECC. This is accomplished using an assembly-level function `__metal_memory_scrub`, located in the file `freedom-metal/src/scrub.S`. The linker script provides the symbol `__metal_eccscrub_bit` as a flag to enable the startup code to initialize memory with ECC. It is important to note that this memory initialization is limited to 64 KB to support RTL simulation run times. If unexpected ECC errors occur, check the range of the startup initialization to ensure it covers the region used by the software application.

17.2 ECC Interrupt Handling and Error Injection

Single-bit errors are automatically repaired by the hardware.

BEU errors are always enabled and thus do not have a control bit in the `mie` (Machine Interrupt Enable) CSR. Likewise, there is no dedicated control bit for BEU errors in the `mideleg` (Machine Interrupt Delegation) CSR, so it cannot be delegated to a lower privilege mode than M-mode. Error injection, and thus software handling of errors, can be accomplished manually by writing the BEU `accrued` register. The BEU is further described in Chapter 12.

Monitoring overall ECC events can be accomplished in software via the interrupt handler.

The L2 Cache Controller contains hardware counters to track ECC events, and optionally inject ECC errors to test the software handling of ECC events. The L2 Cache Controller is further described in Chapter 14.

The exception code value is located in the `mcause` (Machine Trap Cause) CSR. When BEU interrupts are routed through the PLIC, the default exception code value will be 11 (`0xB`).

When ECC interrupts are routed through the CLINT, the default exception code value will be 128 (`0x80`). These exception codes are further detailed in Section 8.7.5.

17.3 Hardware Operation Upon ECC Error

Hardware will operate differently depending on which memory type encounters an ECC error:

- Instruction Cache: The error is corrected, and the cache line is flushed
- ITIM: Single-bit errors are corrected, and written back to the RAM
- Data Cache: The error is corrected and the cache line is invalidated and written back to the next level of memory
- DTIM: Single-bit errors are corrected and written back to the RAM.
- L2 Cache: Single-bit correction for L2 data and metadata (metadata includes index, tag, and directory information). Double-bit detection only on the L2 data array.

Double-bit errors are reported at the Core Complex boundary via the `halt_from_tile_N` signal that, if asserted, remains high until reset.

Appendix A

SiFive Core Complex Configuration Options

This section provides a reference of the key configuration options of the SiFive S7 and U7 Series cores and the larger Core Complex. The file `docs/core_complex_configuration.txt` lists the features and options configured in the U74-MC Core Complex.

A.1 S7 Series

The S7 Series comes with the following set of configuration options. Note that the configuration may be limited to a fixed set of discrete options.

Modes and ISA:

- Configurable number of Cores (1 to 8). In the case where more than one core is selected, all cores are configured the same.
- Optional support for RISC-V user mode
- Optional M, F, D, B, and Zfh extensions
    - If M extension, configurable performance (1-cycle or 4-cycle)
- Optional SiFive Custom Instruction Extension (SCIE)

On-Chip Memory:

- Optional Instruction Cache with configurable size (4 KiB to 64 KiB) and associativity (2-, 4-, or 8-way)
- Optional Instruction-Tightly Integrated Memory (ITIM) with configurable size (4 KiB to 256 KiB) and base address
- Data Tightly-Integrated Memory (DTIM) or Data Cache:
    - If DTIM, then configurable size (4 KiB to 256 KiB) and base address
    - If Data Cache, then configurable size (4 KiB to 256 KiB) and associativity (2-, 4-, 8-, or 16-way)

- Optional Data Local Store (DLS) with the following options:
  - Configurable size (4 KiB to 8 MiB)
  - Configurable base address
  - Configurable pipeline depth (0, 1, or 3 additional stages)
  - Configurable number of banks (1 to 64)
- Optional L2 Cache with the following options:
  - Configurable size (128 KiB to 4 MiB), associativity (2-, 4-, 8-, 16-, or 32-way), and banks (1, 2, or 4)
  - Configurable number of L2 Hardware Prefetcher streams (4, 8, or 16) and queue size (4, 8, 12, or 16)
  - Configurable L1 to L2 bus width (64-, 128-, or 256-bit)
- Optional Fast I/O
- Optional Address Remapper with the following options:
  - Configurable number of entries (4, 8, 16, 32, or 64)
  - "From" region with configurable size (power of 2 up to 64) and base address
  - "To" region with configurable size (power of 2 up to 64) and base address
  - Configurable remap entry size (4 B to "From" region size)
- Configurable number of MMIO registers (4 to 24)

