



# **SiFive E20 Manual**

**20G1.03.00**

© SiFive, Inc.

# SiFive E20 Manual

## Proprietary Notice

Copyright © 2018–2020, SiFive Inc. All rights reserved.

Information in this document is provided “as is,” with all faults.

SiFive expressly disclaims all warranties, representations, and conditions of any kind, whether express or implied, including, but not limited to, the implied warranties or conditions of merchantability, fitness for a particular purpose and non-infringement.

SiFive does not assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation indirect, incidental, special, exemplary, or consequential damages.

SiFive reserves the right to make changes without further notice to any products herein.

## Release Information

| Version             | Date               | Changes   |
|---------------------|--------------------|---|
| 20G1.03.00          | June 13, 2020      | <ul style="list-style-type: none"> <li>No functional changes</li> </ul>   |
| koala.02.00-preview | June 03, 2020      | <ul style="list-style-type: none"> <li>No functional changes</li> </ul>   |
| koala.01.00-preview | May 22, 2020       | <ul style="list-style-type: none"> <li>No functional changes</li> </ul>   |
| koala.00.00-preview | May 15, 2020       | <ul style="list-style-type: none"> <li>Changed clock, reset, and logic I/O ports associated with debug</li> </ul>   |
| v19.08p3p0          | April 30, 2020     | <ul style="list-style-type: none"> <li>Fixed issue in which unused logic in asynchronous crossings (as found in the Debug connection to the core) would cause CDC lint warnings</li> <li>Fixed issue in which the BASE field in the mtvec CSR did not accurately exhibit WARL behavior</li> <li>Fixed issue in which performance counters set to count both exceptions and other retirement events only counted the exceptions</li> <li>Various documentation fixes and improvements</li> </ul> |
| v19.08p2p0          | December 06, 2019  | <ul style="list-style-type: none"> <li>Fixed erratum in which the TDO pin may remain driven after reset</li> </ul>  |
| v19.08p1p0          | November 08, 2019  | <ul style="list-style-type: none"> <li>Corrected issues which caused some 2-series cores to be larger than prior releases</li> <li>Fixed erratum in which Debug.SBCS had incorrect reset value for SBACCESS</li> <li>Fixed typos and other minor documentation errors</li> </ul>  |
| v19.08p0            | September 17, 2019 | <ul style="list-style-type: none"> <li>The Debug Module memory region is no longer accessible in M-mode</li> </ul>  |
| v19.05p2            | August 26, 2019    | <ul style="list-style-type: none"> <li>Fix for errata on E2-series cores in which memory/MMIO operations had no guaranteed order</li> </ul>   |
| v19.05p1            | July 22, 2019      | <ul style="list-style-type: none"> <li>SiFive Insight is enabled</li> <li>Use AHB-Lite for external ports</li> <li>Enable debugger reads of Debug Module registers when periphery is in reset</li> <li>Fix errata to get illegal instruction exception executing DRET outside of debug mode</li> </ul>  |
| v19.05              | June 09, 2019      | <ul style="list-style-type: none"> <li>v19.05 release of the E20 Standard Core. No functional changes.</li> </ul>   |

| Version | Date              | Changes   |
|---------|-------------------|---|
| v19.02  | February 28, 2019 | <ul style="list-style-type: none"> <li>• Changed the date based release numbering system</li> <li>• Top-level module name [E20_CoreIPSub-system]</li> <li>• SiFive Insight [enabled]</li> <li>• External MEIP interrupt [distinguished from local CLIC interrupts]</li> </ul> |
| v1p0    | June 29, 2018     | <ul style="list-style-type: none"> <li>• Updates to support the initial release of the E20 Standard Core</li> <li>• Interrupts chapter now supports CLIC modes and CSRs</li> <li>• Added CLIC chapter (removed PLIC and CLINT chapters)</li> </ul>                            |
| v0p1    | February 28, 2018 | <ul style="list-style-type: none"> <li>• Initial release</li> <li>• Describes the functionality of the SiFive E20 Core Complex</li> </ul>   |

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                              | <b>6</b>  |
| 1.1      | About this Document                              | 6         |
| 1.2      | About this Release                               | 6         |
| 1.3      | E20 Overview                                     | 7         |
| 1.4      | E2 RISC-V Core                                   | 7         |
| 1.5      | Interrupts                                       | 8         |
| 1.6      | Debug Support                                    | 8         |
| 1.7      | Compliance                                       | 8         |
| <b>2</b> | <b>List of Abbreviations and Terms</b>           | <b>9</b>  |
| <b>3</b> | <b>E2 RISC-V Core</b>                            | <b>11</b> |
| 3.1      | Instruction Memory System                        | 11        |
| 3.1.1    | Execution Memory Space                           | 11        |
| 3.1.2    | Instruction Fetch Unit                           | 11        |
| 3.2      | Execution Pipeline                               | 12        |
| 3.3      | Data Memory System                               | 12        |
| 3.4      | Supported Modes                                  | 13        |
| 3.5      | Hardware Performance Monitor                     | 13        |
| 3.5.1    | Performance Monitoring Counters Reset Behavior   | 13        |
| 3.5.2    | Fixed-Function Performance Monitoring Counters   | 13        |
| 3.6      | Ports  | 14        |
| 3.6.1    | System Port                                      | 14        |
| <b>4</b> | <b>Physical Memory Attributes and Memory Map</b> | <b>15</b> |
| 4.1      | Physical Memory Attributes Overview              | 15        |
| 4.2      | Memory Map                                       | 16        |
| <b>5</b> | <b>Programmer's Model</b>                        | <b>17</b> |

|        |   |    |
|--------|---|----|
| 5.1    | Base Instruction Formats .....                                  | 17 |
| 5.2    | I Extension: Standard Integer Instructions .....                | 18 |
| 5.2.1  | R-Type (Register-Based) Integer Instructions .....              | 19 |
| 5.2.2  | I-Type Integer Instructions .....                               | 19 |
| 5.2.3  | I-Type Load Instructions .....                                  | 21 |
| 5.2.4  | S-Type Store Instructions .....                                 | 22 |
| 5.2.5  | Unconditional Jumps .....                                       | 22 |
| 5.2.6  | Conditional Branches .....                                      | 23 |
| 5.2.7  | Upper-Immediate Instructions .....                              | 24 |
| 5.2.8  | Memory Ordering Operations .....                                | 25 |
| 5.2.9  | Environment Call and Breakpoints .....                          | 25 |
| 5.2.10 | NOP Instruction .....   | 25 |
| 5.3    | M Extension: Multiplication Operations .....                    | 25 |
| 5.3.1  | Division Operations .....                                       | 26 |
| 5.4    | C Extension: Compressed Instructions .....                      | 26 |
| 5.4.1  | Compressed 16-bit Instruction Formats .....                     | 27 |
| 5.4.2  | Stack-Pointed-Based Loads and Stores .....                      | 27 |
| 5.4.3  | Register-Based Loads and Stores .....                           | 28 |
| 5.4.4  | Control Transfer Instructions .....                             | 29 |
| 5.4.5  | Integer Computational Instructions .....                        | 30 |
| 5.5    | Zicsr Extension: Control and Status Register Instructions ..... | 32 |
| 5.5.1  | Control and Status Registers .....                              | 34 |
| 5.5.2  | Defined CSRs .....  | 34 |
| 5.5.3  | CSR Access Ordering .....                                       | 38 |
| 5.5.4  | SiFive RISC-V Implementation Version Registers .....            | 38 |
| 5.6    | Base Counters and Timers .....                                  | 39 |
| 5.6.1  | Timer Register .....  | 41 |
| 5.6.2  | Timer API .....   | 41 |
| 5.7    | ABI - Register File Usage and Calling Conventions .....         | 41 |
| 5.7.1  | RISC-V Assembly .....   | 43 |
| 5.7.2  | Assembler to Machine Code .....                                 | 43 |
| 5.7.3  | Calling a Function (Calling Convention) .....                   | 45 |
| 5.8    | Memory Ordering - FENCE Instructions .....                      | 48 |

|          |  |           |
|----------|--|-----------|
| 5.9      | Boot Flow.....                               | 49        |
| 5.10     | Linker File .....                            | 50        |
| 5.10.1   | Linker File Symbols .....                    | 50        |
| 5.11     | RISC-V Compiler Flags .....                  | 52        |
| 5.11.1   | arch, abi, and mtune .....                   | 52        |
| 5.12     | Compilation Process .....                    | 55        |
| 5.13     | Large Code Model Workarounds .....           | 55        |
| 5.13.1   | Workaround Example #1.....                   | 56        |
| 5.13.2   | Workaround Example #2.....                   | 57        |
| 5.14     | Pipeline Hazards.....                        | 58        |
| 5.14.1   | Read-After-Write Hazards .....               | 58        |
| 5.14.2   | Write-After-Write Hazards .....              | 58        |
| <b>6</b> | <b>Custom Instructions.....</b>              | <b>59</b> |
| 6.1      | CEASE .....                                  | 59        |
| 6.2      | PAUSE .....                                  | 59        |
| 6.3      | Other Custom Instructions .....              | 59        |
| <b>7</b> | <b>Interrupts and Exceptions.....</b>        | <b>60</b> |
| 7.1      | Interrupt Concepts .....                     | 60        |
| 7.2      | Exception Concepts .....                     | 61        |
| 7.3      | Trap Concepts.....                           | 62        |
| 7.4      | Interrupt Block Diagram .....                | 63        |
| 7.5      | Local Interrupts.....                        | 64        |
| 7.6      | Interrupt Operation.....                     | 64        |
| 7.6.1    | Interrupt Entry and Exit .....               | 65        |
| 7.6.2    | Critical Sections in Interrupt Handlers..... | 65        |
| 7.7      | Interrupt Control and Status Registers ..... | 65        |
| 7.7.1    | Machine Status Register (mstatus) .....      | 66        |
| 7.7.2    | Machine Trap Vector (mtvec).....             | 66        |
| 7.7.3    | Machine Interrupt Enable (mie) .....         | 68        |
| 7.7.4    | Machine Interrupt Pending (mip) .....        | 69        |
| 7.7.5    | Machine Cause (mcause).....                  | 69        |

|           |   |
|-----------|---|
|           | 4   |
| 7.7.6     | Machine Trap Vector Table (mtvt)..... 71                    |
| 7.7.7     | Handler Address and Interrupt-Enable (mnxti)..... 72        |
| 7.7.8     | Machine Interrupt Status (mintstatus) ..... 72              |
| 7.7.9     | Minimum Interrupt Configuration ..... 72                    |
| 7.8       | Interrupt Latency ..... 73                                  |
| <b>8</b>  | <b>Core-Local Interrupt Controller (CLIC)..... 74</b>       |
| 8.1       | CLIC Interrupt Levels, Priorities, and Preemption ..... 75  |
| 8.2       | CLIC Vector Table ..... 76                                  |
| 8.2.1     | CLIC Vector Table Software Example ..... 76                 |
| 8.3       | CLIC Interrupt Sources ..... 77                             |
| 8.4       | CLIC Interrupt Attribute ..... 78                           |
| 8.4.1     | CLIC Preemption Interrupt Attribute ..... 78                |
| 8.5       | Details for CLIC Modes of Operation..... 79                 |
| 8.6       | Memory Map ..... 79   |
| 8.7       | Register Descriptions ..... 80                              |
| 8.7.1     | Changes to CSRs in CLIC Mode..... 80                        |
| 8.7.2     | CLIC Interrupt Pending Register (clicIntIP) ..... 81        |
| 8.7.3     | CLIC Interrupt Enable Register (clicIntIE) ..... 81         |
| 8.7.4     | CLIC Interrupt Configuration Register (clicIntCfg) ..... 82 |
| 8.7.5     | CLIC Configuration Register (cliccfg) ..... 82              |
| <b>9</b>  | <b>Power Management ..... 84</b>                            |
| 9.1       | Hardware Reset..... 84                                      |
| 9.2       | Early Boot Flow ..... 84                                    |
| 9.3       | Interrupt State During Early Boot ..... 85                  |
| 9.4       | Other Boot Time Considerations..... 86                      |
| 9.5       | Power-Down Flow..... 86                                     |
| <b>10</b> | <b>Debug ..... 88</b>                                       |
| 10.1      | Debug CSRs ..... 88   |
| 10.1.1    | Trace and Debug Register Select (tselect)..... 88           |
| 10.1.2    | Trace and Debug Data Registers (tdata1-3) ..... 89          |

|           |   |
|-----------|---|
|           | 5   |
| 10.1.3    | Debug Control and Status Register (dcsr) ..... 90           |
| 10.1.4    | Debug PC (dpc)..... 90                                      |
| 10.1.5    | Debug Scratch (dscratch) ..... 90                           |
| 10.2      | Breakpoints ..... 90  |
| 10.2.1    | Breakpoint Match Control Register (mcontrol) ..... 90       |
| 10.2.2    | Breakpoint Match Address Register (maddress)..... 92        |
| 10.2.3    | Breakpoint Execution ..... 92                               |
| 10.2.4    | Sharing Breakpoints Between Debug and Machine Mode ..... 93 |
| 10.3      | Debug Memory Map ..... 93                                   |
| 10.3.1    | Debug RAM and Program Buffer (0x300–0x3FF) ..... 93         |
| 10.3.2    | Debug ROM (0x800–0xFFF) ..... 93                            |
| 10.3.3    | Debug Flags (0x100–0x110, 0x400–0x7FF) ..... 94             |
| 10.3.4    | Safe Zero Address..... 94                                   |
| 10.4      | Debug Module Interface..... 94                              |
| 10.4.1    | DM Registers ..... 94                                       |
| 10.4.2    | Abstract Commands ..... 95                                  |
| <b>11</b> | <b>Appendix</b> ..... 96                                    |
| 11.1      | Appendix A..... 96  |
| 11.1.1    | E2 Series..... 96   |
| <b>12</b> | <b>References</b> ..... 98                                  |

# Chapter 1

## Introduction

SiFive's E20 is an efficient implementation of the RISC-V RV32IMC architecture. The SiFive E20 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 E20 can be found in Table 1.

| E20 Feature Set      |                         |
|----------------------|-------------------------|
| Feature              | Description             |
| Number of Harts      | 1 Hart.                 |
| E2 Core              | 1 × E2 RISC-V core.     |
| Hardware Breakpoints | 4 hardware breakpoints. |

**Table 1:** E20 Feature Set

The E20 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 Section 11.1.

### 1.1 About this Document

This document describes the functionality of the E20. To learn more about the production deliverables of the E20, consult the E20 User Guide.

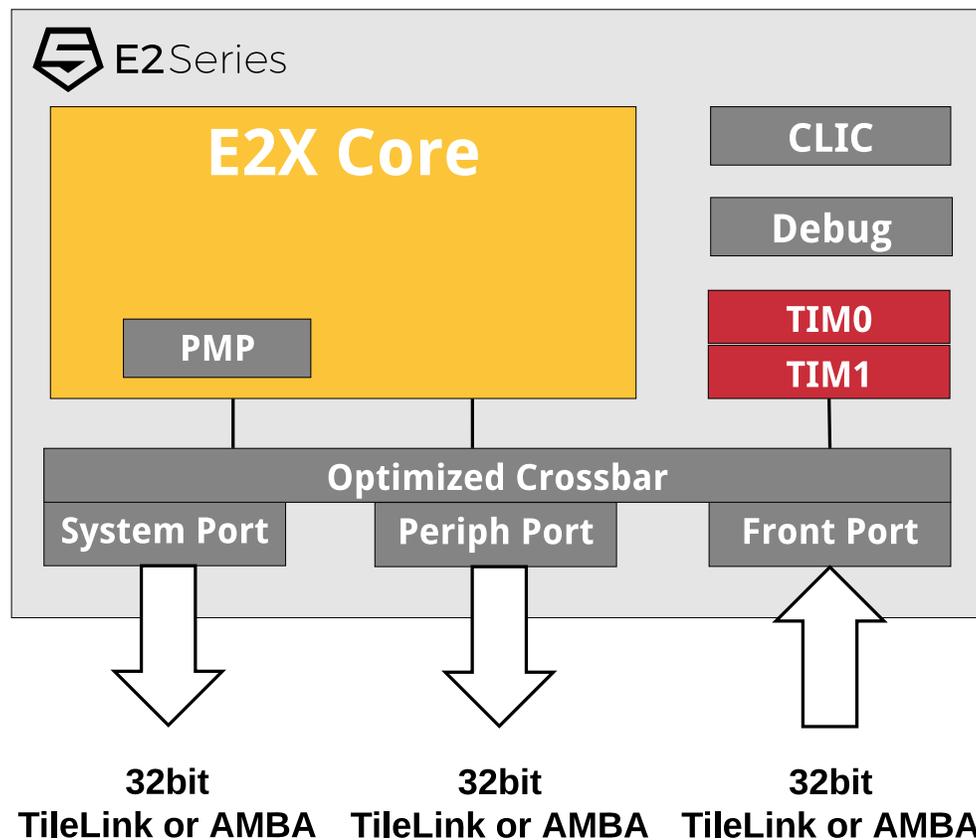
### 1.2 About this Release

This is a general release of the E20, with a supported life cycle of two years from the release date. Contact [support@sifive.com](mailto:support@sifive.com) if you have any questions.

### 1.3 E20 Overview

The E20 includes 1 × E2 32-bit RISC-V core, along with the necessary functional units required to support the core. These units include a Core-Local Interrupt Controller (CLIC) to support local interrupts, a Debug unit to support a JTAG-based debugger host connection, and a local crossbar that integrates the various components together.

An overview of the SiFive E20 is shown in Figure 1.



**Figure 1:** E20 Block Diagram

The E20 memory map is detailed in Section 4.2, and the interfaces are described in full in the E20 User Guide.

### 1.4 E2 RISC-V Core

The E20 includes a 32-bit E2 RISC-V core, which has an efficient, single-issue, in-order execution pipeline, with a peak execution rate of one instruction per clock cycle. The E2 core supports machine mode only, as well as standard Multiply (M) and Compressed (C) RISC-V extensions (RV32IMC).

The core is described in more detail in Chapter 3.

## 1.5 Interrupts

The E20 supports 32 core-local interrupts, in addition to the RISC-V architecturally-defined software, timer, and external interrupts. The Core-Local Interrupt Controller (CLIC) is used to set interrupt levels and priorities, and can support up to 16 interrupt levels.

Interrupts are described in Chapter 7. The CLIC is described in Chapter 8.

## 1.6 Debug Support

The E20 provides external debugger support over an industry-standard JTAG port, including 4 hardware-programmable breakpoints per hart.

Debug support is described in detail in Chapter 10, and the debug interface is described in the E20 User Guide.

## 1.7 Compliance

The E20 is compliant to the following versions of the various RISC-V specifications:

| <b>ISA</b>          | <b>Version</b> | <b>Ratified</b> | <b>Frozen</b> |
|---------------------|----------------|-----------------|---------------|
| RV32I               | 2.1            | Y               |               |
| <b>Extensions</b>   | <b>Version</b> | <b>Ratified</b> | <b>Frozen</b> |
| Multiplication (M)  | 2.0            | Y               |               |
| Compressed (C)      | 2.0            | Y               |               |
| <b>Devices</b>      | <b>Version</b> | <b>Ratified</b> | <b>Frozen</b> |
| Debug specification | 0.13           | Y               |               |

## **Chapter 2**

# **List of Abbreviations and Terms**

| <b>Term</b>     | <b>Definition</b>  |
|-----------------|--|
| <b>AES</b>      | Advanced Encryption Standard   |
| <b>BHT</b>      | Branch History Table   |
| <b>BTB</b>      | Branch Target Buffer   |
| <b>CBC</b>      | Cipher Block Chaining  |
| <b>CCM</b>      | Counter with CBC-MAC   |
| <b>CFM</b>      | Cipher FeedBack  |
| <b>CLIC</b>     | Core-Local Interrupt Controller. Configures priorities and levels for core-local interrupts.   |
| <b>CLINT</b>    | Core-Local Interruptor. Generates per hart software interrupts and timer interrupts.   |
| <b>CTR</b>      | CounTeR mode   |
| <b>DTIM</b>     | Data Tightly Integrated Memory   |
| <b>ECB</b>      | Electronic Code Book   |
| <b>GCM</b>      | Galois/Counter Mode  |
| <b>hart</b>     | HARdware Thread  |
| <b>IJTP</b>     | Indirect-Jump Target Predictor   |
| <b>ITIM</b>     | Instruction Tightly Integrated Memory  |
| <b>JTAG</b>     | Joint Test Action Group  |
| <b>LIM</b>      | Loosely-Integrated Memory. Used to describe memory space delivered in a SiFive Core Complex that is not tightly integrated to a CPU core.  |
| <b>OFB</b>      | Output FeedBack  |
| <b>PLIC</b>     | Platform-Level Interrupt Controller. The global interrupt controller in a RISC-V system.   |
| <b>PMP</b>      | Physical Memory Protection   |
| <b>RAS</b>      | Return-Address Stack   |
| <b>RO</b>       | Used to describe a Read-Only register field.   |
| <b>RW</b>       | Used to describe a Read/Write register field.  |
| <b>SHA</b>      | Secure Hash Algorithm  |
| <b>TileLink</b> | A free and open interconnect standard originally developed at UC Berkeley.   |
| <b>TRNG</b>     | True Random Number Generator   |
| <b>WARL</b>     | Write-Any, Read-Legal field. A register field that can be written with any value, but returns only supported values when read.   |
| <b>WIRI</b>     | 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.                                       |
| <b>WLRL</b>     | 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.  |
| <b>WPRI</b>     | 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. |
| <b>WO</b>       | Used to describe a Write-Only registers field.   |

## Chapter 3

# E2 RISC-V Core

This chapter describes the 32-bit E2 RISC-V processor core, instruction fetch and execution unit, data memory system, and external interfaces.

The E2 feature set is summarized in Table 2.

| Feature                                    | Description      |
|--|------------------|
| ISA  | RV32IMC          |
| Core Interfaces                            | 1 core interface |
| Modes                                      | Machine mode     |
| SiFive Custom Instruction Extension (SCIE) | Not Present      |

**Table 2:** E2 Feature Set

### 3.1 Instruction Memory System

This section describes the instruction memory system of the E2 core.

#### 3.1.1 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.

See Section 4.2 for a description of the executable regions of the E20.

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

#### 3.1.2 Instruction Fetch Unit

The E2 instruction fetch unit is responsible for keeping the pipeline fed with instructions from memory. 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 E2 implements the standard Compressed (C) extension to the RISC-V architecture, which allows for 16-bit RISC-V instructions. As two 16-bit instructions can be fetched per cycle, the instruction fetch unit is often idle when executing programs mostly comprised 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.

The instruction fetch unit always accesses memory sequentially. Conditional branches are predicted not-taken, and not-taken branches incur no penalty. Taken branches and unconditional jumps incur a one-cycle penalty if the target is naturally aligned, i.e., all 16-bit instructions and 32-bit instructions whose address is divisible by 4; or a two-cycle penalty if the target is not naturally aligned.