Error Handling:

- Optional Bus-Error Unit (BEU)
- Optional ECC support

Ports:

- Optional Memory Port, System Port, Peripheral Port, Front Port, Core Local Port, and Core Local Front Port
  - Each port has a configurable base address, width (32-, 64-, or 128-bit), size (64 KiB to 2 GiB), and protocol (AHB, AHB-Lite, APB, AXI4)
  - If AXI4 protocol, configurable AXI ID width (4, 8, or 16). Front, Memory, and System Ports only.
- Optional Front Port Passthrough

Security:

- Optional Physical Memory Protection (PMP), configurable up to 16 regions

- Optional Disable Debug Input
- Optional Password-protected Debug
- Optional Public key based secure Debug
- Optional Hardware Cryptographic Accelerator (HCA) with the following options:
    - Configurable base address
    - Optional AES-128/192/256
    - Optional AES-MAC
    - Optional SHA-224/256/384/512
    - Optional True Random Number Generator (TRNG)
    - Optional Public Key Accelerator (PKA) with the following parameters:
        - Configurable PKA operation maximum width (256 or 384 bits)

SiFive Insight Debug and Trace:

- Optional Debug Module with the following options:
    - Configurable base address
    - Configurable debug interface (JTAG, cJTAG, or APB)
    - Configurable number of Hardware Breakpoints (0 to 16) and External Triggers (0 to 16)
    - Optional System Bus Access
- Configurable number of performance counters (0 to 8)
- Optional Raw Instruction Trace Port
- Optional Nexus Trace Encoder with the following options:
    - Optional Event Trace
    - Configurable Trace Encoder Format (BTM or HTM)
    - Trace Sink (SRAM, ATB Bridge, SWT, System Memory, and/or PIB)
        - If SRAM Sink, configurable Trace Buffer size (256 B to 64 KiB)
        - If PIB Sink, configurable width (1-, 2-, 3-, 5-, or 9-bit) and optional PIB clock input
    - Optional Timestamp capabilities with configurable width (40, 48, or 56 bits) and source (Bus Clock, Core Clock, or External)
    - External Trigger Inputs (0 to 8) and Outputs (0 to 8)
    - Optional Instrumentation Trace Component (ITC)
    - Optional PC Sampling

Interrupts:

- Optional Platform-Level Interrupt Controller (PLIC) with the following parameters:
    - Priority Levels (1 to 7)
    - Number of interrupts (1 to 511)
- A configurable number of Core-Local Interruptor (CLINT) interrupts (0 to 16)

Design For Test:

- Configurable SRAM user-defined inputs (0 to 1024)
- Configurable SRAM user-defined outputs (0 to 1024)
- Optional SRAM Macro Extraction
- Optional Clock Gate Extraction
- Optional Grouping and Wrapping of extracted macros

Note that the SRAM user-defined feature is mutually exclusive to the macro extraction features.

Clocks and Reset:

- Optional Clock Gating
- Configurable Reset Scheme (Synchronous, Asynchronous, Full Asynchronous)

Branch Prediction:

- Configurable Branch Prediction (Area- or Performance-Optimized)

RTL Options:

- Optional custom RTL module name prefix

WorldGuard:

- Optional WorldGuard support with the following options:
    - Configurable number of worlds (2 to 32) and base address
    - Optional, configurable WorldGuard PMPs, filters, markers, and ROM for various component

A.2 U7 Series

The U7 Series comes with the following set of configuration options. Note that the configuration may be limited to a fixed set of discrete options.