## 3.2 Execution Pipeline

The E2 execution unit is a single-issue, in-order pipeline. The pipeline comprises two stages: Instruction Fetch, described in the previous section, and Execute.

The pipeline has a peak execution rate of one instruction per clock cycle. Bypass paths are included so that most instructions have a one-cycle result latency. There are some exceptions:

- The number of stall cycles between a load instruction and the use of its result is equal to the number of cycles between the bus request and bus response. In particular, if a load is satisfied the cycle after it is demanded, then there is one stall cycle between the load and its use. In this special case, the stall can be obviated by scheduling an independent instruction between the load and its use.
- Integer multiplication instructions have a latency of four cycles. Multiplication operations can be interrupted, so they have no effect on worst-case interrupt latency.
- Integer division instructions have variable latency of at most 32 cycles. Division operations can be interrupted, so they have no effect on worst-case interrupt latency.

In the Execute stage of the pipeline, instructions are decoded and checked for exceptions, and their operands are read from the integer register file. Arithmetic instructions compute their results in this stage, whereas memory-access instructions compute their effective addresses and send their requests to the bus interface.

In this stage, instructions write their results to the integer register file. Instructions that take more than one cycle to produce their results will interlock the pipeline. In particular, load and division instructions with result latency greater than one cycle will interlock the pipeline.

## 3.3 Data Memory System

The data memory system consists of the ports shown in the Memory Map in Section 4.2.

The E2 pipeline allows for two outstanding memory accesses. Store instructions incur no stalls if acknowledged by the bus on the cycle after they are sent. Otherwise, the pipeline will interlock on the next memory-access instruction until the store is acknowledged. Misaligned accesses are not allowed to any memory region and result in a trap to allow for software emulation.

## 3.4 Supported Modes

The E2 supports RISC-V machine mode only.

See *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10* for more information on the privilege modes.

## 3.5 Hardware Performance Monitor

The E2 processor core supports a basic hardware performance monitoring (HPM) facility. The performance monitoring faculty consists of a set of fixed counters and their counter-enable registers. The registers are available to control the behavior of the counters. Performance monitoring can be useful for multiple purposes, from optimization to debug.

### 3.5.1 Performance Monitoring Counters Reset Behavior

At system reset, the hardware performance monitor counters are not reset and thus have an arbitrary value. Users can write desired values to the counter control and status registers (CSRs) to start counting at the given, known value.

### 3.5.2 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 E2 processor core contains two fixed-function performance monitoring counters.

#### Fixed-Function Cycle Counter (`mcyc1e`)

The fixed-function performance monitoring counter `mcyc1e` holds a count of the number of clock cycles the hart has executed since some arbitrary time in the past. The `mcyc1e` counter is read-write and 64 bits wide. Reads of `mcyc1e` return the lower 32 bits, while reads of `mcyc1eh` return the upper 32 bits of the 64-bit `mcyc1e` counter.

### **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 the lower 32 bits, while reads of `minstreth` return the upper 32 bits of the 64-bit `minstret` counter.

## **3.6 Ports**

This section describes the Port interfaces to the E2 core.

### **3.6.1 System Port**

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

Consult Section 4.1 for further information about the System Port and its Physical Memory Attributes.

See the E20 User Guide for a description of the System Port implementation in the E20.

Note that the System Port does not support Atomic instructions.

## Chapter 4

# Physical Memory Attributes and Memory Map

This chapter describes the E20 physical memory attributes and memory map.

### 4.1 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 E20 memory regions are shown in Table 3 and Table 4 for external and internal regions, respectively.

The configured memory regions of the E20 are listed with their attributes in Table 5.

| Port Type   | Access Properties    | Attributes |
|-------------|----------------------|------------|
| System Port | Read, Write, Execute | N/A        |

**Table 3:** Physical Memory Attributes for External Regions

| Region   | Access Properties | Attributes |
|----------|-------------------|------------|
| CLIC     | Read, Write       | N/A        |
| Debug    | None              | N/A        |
| Reserved | None              | N/A        |

**Table 4:** Physical Memory Attributes for Internal Regions

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

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

All accesses to the Debug Module from the core in non-Debug mode will trap.

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.

## 4.2 Memory Map

The memory map of the E20 is shown in Table 5.

| Base        | Top         | Attr. | Description           |
|-------------|-------------|-------|-----------------------|
| 0x0000_0000 | 0x0000_0FFF |       | Debug                 |
| 0x0000_1000 | 0x01FF_FFFF |       | Reserved              |
| 0x0200_0000 | 0x02FF_FFFF | RW    | CLIC                  |
| 0x0300_0000 | 0x1FFF_FFFF |       | Reserved              |
| 0x2000_0000 | 0x3FFF_FFFF | RWX   | System Port (512 MiB) |
| 0x4000_0000 | 0xFFFF_FFFF |       | Reserved              |

**Table 5:** E20 Memory Map. Physical Memory Attributes: **R**–Read, **W**–Write, **X**–Execute, **I**–Instruction Cacheable, **D**–Data Cacheable, **A**–Atomics

# Chapter 5

## Programmer's Model

The E20 implements the 32-bit RISC-V architecture. The following chapter provides a reference for programmers and an explanation of the extensions supported by RV32IMC.

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 E20 is an implementation of the RISC-V RV32IMC architecture, and is guaranteed to be compatible with all applicable RISC-V standards. RV32IMC can emulate almost any other RISC-V ISA extension (except the A extension, which requires additional hardware support for atomicity).

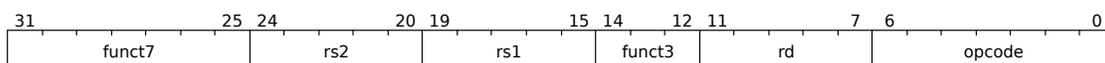
### 5.1 Base Instruction Formats

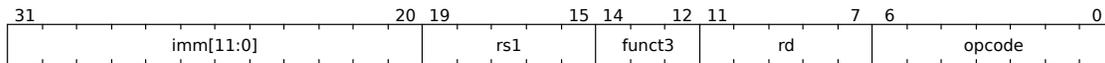
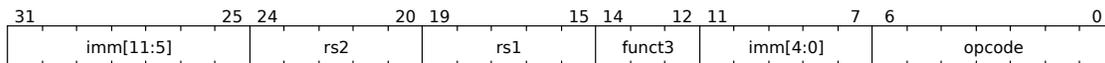
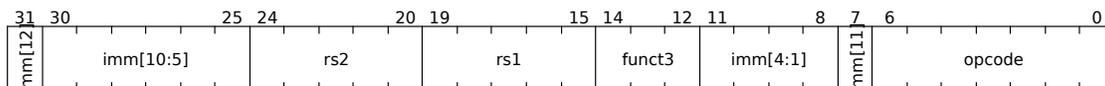
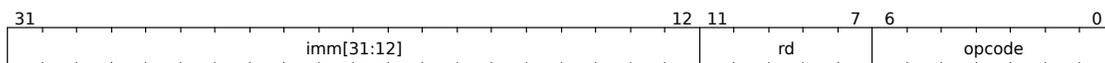
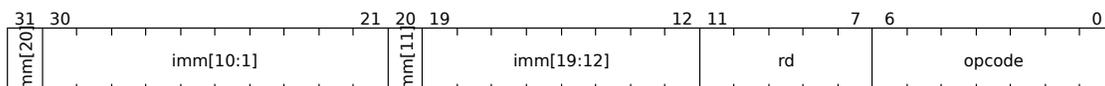
RISC-V base instructions are fixed to 32 bits in length and must be aligned on a four-byte boundary in memory. RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding, with the exception of the 5-bit immediates used in CSR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

The various formats are described in Table 6 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 6:** Base Instruction Formats



**Figure 2: R-Type****Figure 3: I-Type****Figure 4: S-Type****Figure 5: B-Type****Figure 6: U-Type****Figure 7: 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.

## 5.2 I Extension: Standard Integer Instructions

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

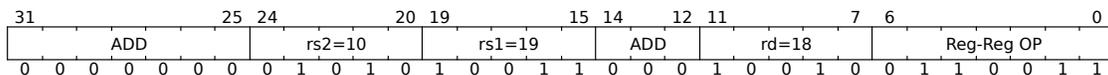
### 5.2.1 R-Type (Register-Based) Integer Instructions

| 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         |

| 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.  |

Below is an example of an ADD instruction.

**add x18, x19, x10**



**Figure 8:** ADD Instruction Example

### 5.2.2 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        |

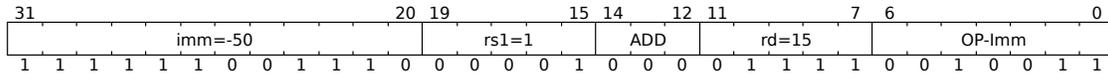
One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI).

| Instruction | Description  |
|-------------|--|
| ADDI        | Adds the sign-extended 12-bit immediate to register <i>rs1</i> . Arithmetic overflow is ignored and the result is simply the low 32-bits of the result. <code>ADDI rd, rs1, 0</code> is used to implement the <code>MV rd, rs1</code> assembler pseudoinstruction.   |
| SLTI        | Set less than immediate. Places the value 1 in register <i>rd</i> if register <i>rs1</i> is less than the sign extended immediate when both are treated as signed numbers, else 0 is written to <i>rd</i> .  |
| SLTIU       | Compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 32-bits then treated as an unsigned number). Note: <code>SLTIU rd, rs1, 1</code> sets <i>rd</i> to 1 if <i>rs1</i> equals zero, otherwise sets <i>rd</i> to 0 (assembler pseudo instruction <code>SEQZ rd, rs</code> ). |
| XORI        | Bitwise XOR on register <i>rs1</i> and the sign-extended 12-bit immediate and place the result in <i>rd</i> .  |
| ORI         | Bitwise OR on register <i>rs1</i> and the sign-extended 12-bit immediate and place the result in <i>rd</i> .   |
| ANDI        | Bitwise AND on register <i>rs1</i> and the sign-extended 12-bit immediate and place the result in <i>rd</i> .  |
| SLLI        | Shift Left Logical. The operand to be shifted is in <i>rs1</i> , 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 <i>rs1</i> , 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 <i>rs1</i> , 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).   |

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 9:** ADDI Instruction Example

### 5.2.3 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 7 below, **funct3** field encodes size and signedness of load data.

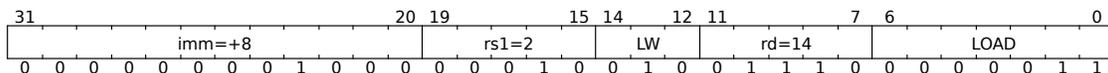
| imm       |     | funct3 |    | opcode   | Instruction |
|-----------|-----|--------|----|----------|-------------|
| imm[11:0] | rs1 | 000    | rd | 00000011 | LB          |
| imm[11:0] | rs1 | 001    | rd | 00000011 | LH          |
| imm[11:0] | rs1 | 010    | rd | 00000011 | LW          |
| imm[11:0] | rs1 | 100    | rd | 00000011 | LBU         |
| imm[11:0] | rs1 | 101    | rd | 00000011 | LHU         |

**Table 7:** 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.      |

Below is an example of a LW instruction.

**lw x14, 8(x2)**



**Figure 10:** LW Instruction Example

### 5.2.4 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.

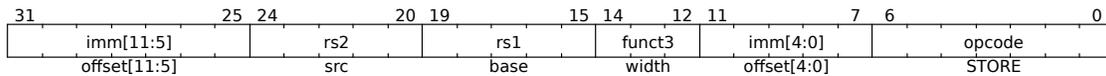


Figure 11: Store Instructions

| 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 8: S-Type Store Instructions

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

Below is an example SW instruction.

**sw x14, 8(x2)**

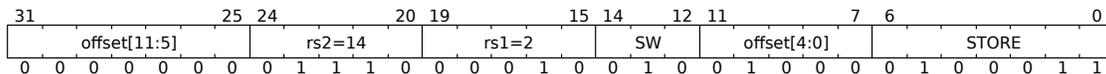


Figure 12: SW Instruction Example

### 5.2.5 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  $\pm 1$  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.

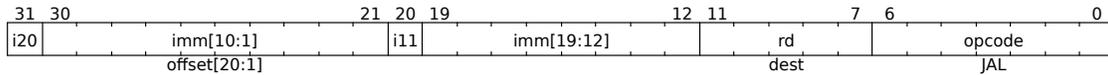


Figure 13: JAL Instruction

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.

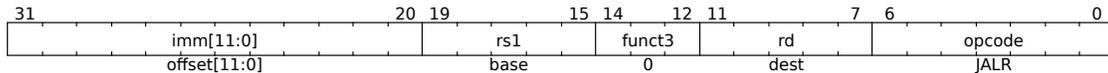


Figure 14: JALR Instruction

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 |

## 5.2.6 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  $\pm 4$  KiB.

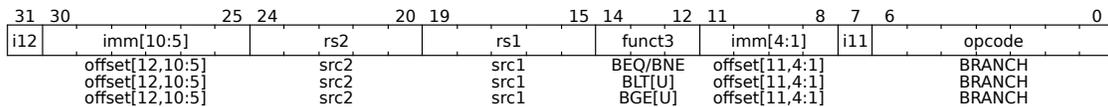


Figure 15: Branch Instructions

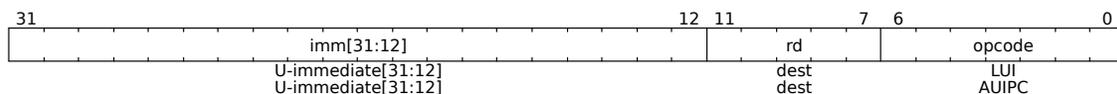
| imm          | rs2 | rs1 | func3 | imm         | opcode | Instruction |
|--------------|-----|-----|-------|-------------|--------|-------------|
| imm[12,10:5] | rs2 | rs1 | 000   | imm[4:1,11] | 110011 | BEQ         |
| imm[12,10:5] | rs2 | rs1 | 001   | imm[4:1,11] | 110011 | BNE         |
| imm[12,10:5] | 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        |

| Instruction                 | Description  |
|-----------------------------|--|
| BEQ rs1, rs2,<br>imm[12:1]  | Take the branch if registers rs1 and rs2 are equal.                |
| BNE rs1, rs2,<br>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,<br>imm[12:1]  | Take the branch if rs1 is greater than or equal to rs2.            |
| BLTU rs1, rs2,<br>imm[12:1] | Take the branch if rs1 is less than rs2 (unsigned).                |
| BGEU rs1, rs2,<br>imm[12:1] | Take the branch if rs1 is greater than or equal to rs2 (unsigned). |

**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.

| ISA Base Instruction | Assembly pseudo instruction |                  |
|----------------------|-----------------------------|------------------|
| BEQ rs, x0, offset   | beqz rs, offset             | Branch if = zero |

**5.2.7 Upper-Immediate Instructions****Figure 16:** 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.

### 5.2.8 Memory Ordering Operations

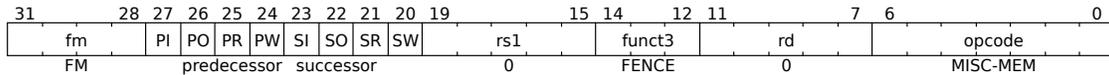


Figure 17: FENCE Instructions

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 5.8.

### 5.2.9 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.

### 5.2.10 NOP Instruction

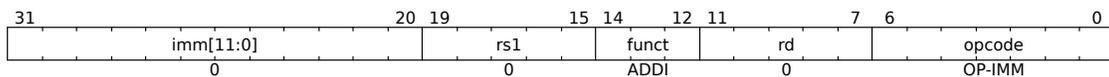


Figure 18: NOP Instructions

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**.

## 5.3 M Extension: Multiplication Operations

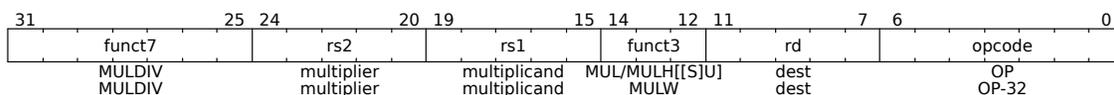


Figure 19: Multiplication Operations

| Instruction         | Description  |
|---------------------|--|
| MUL rd, rs1, rs2    | Multiplication of rs1 by rs2 and places the lower 32-bits in the destination register.       |
| MULH rd, rs1, rs2   | Multiplication that return the upper 32-bits of the full 2×32-bit product.                   |
| MULHU rd, rs1, rs2  | Unsigned multiplication that return the upper 32-bits of the full 2×32-bit product.          |
| MULHSU rd, rs1, rs2 | Signed rs1 multiple unsigned rs2 that return the upper 32-bits of the full 2×32-bit product. |

Combining MUL and MULH together creates one multiplication operation.

### 5.3.1 Division Operations

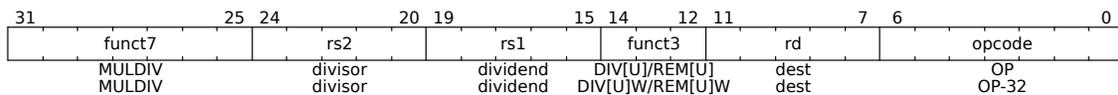


Figure 20: Division Operations

| Instruction        | Description   |
|--------------------|---|
| DIV rd, rs1, rs2   | 32-bits by 32-bits signed division of r1 by rs2 rounding towards zero.                      |
| DIVU rd, rs1, rs2  | 32-bits by 32-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.   |
| 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.  |

Combining DIV and REM together creates on division operation.

## 5.4 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.

### 5.4.1 Compressed 16-bit Instruction Formats



Figure 21: CR Format - Register

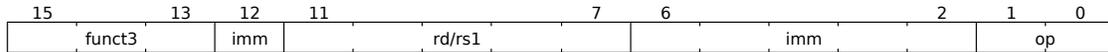


Figure 22: CI Format - Immediate

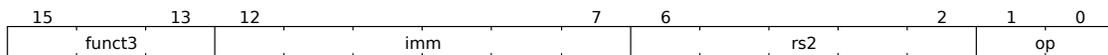


Figure 23: CSS Format - Stack-relative Store

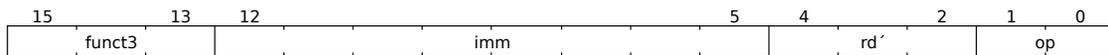


Figure 24: CIW Format - Wide Immediate

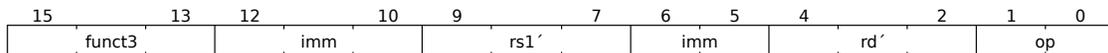


Figure 25: CL Format - Load

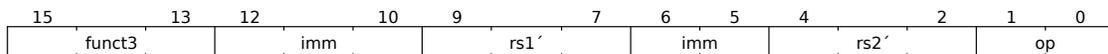


Figure 26: CS Format - Store

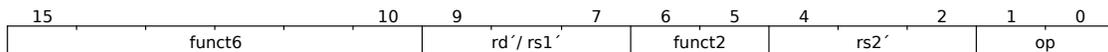


Figure 27: CA Format - Arithmetic

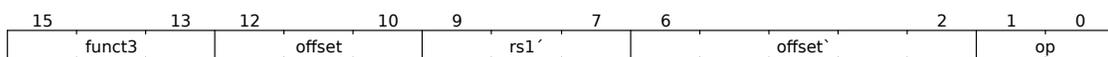


Figure 28: CJ Format - Jump

### 5.4.2 Stack-Pointed-Based Loads and Stores

The compressed load instructions are expressed in CI format.

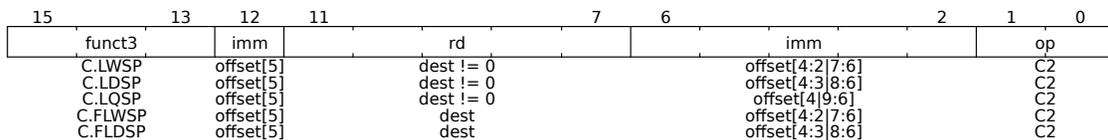


Figure 29: Stack-Pointed-Based Loads

| 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. |

The compressed store instructions are expressed in CSS format.

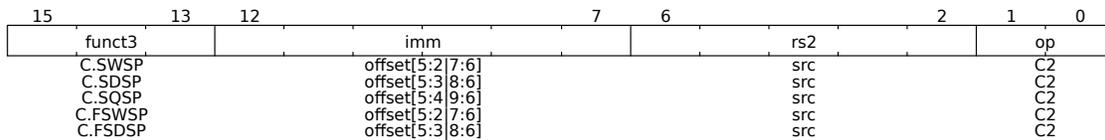


Figure 30: Stack-Pointed-Based Stores

| 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. |

### 5.4.3 Register-Based Loads and Stores

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

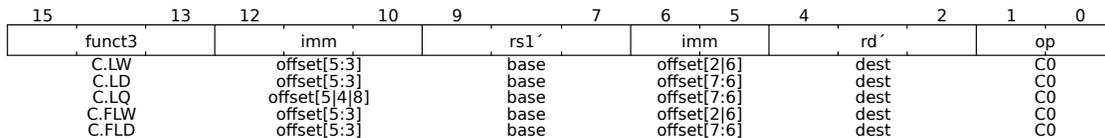


Figure 31: Register-Based Loads

| 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. |

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

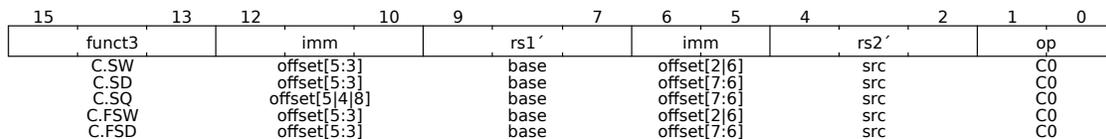


Figure 32: Register-Based Stores

| 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. |

#### 5.4.4 Control Transfer Instructions

RVC provides unconditional jump instructions and conditional branch instructions.

The unconditional jump instructions are expressed in CJ format.

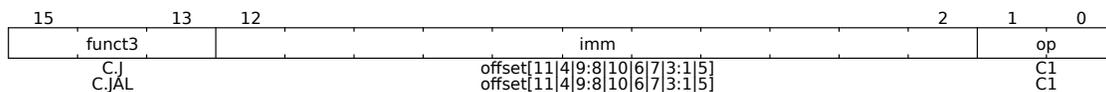
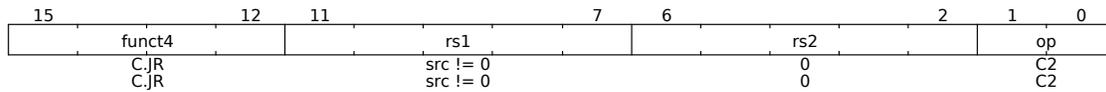


Figure 33: Unconditional Jump Instructions

| 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. |

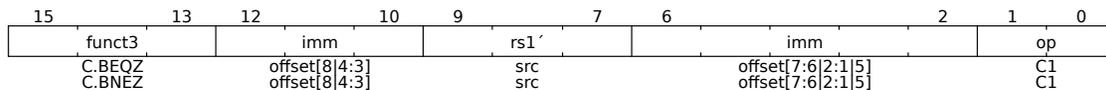
The unconditional control transfer instructions are expressed in CR format.