Modes and ISA:

- Configurable number of Cores (1 to 8). In the case where more than one core is selected, all cores are configured the same.
- Optional M, F, D, B, and Zfh extensions
- Optional SiFive Custom Instruction Extension (SCIE)
- Configurable Virtual Addressing Modes (Sv39 or Sv39+Sv48)

On-Chip Memory:

- Optional Instruction Cache with configurable size (4 KiB to 64 KiB) and associativity (2-, 4-, or 8-way)
- Optional Instruction-Tightly Integrated Memory (ITIM) with configurable base address
- Data Cache with configurable size (4 KiB to 256 KiB) and associativity (2-, 4-, 8-, or 16-way)
- Optional Data Local Store (DLS) with the following options:
  - Configurable size (4 KiB to 8 MiB)
  - Configurable base address
  - Configurable pipeline depth (0, 1, or 3 additional stages)
  - Configurable number of banks (1 to 64)
- Optional L2 Cache with the following options:
  - Configurable size (128 KiB to 4 MiB), associativity (2-, 4-, 8-, 16-, or 32-way), and banks (1, 2, or 4)
  - Configurable number of L2 Hardware Prefetcher streams (4, 8, or 16) and queue size (4, 8, 12, or 16)
  - Configurable L1 to L2 bus width (64-, 128-, or 256-bit)
- Configurable number of MMIO registers (4 to 24)

Error Handling:

- Optional Bus-Error Unit (BEU)
- Optional ECC support

Ports:

- Optional System Port, Peripheral Port, Front Port, Core Local Port, and Core Local Front Port
  - Each port has a configurable base address, width (32-, 64-, or 128-bit), size (64 KiB to 2 GiB), and protocol (AHB, AHB-Lite, APB, AXI4)

  - If AXI4 protocol, configurable AXI ID width (4, 8, or 16). Front, Memory, and System Ports only.
- Optional Front Port Passthrough

Security:

- Optional Physical Memory Protection (PMP), configurable up to 16 regions
- Optional Disable Debug Input
- Optional Password-protected Debug
- Optional Public key based secure Debug
- Optional Hardware Cryptographic Accelerator (HCA) with the following options:
  - Configurable base address
  - Optional AES-128/192/256
  - Optional AES-MAC
  - Optional SHA-224/256/384/512
  - Optional True Random Number Generator (TRNG)
  - Optional Public Key Accelerator (PKA) with the following parameters:
    - Configurable PKA operation maximum width (256 or 384 bits)

SiFive Insight Debug and Trace:

- Optional Debug Module with the following options:
  - Configurable base address
  - Configurable debug interface (JTAG, cJTAG, or APB)
  - Configurable number of Hardware Breakpoints (0 to 16) and External Triggers (0 to 16)
  - Optional System Bus Access
- Configurable number of performance counters (0 to 8)
- Optional Raw Instruction Trace Port
- Optional Nexus Trace Encoder with the following options:
  - Optional Event Trace
  - Configurable Trace Encoder Format (BTM or HTM)
  - Trace Sink (SRAM, ATB Bridge, SWT, System Memory, and/or PIB)
    - If SRAM Sink, configurable Trace Buffer size (256 B to 64 KiB)

        - If PIB Sink, configurable width (1-, 2-, 3-, 5-, or 9-bit) and optional PIB clock input
    - Optional Timestamp capabilities with configurable width (40, 48, or 56 bits) and source (Bus Clock, Core Clock, or External)
    - External Trigger Inputs (0 to 8) and Outputs (0 to 8)
    - Optional Instrumentation Trace Component (ITC)
    - Optional PC Sampling

Interrupts:

- Optional Platform-Level Interrupt Controller (PLIC) with the following parameters:
    - Priority Levels (1 to 7)
    - Number of interrupts (1 to 511)
- A configurable number of Core-Local Interruptor (CLINT) interrupts (0 to 16)

Design For Test:

- Configurable SRAM user-defined inputs (0 to 1024)
- Configurable SRAM user-defined outputs (0 to 1024)
- Optional SRAM Macro Extraction
- Optional Clock Gate Extraction
- Optional Grouping and Wrapping of extracted macros

Note that the SRAM user-defined feature is mutually exclusive to the macro extraction features.

Clocks and Reset:

- Optional Clock Gating
- Configurable Reset Scheme (Synchronous, Asynchronous, Full Asynchronous)

Branch Prediction:

- Configurable Branch Prediction (Area- or Performance-Optimized)

RTL Options:

- Optional custom RTL module name prefix

WorldGuard:

- Optional WorldGuard support with the following options:
    - Configurable number of worlds (2 to 32) and base address

  - Optional, configurable WorldGuard PMPs, filters, markers, and ROM for various component

Appendix B

SiFive RISC-V Implementation Registers

This section provides a reference to the SiFive RISC-V implementation version registers `marchid` and `mimpid`.