**Figure 34:** Unconditional Control Transfer Instructions

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

The conditional control transfer instructions are expressed in CB format.

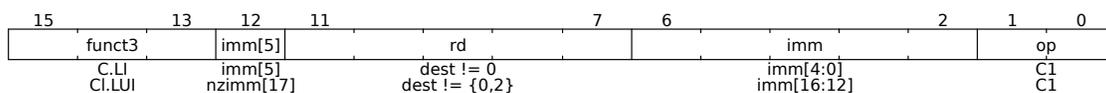


**Figure 35:** Conditional Control Transfer Instructions

| Instruction | Description  |
|-------------|--|
| C.BEQZ      | Conditional control transfers. Takes the branch if the value in register <i>rs1</i> is zero. |
| C.BNEZ      | Conditional control transfers. Takes the branch if <i>rs1</i> contains a nonzero value.      |

## 5.4.5 Integer Computational Instructions

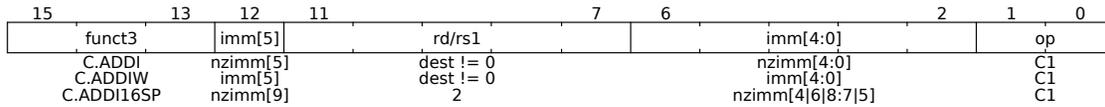
### Integer Constant-Generation Instructions



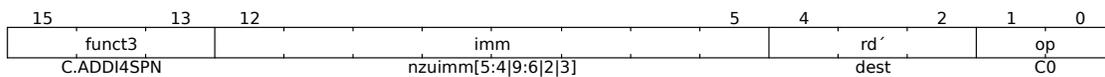
**Figure 36:** Constant Generation Instructions

| Instruction | Description  |
|-------------|--|
| C.LI        | Loads the sign-extended 6-bit immediate, <i>imm</i> , into register <i>rd</i> .  |
| 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 |

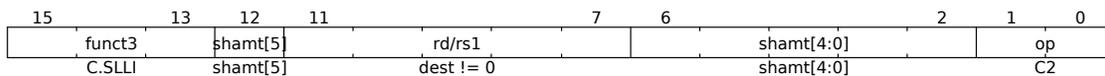
### Integer Register-Immediate Operations



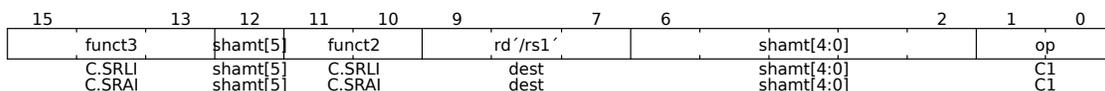
| Instruction | Description   |
|-------------|---|
| C.ADDI      | Adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd.   |
| C.ADDIW     | RV64C/RV128C instruction that performs the same computation but produces a 32-bit result, then sign-extends 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. |



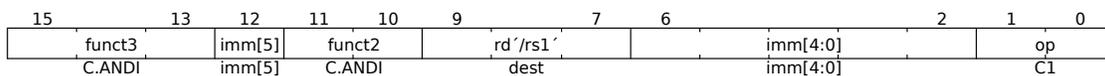
| Instruction | Description   |
|-------------|---|
| C.ADDI4SPN  | Adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, x2, and writes the result to rd'. |



| 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. |

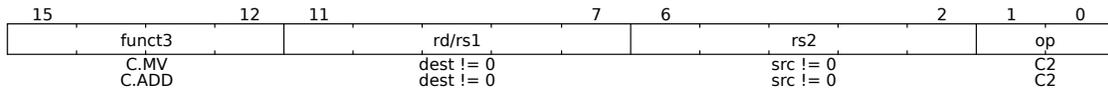


| Instruction | Description  |
|-------------|--|
| C.SRLI      | Logical right shift of the value in register rd' then writes the result to rd'. The shift amount is encoded in the shamt field.    |
| C.SRAI      | Arithmetic right shift of the value in register rd' then writes the result to rd'. The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

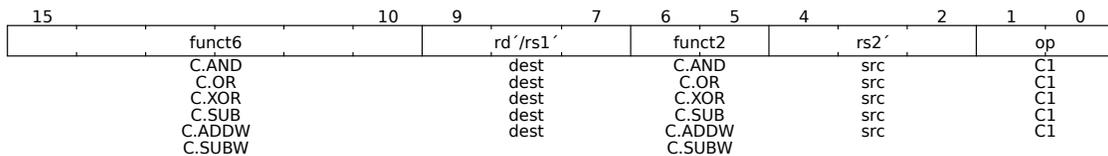


| Instruction | Description   |
|-------------|---|
| C.ANDI      | Computes the bitwise AND of the value in register rd' and the sign-extended 6-bit immediate, then writes the result to rd'. |

### 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. |



| 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 rs2'.   |
| 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. |

### Defined Illegal Instruction

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

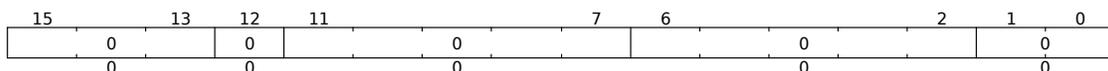
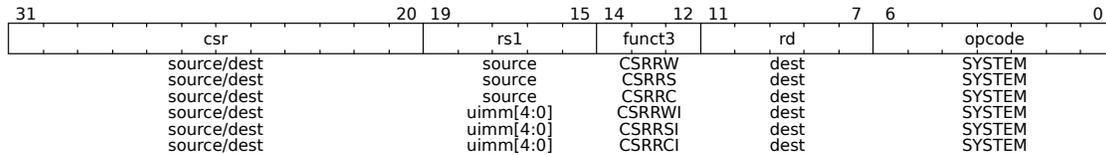


Figure 37: Defined Illegal Instruction

## 5.5 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.



**Figure 38:** 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, zeroextends the value to 32-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, zeroextends the value to 32-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 a 32-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 a 32-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.  |

The CSRRWI, CSRRSI, and CSRRCI instructions are similar in kind to CSRRW, CSRRS, and CSRRC respectively, except in that they update the CSR using a 32-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 side 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 9 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 9:** CSR Reads and Writes

### 5.5.1 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.

### 5.5.2 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, machine mode, and Debug 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. A read/write register might also contain some bits that are read-only, in which case writes to the read-only bits are ignored.

| Number                          | Privilege | Name          | Description  |
|---------------------------------|-----------|---------------|--|
| <b>User Trap Setup</b>          |           |               |  |
| 0x000                           | RW        | ustatus       | User status register.                                      |
| 0x004                           | RW        | uie           | User interrupt-enable register.                            |
| 0x005                           | RW        | utvec         | User trap handler base address.                            |
| <b>User Trap Handling</b>       |           |               |  |
| 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.                                    |
| <b>User Floating-Point CSRs</b> |           |               |  |
| 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). |
| <b>User Counter/Timers</b>      |           |               |  |
| 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.                            |
| 0xC80                           | RO        | cycleh        | Upper 32 bits of cycle, RV32I only.                        |
| 0xC81                           | RO        | timeh         | Upper 32 bits of time, RV32I only.                         |
| 0xC82                           | RO        | instreth      | Upper 32 bits of instret, RV32I only.                      |
| 0xC83                           | RO        | hpmcounter3h  | Upper 32bits of hpmcounter3, RV32I only.                   |
| 0xC84                           | RO        | hpmcounter4h  | Upper 32bits of hpmcounter4, RV32I only.                   |
|                                 |           | ...           |  |
| 0xC9F                           | RO        | hpmcounter31h | Upper 32bits of hpmcounter31, RV32I only.                  |

**Table 10:** User Mode CSRs

| Number                                       | Privilege | Name     | Description                                    |
|--|-----------|----------|--|
| <b>Supervisor Trap Setup</b>                 |           |          |  |
| 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.          |
| <b>Supervisor Trap Handling</b>              |           |          |  |
| 0x140  | RW        | sscratch | Scratch register for supervisor trap handlers. |
| 0x141  | RW        | sepc     | Supervisor exception program counter.          |
| 0x142  | RW        | scause   | Supervisor trap cause.                         |
| 0x143  | RW        | sbadaddr | Supervisor bad address.                        |
| 0x144  | RW        | sip      | Supervisor interrupt pending.                  |
| <b>Supervisor Protection and Translation</b> |           |          |  |
| 0x180  | RW        | sptbr    | Page-table base register.                      |

**Table 11:** Supervisor Mode CSRs

| Number   | Privilege | Name           | Description                                 |
|--|-----------|----------------|---|
| <b>Machine Information Registers</b>                 |           |                |   |
| 0xF11  | RO        | mvendorid      | Vendor ID.                                  |
| 0xF12  | RO        | marchid        | Architecture ID.                            |
| 0xF13  | RO        | mimpid         | Implementation ID.                          |
| 0xF14  | RO        | mhartid        | Hardware thread ID.                         |
| <b>Machine Trap Setup</b>                            |           |                |   |
| 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.          |
| <b>Machine Trap Handling</b>                         |           |                |   |
| 0x340  | RW        | mscratch       | Scratch register for machine trap handlers. |
| 0x341  | RW        | mepc           | Machine exception program counter.          |
| 0x342  | RW        | mcause         | Machine trap cause.                         |
| 0x343  | RW        | mbadaddr       | Machine bad address.                        |
| 0x344  | RW        | mip            | Machine interrupt pending.                  |
| <b>Machine Protection and Translation</b>            |           |                |   |
| 0x380  | RW        | mbase          | Base register.                              |
| 0x381  | RW        | mbound         | Bound register.                             |
| 0x382  | RW        | mibase         | Instruction base register.                  |
| 0x383  | RW        | mibound        | Instruction bound register.                 |
| 0x384  | RW        | mdbase         | Data base register.                         |
| 0x385  | RW        | mdbound        | Data bound register.                        |
| <b>Machine Counter/Timers</b>                        |           |                |   |
| 0xB00  | RW        | mcycle         | Machine cycle counter.                      |
| 0xB02  | RW        | minstret       | Machine instruction-retired counter.        |
| 0xB03  | RW        | mhpmcounter3   | Machine performance-monitoring counter.     |
| 0xB04  | RW        | mhpmcounter4   | Machine performance-monitoring counter.     |
|  |           | ...            |   |
| 0xB1F  | RW        | mhpmcounter31  | Machine performance-monitoring counter.     |
| 0xB80  | RW        | mcycleh        | Upper 32 bits of mcycle, RV32I only.        |
| 0xB82  | RW        | minstreth      | Upper 32 bits of minstret, RV32I only.      |
| 0xB83  | RW        | mhpmcounter3h  | Upper 32 bits of mhpmcounter3, RV32I only.  |
| 0xB84  | RW        | mhpmcounter4h  | Upper 32 bits of mhpmcounter4, RV32I only.  |
|  |           | ...            |   |
| 0xB9F  | RW        | mhpmcounter31h | Upper 32 bits of mhpmcounter31, RV32I only. |
| <b>Debug/Trace Register (shared with Debug Mode)</b> |           |                |   |
| 0x7A0  | RW        | tselect        | Debug/Trace trigger register select.        |
| 0x7A1  | RW        | tdata1         | First Debug/Trace trigger data register.    |

Table 12: Machine Mode CSRs

| Number | Privilege | Name   | Description                               |
|--------|-----------|--------|---|
| 0x7A2  | RW        | tdata2 | Second Debug/Trace trigger data register. |
| 0x7A3  | RW        | tdata3 | Third Debug/Trace trigger data register.  |

**Table 12:** 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 13:** Debug Mode Registers

### 5.5.3 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 5.8. 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.

### 5.5.4 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                     |
|-----------|------------------------------------|
| 0x80...02 | 2-Series Processor (E2, S2 series) |

**Table 14:** Core Generator Encoding of `marchid`

**mimpid**

The value in `mimpid` holds the release tag for the generator used to build this implementation.

**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:**

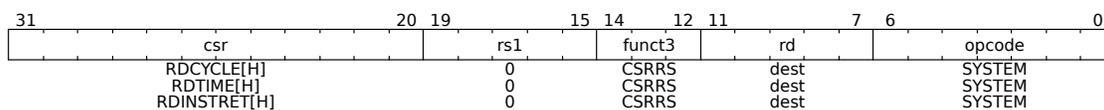
```
csrr a5, mimpid
```

## 5.6 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 32-bit read-only CSR registers 0xC00–0xC1F, with the upper 32 bits accessed via CSR registers 0xC80–0xC9F on RV32. The first three of these (CYCLE, TIME, and INSTRET) have dedicated functions; while the remaining counters, if implemented, provide programmable event counting.

The E20 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 E20 implements event counters in the form of `mhpmcounter`, which is used to monitor user requested events.



**Figure 39:** Timers & Counters

| Instruction                 | Description  |
|-----------------------------|--|
| RDCYCLE rd, rs1, cycle      | Reads the low 32-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. |
| RDCYCLEH rd, rs1, cycle     | RV32I instruction that reads bits 63–32 of the same cycle counter.   |
| RDTIME rd, rs1, time        | Reads the low 32-bits of the time CSR, which counts wall-clock real time that has passed from an arbitrary start time in the past.   |
| RDTIMEH rd, rs1, time       | RV32I-only instruction that reads bits 63–32 of the same real-time counter.  |
| RDINSTRET rd, rs1, instret  | reads the low 32-bits of the instret CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past.  |
| RDINSTRETH rd, rs1, instret | RV32I-only instruction that reads bits 63–32 of the same instruction counter.  |

RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions read the full 64 bits of the `cycle`, `time`, and `instret` counters. The RDCYCLE pseudoinstruction reads the low 32-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 32-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 32-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 5.6.2 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 |
| 0xC80  | RO        | cycleh   | Upper 32 bits of cycle, RV32 only.                    |
| 0xC81  | RO        | timeh    | Upper 32 bits of time, RV32 only.                     |
| 0xC82  | RO        | instreth | Upper 32 bits of instret, RV32 only                   |

### 5.6.1 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 E20 User Guide. On reset, `mtime` is cleared to zero.

### 5.6.2 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

## 5.7 ABI - Register File Usage and Calling Conventions

RV32IMC has 32 x registers that are each 32 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-ponter       | 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 15:** 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)
```

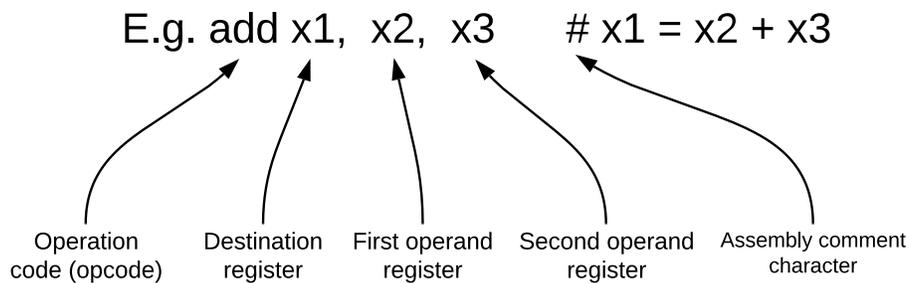
```
LW t1 , 0x7FF(gp)
```

- 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.

### 5.7.1 RISC-V Assembly

RISC-V instructions have opcodes and operands.

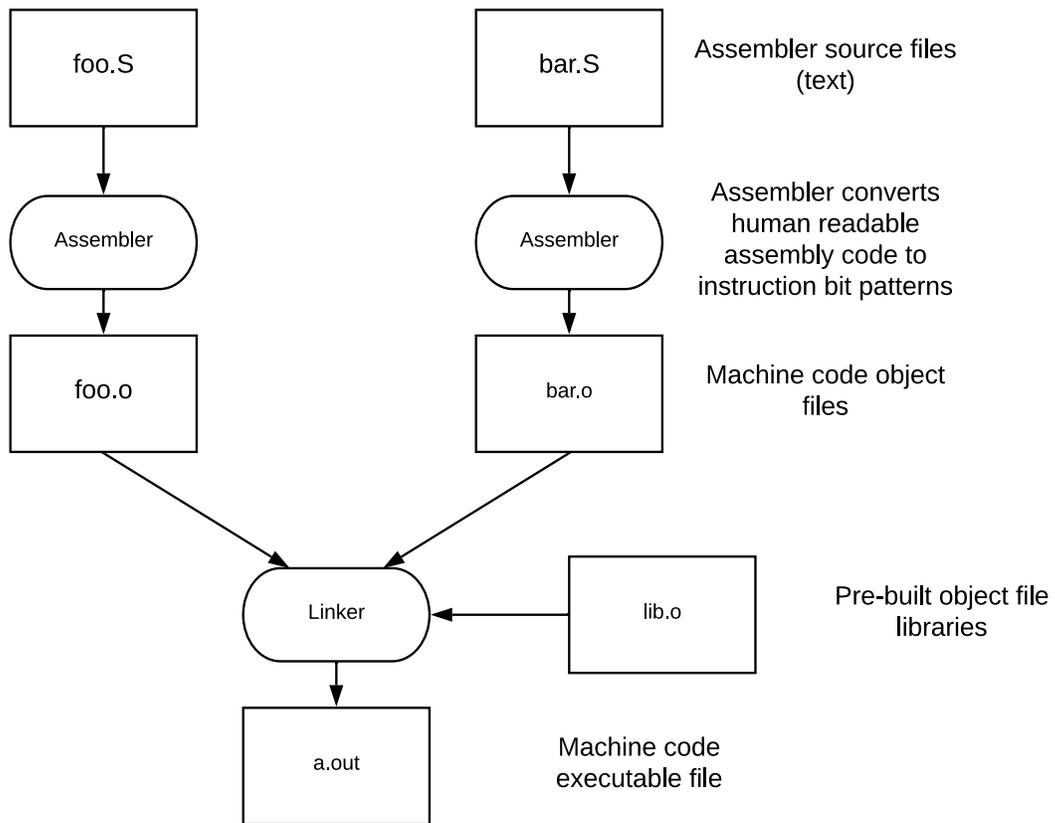


**Figure 40:** RISC-V Assembly Example

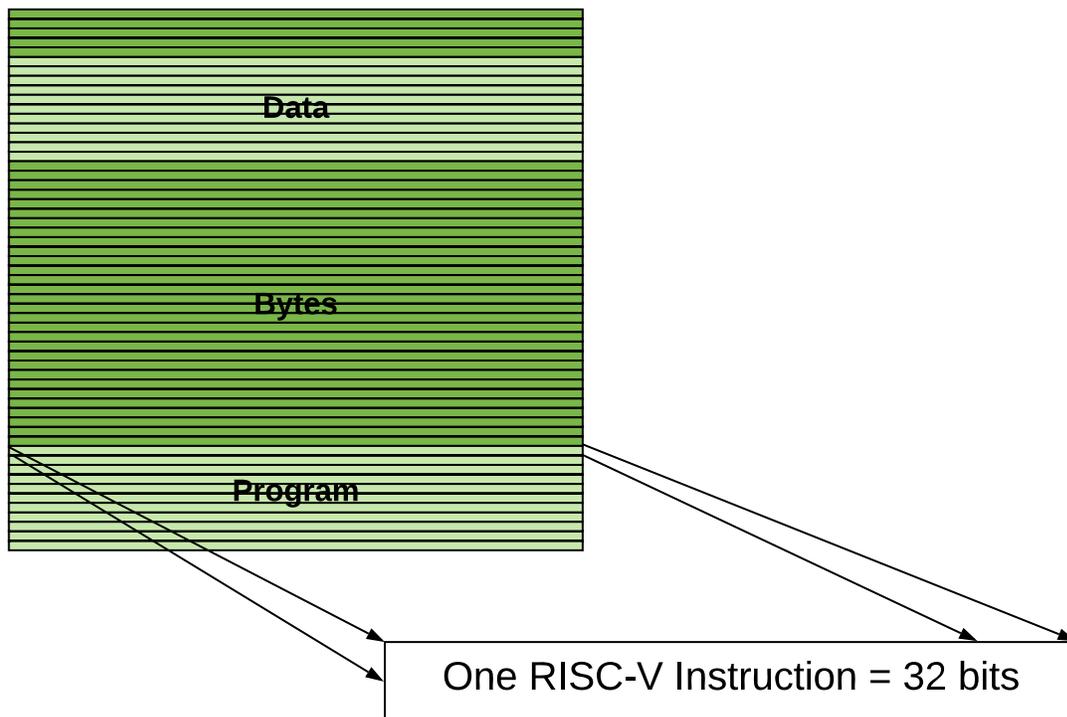
| Assembly  | C   | Description   |
|---|---|---|
| add x1, x2, x3  | a = b + c                                       | a=x1, b=x2, c=x3                                    |
| sub x3, x4, x5  | d = e - f                                       | d=x3, e=x4, f=x5                                    |
| add 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<br>add x11, x12, x10                                       | int A[100];<br>g = h + A[3];                    | Reg x10 gets A[3]<br>g=x11, h=x12                   |
| lw x10, 12(x13) # 12 = 3x4<br>add x10, x12, x10<br>sw x10, 40(x13) # 40 = 10x4        | int A[100];<br>A[10] = h + A[3];                | Reg x10 gets A[3]<br>h=x12<br>Reg x10 gets h + A[3] |
| bne x13, x14, done<br>add x10, x11, x12<br>done:                                      | if (i == j)<br>f = g + h;                       | f=x10, g=x11, h=x12, i=x13, j=x14                   |
| bne x10, x14, else<br>add x10, x11, x12<br>j done<br>else: sub x10, x11, x12<br>done: | if (i == j)<br>f = g + h;<br>else<br>f = g - h; | f=x10, g=x11, h=x12, i=x13, j=x14                   |

### 5.7.2 Assembler to Machine Code

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



**Figure 41:** RISC-V Assembly to Machine Code



### 5.7.3 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 s1 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 sp, 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:

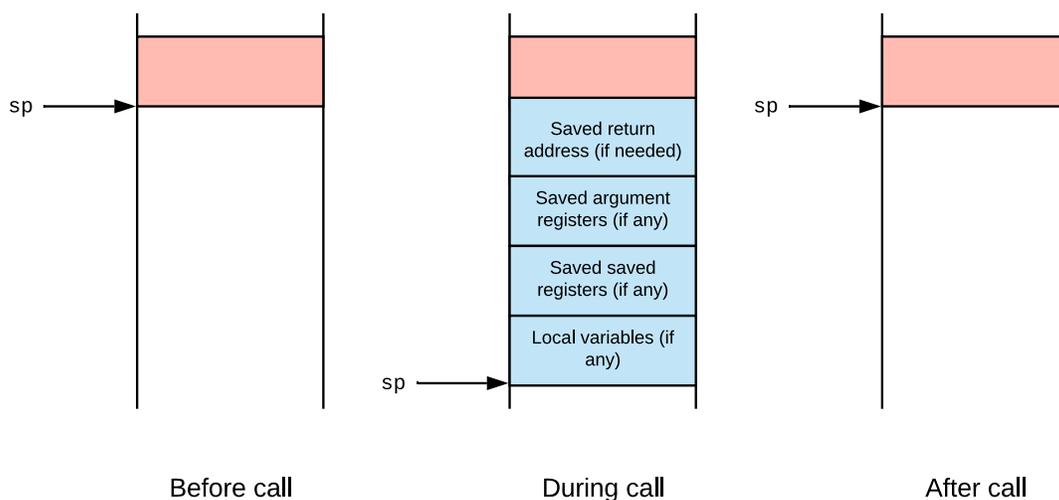
```

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 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.