B.1 Machine Architecture ID Register (`marchid`)

Value	Core Generator
`0x8000_0007`	6/7/P200/X200-Series Processor

Table 183: Core Generator Encoding of `marchid`

B.2 Machine Implementation ID Register (`mimpid`)

Value	Generator Release Version
`0x0000_0000`	Pre-19.02
`0x2019_0228`	19.02
`0x2019_0531`	19.05
`0x2019_0919`	19.08p0p0 / 19.08.00
`0x2019_1105`	19.08p1p0 / 19.08.01.00
`0x2019_1204`	19.08p2p0 / 19.08.02.00
`0x2020_0423`	19.08p3p0 / 19.08.03.00
`0x0120_0626`	19.08p4p0 / 19.08.04.00
`0x0220_0515`	koala.00.00-preview and koala.01.00-preview
`0x0220_0603`	koala.02.00-preview
`0x0220_0630`	20G1.03.00 / koala.03.00-general
`0x0220_0710`	20G1.04.00 / koala.04.00-general
`0x0220_0826`	20G1.05.00 / koala.05.00-general
`0x0320_0908`	kiwi.00.00-preview
`0x0220_1013`	20G1.06.00 / koala.06.00-general
`0x0220_1120`	20G1.07.00 / koala.07.00-general
`0x0421_0205`	llama.00.00-preview
`0x0421_0324`	21G1.01.00 / llama.01.00-general
`0x0421_0427`	21G1.02.00 / llama.02.00-general
`0x0521_0528`	mongoose.00.00-preview
`0x0521_0714`	21G2.01.00 / mongoose.01.00-general
`0x0521_1008`	21G2.02.00 / mongoose.02.00-general
`0x0621_1027`	narwhal.00.00-preview
`0x0621_1203`	narwhal.01.00-preview
`0x0621_1222`	21G3.02.00 / narwhal.02.00-general

Table 184: Generator Release Encoding of `mimpid`

Appendix C

SiFive Custom CSRs

This section provides a reference for the custom RISC-V CSRs configured in the U74-MC Core Complex.

CSR	Name	Notes
`0x7C0`	Branch Prediction Mode CSR	See Section 7.2.1 for more information
`0x7C1`	SiFive Feature Disable CSR	See Section 7.2.2 for more information
`0x7C8`	Power Dial CSR	See Section 15.2.1 for more information

Table 185: SiFive Custom CSRs

Appendix D

Floating-Point Unit Instruction Timing

This section provides a reference for the instruction timings of the single- and double-precision floating-point units in the U74-MC Core Complex.

D.1 S7 Floating-Point Instruction Timing

Single-precision floating-point unit instruction latency and repeat rates are described in Table 186.