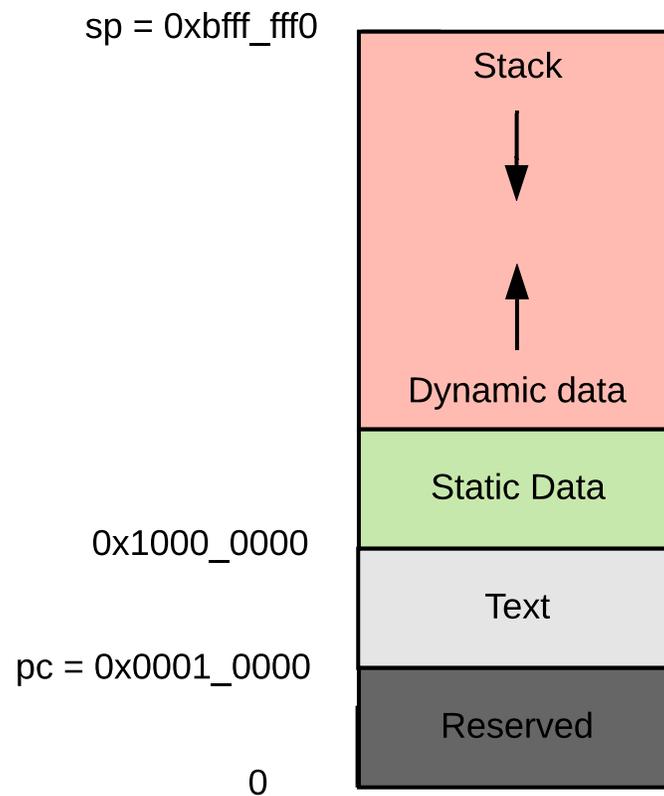


**Figure 42:** Stack Memory during Function Calls

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:...
```

### Memory Layout

**Figure 43:** RV32 Memory Layout

## 5.8 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.

## 5.9 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. Initialize global pointer `gp` register using the generated symbol `__global_pointer$`.
3. Write `mtvec` register with `early_trap_vector` as default exception handler.
4. Clear chicken bits (usage for this register is not made public).
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  $(\_sp + \_\_stack\_size \times mhartid)$ . 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, L2 if they exist in the design. This method provides full early initialization prior to entering the main application.
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 `cr0.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.

## 5.10 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=scratchpadsoftware
```

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.

### 5.10.1 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  $(\_sp + \_stack\_size \times mhartid)$

**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

## 5.11 RISC-V Compiler Flags

### 5.11.1 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 (`i1p32` or `lp64`), with an argumental single letter appended to select the floating-point registers used by the ABI (`i1p32` vs. `i1p32f` vs. `i1p32d`). 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.

- `i1p32`: `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 `i1p32`.

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 to ARM's 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 to ARM's 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 to ARM's 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 or 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
```

## 5.12 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).

## 5.13 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 toolchain and it defines a method to generate the necessary instruction combinations to access global symbols within the software program. This is done using `-mmodel=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 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 = 0x80,000,000, 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:
```

### 5.13.1 Workaround Example #1

Even if global symbols cannot be linked with the toolchain, we can still access any 64-bit addressable space using pointers. The following example is a straightforward approach to accessing data within any 64-bit addressable space:

```
// Create defines for new memory region
#define LARGE_DATA_SECTION_ADDRESS 0x100000000
#define LARGE_DATA_SECTION_SIZE_IN_BYTES 0x4000
#define DWORD_SIZE 8

int main(void) {

/*****/
```

```

    /* Example #1 - defining and accessing data outside 32-bit range using array
    pointer */
/*****
    uint32_t idx;
    uint64_t *data_array, addr;

    data_array = (uint64_t *)LARGE_DATA_SECTION_ADDRESS;
    for (addr = 0, idx = 0; addr < LARGE_DATA_SECTION_SIZE_IN_BYTES; addr +=
    DWORD_SIZE, idx++) {

        // Simply writing data to our region outside of 32-bit range
        data_array[idx] = addr;
    }
}

```

### 5.13.2 Workaround Example #2

Here we use an existing freedom-metal data structure to define a new region and API to access attributes of the region.

```

#include <metal/memory.h> // required for data struct

// Create defines for new memory region
#define LARGE_DATA_SECTION_ADDRESS 0x100000000
#define LARGE_DATA_SECTION_SIZE_IN_BYTES 0x4000
#define DWORD_SIZE 8

// Create our struct using existing metal_memory type in freedom-metal
const struct metal_memory large_data_mem_struct;
const struct metal_memory large_data_mem_struct = {
    ._base_address = LARGE_DATA_SECTION_ADDRESS,
    ._size = LARGE_DATA_SECTION_SIZE_IN_BYTES,
    ._attrs = {.R = 1, .W = 1, .X = 0, .C = 1, .A = 0},
};

int main(void) {
    // Example #2 - Creating data structure which defines 64-bit addressable regions,
    // using existing structure type to define base addr, size, and permissions

    size_t _large_data_size;
    uintptr_t _large_data_base_addr;
    int _atomics_enabled, _cachable_enabled;
    uint64_t *large_data_array;

    _large_data_base_addr = metal_memory_get_base_address(&large_data_mem_struct);
    _large_data_size = metal_memory_get_size(&large_data_mem_struct);
    _atomics_enabled = metal_memory_supports_atomics(&large_data_mem_struct);
    _cachable_enabled = metal_memory_is_cachable(&large_data_mem_struct);

    large_data_array = (uint64_t *)_large_data_base_addr;

    // Access our new memory region
    // large_data_array[x] = 0x0;
    // ... add functional code ...

```

```
    return 0;  
}
```

This example can be used if multiple data regions are required with different attributes. Once the base address is assigned from the required data structure, then pointers can be used to access memory, similar to Example #1 above. The existing struct and API format allows for multiple regions to be created easily.

## 5.14 Pipeline Hazards

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

### 5.14.1 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.

### 5.14.2 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.

## Chapter 6

# 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`.

### 6.1 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_X` signal to the outside of the Core Complex, indicating that it is safe to power down.

### 6.2 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 is simply a no-op instruction.

### 6.3 Other Custom Instructions

Other custom instructions may be implemented, but their functionality is not documented further here and they should not be used in this version of the E20.

## Chapter 7

# Interrupts and Exceptions

This chapter describes how interrupt and exception concepts in the RISC-V architecture apply to the E20.

Specifically, the E20 implements the *RISC-V Core-Local Interrupt Controller (CLIC) specification, Version 20180831*. The CLIC represents a new RISC-V interrupt specification which differs from the *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. As of June 2018, the CLIC is currently a RISC-V draft proposal of the RISC-V foundation's Fast Interrupts Task Group. Future versions of this core may implement later versions of the CLIC specification.

### 7.1 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 E20 supports machine mode interrupts.

The Core Complex also has support for the following types of RISC-V interrupts: local and global. Local interrupts are routed into the Core-Local Interrupt Controller (CLIC) where they have a dedicated interrupt exception code and programmable priority. This allows flexibility in configuring all low latency local interrupts routed into the hart from the CLIC interface. The E20 has 32 interrupts that are delivered to the core via the CLIC, along with the software and timer interrupts.

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 hart allows 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 8 describes the CLIC. The E20 does not implement a PLIC. Instead a Machine External Interrupt input signal is exposed at the boundary of the Core Complex which can be connected to a PLIC in a larger design.

## 7.2 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 16 describes the synchronous exception priority order.

| Priority       | Interrupt Exception Code | Description                       |
|----------------|--------------------------|-----------------------------------|
| <i>Highest</i> | 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                   |
| <i>Lowest</i>  | 7                        | Store/AMO access fault            |
|                | 5                        | Load access fault                 |

**Table 16:** Exception Priority

Refer to Table 23 for the full table of interrupt 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 (0x0) 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.

## 7.3 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.

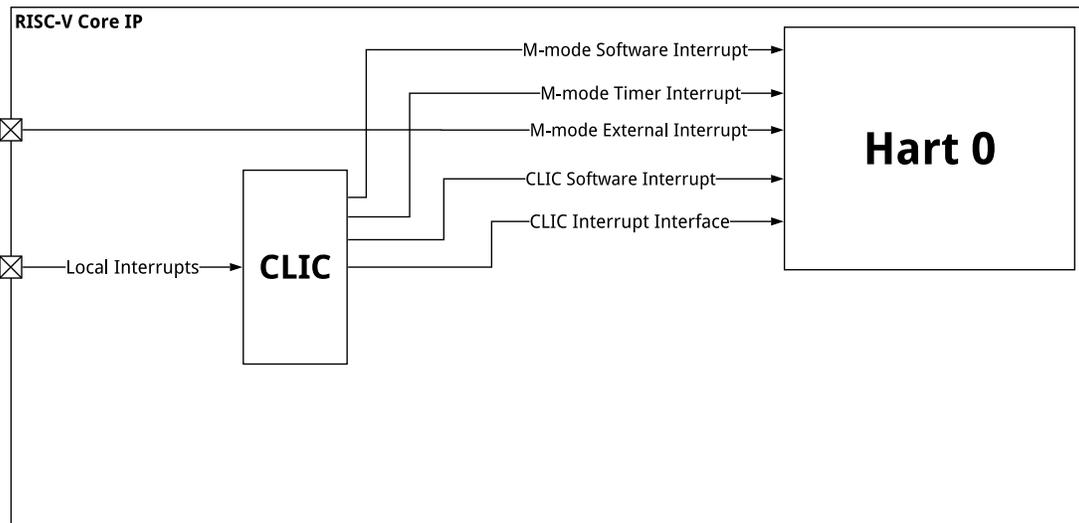
1. Interrupts should first be disabled globally by writing `mstatus.MIE` to 0, which is the default reset value.
2. Write `mtvec` CSR with the base address of the Machine mode exception handler. This is a required step in any boot flow.
3. Write `mstatus.MPP` to 0 to set the previous mode to User which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

## 7.4 Interrupt Block Diagram

The E20 interrupt architecture is depicted in Figure 44.



**Figure 44:** E20 Interrupt Architecture Block Diagram

## 7.5 Local Interrupts

Software interrupts (Interrupt ID #3) are triggered by writing the memory-mapped interrupt pending register `msip` for a particular hart when operating in CLINT modes of operation, or the `cllicIntIE` register when in CLIC modes of operation. The `msip` register is described in Table 21 and `cllicIntIE` is described in Table 31.

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 CLIC memory map. The `mtime` and `mtimecmp` registers are generally only available in machine mode, unless the PMP grants user mode access to the memory-mapped region in which they reside.