Assembly	Operation	Latency	Repeat Rate
Sign Inject			
`fabs.s rd, rs1`	`f[rd] = |f[rs1]|`	2	1
`fsgnj.s rd, rs1, rs2`	`f[rd] = {f[rs2][31], f[rs1][30:0]}`	2	1
`fsgnjn.s rd, rs1, rs2`	`f[rd] = {~f[rs2][31], f[rs1][30:0]}`	2	1
`fsgnjx.s rd, rs1, rs2`	`f[rd] = {f[rs1][31] ^ f[rs2][31], f[rs1][30:0]}`	2	1
Arithmetic			
`fadd.s rd, rs1, rs2`	`f[rd] = f[rs1] + f[rs2]`	5	1
`fsub.s rd, rs1, rs2`	`f[rd] = f[rs1] − f[rs2]`	5	1
`fdiv.s rd, rs1, rs2`	`f[rd] = f[rs1] ÷ f[rs2]`	9–36	8–33
`fmul.s rd, rs1, rs2`	`f[rd] = f[rs1] × f[rs2]`	5	1
`fsqrt.s rd, rs1`	`f[rd] = √f[rs1]`	9–28	8–33
`fmadd.s rd, rs1, rs2, rs3`	`f[rd] = (f[rs1] × f[rs2]) + f[rs3]`	5	1
`fmsub.s rd, rs1, rs2, rs3`	`f[rd] = (f[rs1] × f[rs2]) − f[rs3]`	5	1
Negate Arithmetic			
`fneg.s rd, rs1`	`f[rd] = −f[rs1]`	2	1
`fnmadd.s rd, rs1, rs2, rs3`	`f[rd] = −(f[rs1] × f[rs2]) − f[rs3]`	5	1
`fnmsub.s rd, rs1, rs2, rs3`	`f[rd] = −(f[rs1] × f[rs2]) + f[rs3]`	5	1
Compare			
`feq.s rd, rs1, rs2`	`x[rd] = f[rs1] == f[rs2]`	4	1
`fle.s rd, rs1, rs2`	`x[rd] = f[rs1] ≤ f[rs2]`	4	1
`flt.s rd, rs1, rs2`	`x[rd] = f[rs1] < f[rs2]`	4	1
`fmax.s rd, rs1, rs2`	`f[rd] = max(f[rs1], f[rs2])`	2	1
`fmin.s rd, rs1, rs2`	`f[rd] = min(f[rs1], f[rs2])`	2	1
Categorize			
`fclass.s rd, rs1`	x[rd] = $classify_s$(f[rs1])	4	1
Convert Data Type			
`fcvt.w.s rd, rs1`	x[rd] = sext($s32_{f32}$(f[rs1]))	4	1
`fcvt.l.s rd, rs1`	x[rd] = $s64_{f32}$(f[rs1])	4	1
`fcvt.s.w rd, rs1`	f[rd] = $f32_{s32}$(x[rs1])	2	1
`fcvt.s.l rd, rs1`	f[rd] = $f32_{s64}$(x[rs1])	4	1
`fcvt.wu.s rd, rs1`	x[rd] = sext($u32_{f32}$(f[rs1]))	4	1
`fcvt.lu.s rd, rs1`	x[rd] = $u64_{f32}$(f[rs1])	4	1
`fcvt.s.wu rd, rs1`	f[rd] = $f32_{u32}$(x[rs1])	2	1
`fcvt.s.lu rd, rs1`	f[rd] = $f32_{u64}$(x[rs1])	4	1
Move			
`fmv.s rd, rs1`	`f[rd] = f[rs1]`	2	1
`fmv.w.x rd, rs1`	`f[rd] = x[rs1][31:0]`	2	1
`fmv.x.w rd, rs1`	`x[rd] = sext(f[rs1][31:0])`	1	1
Load/Store			
`flw rd, offset(rs1)`	`f[rd] = M[x[rs1] + sext(offset)][31:0]`	2	1
`fsw rs2, offset(rs1)`	`M[x[rs1] + sext(offset)] = f[rs2][31:0]`	4	1

Table 186: S7 Single-Precision FPU Instruction Latency and Repeat Rates

*Instruction and data are in the instruction cache and data cache, respectively.

D.2 U7 Floating-Point Instruction Timing

Single-precision floating-point unit instruction latency and repeat rates are described in Table 187.

Assembly	Operation	Latency	Repeat Rate
Sign Inject			
fabs.s rd, rs1	f[rd] = |f[rs1]|	2	1
fsgnj.s rd, rs1, rs2	f[rd] = {f[rs2][31], f[rs1][30:0]}	2	1
fsgnjn.s rd, rs1, rs2	f[rd] = {~f[rs2][31], f[rs1][30:0]}	2	1
fsgnjx.s rd, rs1, rs2	f[rd] = {f[rs1][31] ^ f[rs2][31], f[rs1][30:0]}	2	1
Arithmetic			
fadd.s rd, rs1, rs2	f[rd] = f[rs1] + f[rs2]	5	1
fsub.s rd, rs1, rs2	f[rd] = f[rs1] − f[rs2]	5	1
fdiv.s rd, rs1, rs2	f[rd] = f[rs1] ÷ f[rs2]	9–36	8–33
fmul.s rd, rs1, rs2	f[rd] = f[rs1] × f[rs2]	5	1
fsqrt.s rd, rs1	f[rd] = √f[rs1]	9–28	8–33
fmadd.s rd, rs1, rs2, rs3	f[rd] = (f[rs1] × f[rs2]) + f[rs3]	5	1
fmsub.s rd, rs1, rs2, rs3	f[rd] = (f[rs1] × f[rs2]) − f[rs3]	5	1
Negate Arithmetic			
fneg.s rd, rs1	f[rd] = −f[rs1]	2	1
fnmadd.s rd, rs1, rs2, rs3	f[rd] = −(f[rs1] × f[rs2]) − f[rs3]	5	1
fnmsub.s rd, rs1, rs2, rs3	f[rd] = −(f[rs1] × f[rs2]) + f[rs3]	5	1
Compare			
feq.s rd, rs1, rs2	x[rd] = f[rs1] == f[rs2]	4	1
fle.s rd, rs1, rs2	x[rd] = f[rs1] ≤ f[rs2]	4	1
flt.s rd, rs1, rs2	x[rd] = f[rs1] < f[rs2]	4	1
fmax.s rd, rs1, rs2	f[rd] = max(f[rs1], f[rs2])	2	1
fmin.s rd, rs1, rs2	f[rd] = min(f[rs1], f[rs2])	2	1
Categorize			
fclass.s rd, rs1	x[rd] = classify$_{S}$(f[rs1])	4	1
Convert Data Type			
fcvt.w.s rd, rs1	x[rd] = sext(s32$_{f32}$(f[rs1]))	4	1
fcvt.l.s rd, rs1	x[rd] = s64$_{f32}$(f[rs1])	4	1
fcvt.s.w rd, rs1	f[rd] = f32$_{s32}$(x[rs1])	2	1
fcvt.s.l rd, rs1	f[rd] = f32$_{s64}$(x[rs1])	4	1
fcvt.wu.s rd, rs1	x[rd] = sext(u32$_{f32}$(f[rs1]))	4	1
fcvt.lu.s rd, rs1	x[rd] = u64$_{f32}$(f[rs1])	4	1
fcvt.s.wu rd, rs1	f[rd] = f32$_{u32}$(x[rs1])	2	1
fcvt.s.lu rd, rs1	f[rd] = f32$_{u64}$(x[rs1])	4	1
Move			
fmv.s rd, rs1	f[rd] = f[rs1]	2	1
fmv.w.x rd, rs1	f[rd] = x[rs1][31:0]	2	1
fmv.x.w rd, rs1	x[rd] = sext(f[rs1][31:0])	1	1
Load/Store			
flw rd, offset(rs1)	f[rd] = M[x[rs1] + sext(offset)][31:0]	2	1
fsw rs2, offset(rs1)	M[x[rs1] + sext(offset)] = f[rs2][31:0]	4	1

Table 187: U7 Single-Precision FPU Instruction Latency and Repeat Rates

*Instruction and data are in the instruction cache and data cache, respectively.

Double-precision floating-point unit latency and repeat rates are described in Table 188.

Assembly	Operation	Latency	Repeat Rate
Sign Inject			
fabs.d rd, rs1	f[rd] = |f[rs1]|	2	1
fsgnj.d rd, rs1, rs2	f[rd] = {f[rs2][63], f[rs1][62:0]}	2	1
fsgnjn.d rd, rs1, rs2	f[rd] = {~f[rs2][63], f[rs1][62:0]}	2	1
fsgnjx.d rd, rs1, rs2	f[rd] = {f[rs1][63] ^ f[rs2][63], f[rs1][62:0]}	2	1
Arithmetic			
fadd.d rd, rs1, rs2	f[rd] = f[rs1] + f[rs2]	7	1
fsub.d rd, rs1, rs2	f[rd] = f[rs1] − f[rs2]	7	1
fdiv.d rd, rs1, rs2	f[rd] = f[rs1] ÷ f[rs2]	9–58	8–58
fmul.d rd, rs1, rs2	f[rd] = f[rs1] × f[rs2]	7	1
fsqrt.d rd, rs1	f[rd] = √f[rs1]	9–57	8–58
fmadd.d rd, rs1, rs2, rs3	f[rd] = (f[rs1] × f[rs2]) + f[rs3]	7	1
fmsub.d rd, rs1, rs2, rs3	f[rd] = (f[rs1] × f[rs2]) − f[rs3]	7	1
Negate Arithmetic			
fneg.d rd, rs1	f[rd] = −f[rs1]	2	1
fnmadd.d rd, rs1, rs2, rs3	f[rd] = −(f[rs1] × f[rs2]) − f[rs3]	7	1
fnmsub.d rd, rs1, rs2, rs3	f[rd] = −(f[rs1] × f[rs2]) + f[rs3]	7	1
Compare			
feq.d rd, rs1, rs2	x[rd] = f[rs1] == f[rs2]	4	1
fle.d rd, rs1, rs2	x[rd] = f[rs1] ≤ f[rs2]	4	1
flt.d rd, rs1, rs2	x[rd] = f[rs1] < f[rs2]	4	1
fmax.d rd, rs1, rs2	f[rd] = max(f[rs1], f[rs2])	2	1
fmin.d rd, rs1, rs2	f[rd] = min(f[rs1], f[rs2])	2	1
Categorize			
fclass.d rd, rs1	x[rd] = classify$_{d}$(f[rs1])	4	1
Convert Data Type			
fcvt.w.d rd, rs1	x[rd] = sext(s32$_{f64}$(f[rs1]))	4	1
fcvt.l.d rd, rs1	x[rd] = s64$_{f64}$(f[rs1])	4	1
fcvt.d.w rd, rs1	f[rd] = f64$_{s32}$(x[rs1])	2	1
fcvt.d.l rd, rs1	f[rd] = f64$_{s64}$(x[rs1])	6	1
fcvt.wu.d rd, rs1	x[rd] = sext(u32$_{f64}$(f[rs1]))	4	1
fcvt.lu.d rd, rs1	x[rd] = u64$_{f64}$(f[rs1])	4	1
fcvt.d.wu rd, rs1	f[rd] = f64$_{u32}$(x[rs1])	2	1
fcvt.d.lu rd, rs1	f[rd] = f64$_{u64}$(x[rs1])	6	1
fcvt.s.d rd, rs1	f[rd] = f32$_{f64}$(f[rs1])	2	1
fcvt.d.s rd, rs1	f[rd] = f64$_{f32}$(f[rs1])	2	1
Move			
fmv.d rd, rs1	f[rd] = f[rs1]	2	1
fmv.d.x rd, rs1	f[rd] = x[rs1][63:0]	6	1
fmv.x.d rd, rs1	x[rd] = f[rs1][63:0]	1	1
Load/Store			
fld rd, offset(rs1)	f[rd] = M[x[rs1] + sext(offset)][63:0]	2	1
fsd rs2, offset(rs1)	M[x[rs1] + sext(offset)] = f[rs2][63:0]	4	1

Table 188: U7 Double-Precision FPU Instruction Latency and Repeat Rates

*Instruction and data are in the instruction cache and data cache, respectively.

Appendix E

Revision History

This section describes the changes in this document between release versions.

Version	Date	Document Changes
21G3.02.00	December 22, 2021	• Initial release

Table 189: U74-MC Core Complex Manual Revision History

Appendix F

Knowledge Base Articles

The SiFive support team provides access to an index of Knowledge Base articles in order to further assist users developing with and integrating their RISC-V IP. These articles focus on topics ranging from architectural design and software development to board bring-up and implementation.

To view more than 100 articles in the SiFive Knowledge Base, access the link below:

https://sifive.atlassian.net/servicedesk/customer/portal/47/article/465732086

References

Visit the SiFive forums for support and answers to frequently asked questions: https://forums.sifive.com

[1] A. Waterman and K. Asanovic, Eds., The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2, June 2019. [Online]. Available: https://riscv.org/specifications/

[2] ——, The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 1.11, June 2019. [Online]. Available: https://riscv.org/specifications/privileged-isa/

[3] SiFive TileLink Specification Version 1.8.0, August 2019. [Online]. Available: https://sifive.com/documentation/tilelink/tilelink-spec

[4] A. Chang, D. Barbier, and P. Dabbelt, RISC-V Platform-Level Interrupt Controller (PLIC) Specification. [Online]. Available: https://github.com/riscv/riscv-plic-spec

[5] E. Edgar and T. Newsome, Eds., RISC-V Debug Support, Version 1.0, May 2021. [Online]. Available: https://github.com/riscv/riscv-debug-spec

[6] RISC-V Bit-Manipulation ISA-extensions, Version 1.0, June 2021. [Online]. Available: https://github.com/riscv/riscv-bitmanip/releases/tag/1.0.0