Global interrupts are usually first routed to a PLIC, then into the hart using external interrupts (Interrupt ID #11). As the E20 does not implement a PLIC, this interrupt can optionally be disabled by tying it to logic 0.

The CLIC software interrupt (Interrupt ID #12) serves a similar function as the legacy machine software interrupt, except its typical use interrupting software threads.

Local external interrupts (Interrupt ID #16–48) may connect directly to an interrupt source. The E20 has 32 local external interrupts.

## 7.6 Interrupt Operation

If the global interrupt-enable `mstatus.MIE` is clear, then no interrupts will be taken. If `mstatus.MIE` is set, then pending-enabled interrupts at a higher interrupt level 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 and interrupt level are modified to reflect the new privilege mode and interrupt level. The global interrupt-enable bit of the handler's privilege mode is cleared.

CLIC interrupt levels, priorities, and preemption are described in Section 8.1.

### 7.6.1 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.
- When in CLIC mode, the interrupted interrupt level is copied into `mcause.MPIL`, and the interrupt level is set to that of the incoming interrupt as defined in its `clicintcfg` register.
- 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 19.

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`.
- When in CLIC mode, the interrupt level is set to the value encoded in `mcause.MPIL`.
- 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.

The Control and Status Registers (CSRs) involved in handling RISC-V interrupts are described in Section 7.7.

### 7.6.2 Critical Sections in Interrupt Handlers

To implement a critical section between interrupt handlers at different levels, an interrupt handler at any interrupt level can clear global interrupt-enable bit, `mstatus.MIE`, to prevent interrupts from being taken.

## 7.7 Interrupt Control and Status Registers

The E20 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 1.10* and the *RISC-V Core-Local Interrupt Controller (CLIC) specification, Version 20180831*.

### 7.7.1 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 E20 is provided in Table 17. 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 1.10*.

| Machine Status Register |            |       |                                   |
|-------------------------|------------|-------|-----------------------------------|
| CSR                     | mstatus    |       |                                   |
| Bits                    | Field Name | Attr. | Description                       |
| [2:0]                   | Reserved   | WPRI  |                                   |
| 3                       | MIE        | RW    | Machine Interrupt Enable          |
| [6:4]                   | Reserved   | WPRI  |                                   |
| 7                       | MPIE       | RW    | Machine Previous Interrupt Enable |
| [10:8]                  | Reserved   | WPRI  |                                   |
| [12:11]                 | MPP        | RW    | Machine Previous Privilege Mode   |

**Table 17:** E20 mstatus 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 or clicIntIE, depending on CLINT or CLIC modes of operation.

Note that when operating in CLIC mode, mstatus.MPP and mstatus.MPIE are accessible in the mcause register described in Section 7.7.5.

### 7.7.2 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 E20 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 18, and the mtvec.MODE field is described in Table 19.

| Machine Trap Vector Register |            |       |  |
|------------------------------|------------|-------|--|
| CSR                          | mtvec      |       |  |
| Bits                         | Field Name | Attr. | Description  |
| [1:0]                        | MODE       | WARL  | MODE Sets the interrupt processing mode. The encoding for the E20 supported modes is described in Table 19.  |
| [31:2]                       | BASE[31:2] | WARL  | Interrupt Vector Base Address.<br>Operating in CLINT Direct Mode requires 4-byte alignment.<br>Operating in CLINT Vectored Mode requires 128-byte alignment.<br>Operating in CLIC mode requires minimum 64-byte alignment. |

Table 18: mtvec 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                            | CLIC Direct   | All interrupts and exceptions set pc to BASE.   |
| 0x3                            | CLIC Vectored | Exceptions set pc to BASE, interrupts set pc to the address in the vector table located at mtvt + (mcause.EXCCODE × 4). |

Table 19: Encoding of mtvec.MODE

Note that when in either of the non-CLIC modes, the only interrupts that can be serviced are the architecturally defined software, timer, and external interrupts.

#### Mode CLINT 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 22.

When operating in CLINT Direct Mode, BASE must be 4-byte aligned.

#### Mode CLINT 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 CLINT vectored interrupt mode, `BASE` must be 128-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.

### Mode CLIC Direct

In CLIC Direct mode, the processor jumps to the 64-byte-aligned trap handler address held in the upper 26 bits of `mtvec` for all exceptions and interrupts.

In CLIC direct interrupt mode, `BASE` must be a minimum of 64-byte aligned.

### Mode CLIC Vectored

In vectored CLIC mode, on an interrupt, the processor switches to the handler's privilege mode and sets the hardware vectoring bit `mcause.MINHV`, then fetches a 32-bit handler address from the in-memory vector table pointed to by `mtvt`, which is described in Section 7.7.6. The address fetched is defined in the following formula:  $mtvt + (mcause.EXCCODE \times 4)$ .

If the fetch is successful, the processor clears the low bit of the handler address, sets the PC to this handler address, then clears `mcause.MINHV`. The hardware vectoring bit `minhv` is provided to allow resumable traps on fetches to the trap vector table.

Synchronous exceptions always trap to `mtvec.BASE` in machine mode.

In CLIC vectored interrupt mode, `BASE` must be 64-byte aligned.

## 7.7.3 Machine Interrupt Enable (`mie`)

Individual interrupts are enabled by setting the appropriate bit in the `mie` register. The `mie` register is described in Table 20.

| Machine Interrupt Enable Register |            |       |                                   |
|-----------------------------------|------------|-------|-----------------------------------|
| CSR                               | mie        |       |                                   |
| Bits                              | Field Name | Attr. | Description                       |
| [2:0]                             | Reserved   | WPRI  |                                   |
| 3                                 | MSIE       | RW    | Machine Software Interrupt Enable |
| [6:4]                             | Reserved   | WPRI  |                                   |
| 7                                 | MTIE       | RW    | Machine Timer Interrupt Enable    |
| [10:8]                            | Reserved   | WPRI  |                                   |
| 11                                | MEIE       | RW    | Machine External Interrupt Enable |
| [31:12]                           | Reserved   | WPRI  |                                   |

Table 20: mie Register

When in either of the CLIC modes, the mie register is hardwired to zero and individual interrupt enables are controlled by the clicIntIE CLIC memory-mapped registers. See Chapter 8 for a detailed description of clicIntIE.

#### 7.7.4 Machine Interrupt Pending (mip)

The machine interrupt pending (mip) register indicates which interrupts are currently pending. The mip register is described in Table 21.

| Machine Interrupt Pending Register |            |       |                                    |
|------------------------------------|------------|-------|------------------------------------|
| CSR                                | mip        |       |                                    |
| Bits                               | Field Name | Attr. | Description                        |
| [2:0]                              | Reserved   | WIRI  |                                    |
| 3                                  | MSIP       | RO    | Machine Software Interrupt Pending |
| [6:4]                              | Reserved   | WIRI  |                                    |
| 7                                  | MTIP       | RO    | Machine Timer Interrupt Pending    |
| [10:8]                             | Reserved   | WIRI  |                                    |
| 11                                 | MEIP       | RO    | Machine External Interrupt Pending |
| [31:12]                            | Reserved   | WIRI  |                                    |

Table 21: mip Register

When in either of the CLIC modes, the mip register is hardwired to zero and individual interrupt enables are controlled by the clicIntIP CLIC memory-mapped registers. See Chapter 8 for a detailed description of clicIntIP.

#### 7.7.5 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_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.

When in either of the CLIC modes, `mcause` is extended to record more information about the interrupted context which is used to reduce the overhead to save and restore that context for an `mret` instruction. CLIC mode `mcause` also adds state to record progress through the trap handling process.

See Table 22 for more details about the `mcause` register. Refer to Table 23 for a list of synchronous exception codes.

| Machine Cause Register |                |       |   |
|------------------------|----------------|-------|---|
| CSR                    | mcause         |       |   |
| Bits                   | Field Name     | Attr. | Description   |
| [9:0]                  | Exception Code | WLRL  | A code identifying the last exception.  |
| [22:10]                | Reserved       | WLRL  |   |
| 23                     | MPIE           | WLRL  | Previous interrupt enable, same as <code>mstatus.mpie</code> . CLIC mode only.        |
| [27:24]                | MPIL           | WLRL  | Previous interrupt level. CLIC mode only.   |
| [29:28]                | MPP            | WLRL  | Previous interrupt privilege mode, same as <code>mstatus.mpp</code> . CLIC mode only. |
| 30                     | MINHV          | WIRL  | Hardware vectoring in progress when set. CLIC mode only.                              |
| 31                     | Interrupt      | WARL  | 1, if the trap was caused by an interrupt; 0 otherwise.                               |

**Table 22:** `mcause` Register

| Interrupt Exception Codes |                |  |
|---------------------------|----------------|--|
| Interrupt                 | Exception Code | Description                            |
| 1                         | 0–2            | Reserved                               |
| 1                         | 3              | Machine software interrupt             |
| 1                         | 4–6            | Reserved                               |
| 1                         | 7              | Machine timer interrupt                |
| 1                         | 8–10           | Reserved                               |
| 1                         | 11             | Machine external interrupt             |
| 1                         | 12             | CLIC Software Interrupt Pending (CSIP) |
| 1                         | 13–15          | Reserved                               |
| 1                         | 16             | CLIC Local Interrupt 0                 |
| 1                         | 17             | CLIC Local Interrupt 1                 |
| 1                         | 18–46          | ...                                    |
| 1                         | 47             | CLIC Local Interrupt 31                |
| 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                         | 8–10           | Reserved                               |
| 0                         | 11             | Environment call from M-mode           |
| 0                         | ≥ 12           | Reserved                               |

Table 23: mcause Exception Codes

### 7.7.6 Machine Trap Vector Table (mtvt)

The mtvt register holds the Machine Trap Vector base address for CLIC vectored interrupts. mtvt allows for relocatable vector tables, where mtvt.BASE must be 64-byte aligned. Values other than 0 in the low 6 bits of mtvt are reserved.

| Machine Trap Vector Table Register |            |       |   |
|------------------------------------|------------|-------|---|
| CSR                                | mtvt       |       |   |
| Bits                               | Field Name | Attr. | Description   |
| [5:0]                              | Reserved   | WARL  |   |
| [31:6]                             | BASE       | WARL  | Base address of the CLIC Vector Table. See Section 8.2. |

Table 24: mtvt Register

### 7.7.7 Handler Address and Interrupt-Enable (mnxti)

The `mnxti` CSR can be used by software to service the next horizontal interrupt when it has greater level than the saved interrupt context (held in `mcause.PIL`), without incurring the full cost of an interrupt pipeline flush and context save/restore. The `mnxti` CSR is designed to be accessed using CSRRSI/CSRRCI instructions, where the value read is a pointer to an entry in the trap handler table and the write back updates the interrupt-enable status. In addition, accesses to the `mnxti` register have side-effects that update the interrupt context state.

Note that this is different than a regular CSR instruction as the value returned is different from the value used in the read-modify-write operation.

A read of the `mnxti` CSR returns either zero, indicating there is no suitable interrupt to service, or the address of the entry in the trap handler table for software trap vectoring.

If the CSR instruction that accesses `mnxti` includes a write, the `mstatus` CSR is the one used for the read-modify-write portion of the operation, while the exception code in `mcause` and the `mintstatus` register's `mil` field can also be updated with the new interrupt level. If the CSR instruction does not include write side effects (e.g., `csrr t0, mnxti`), then no state update on any CSR occurs.

The `mnxti` CSR is intended to be used inside an interrupt handler after an initial interrupt has been taken and `mcause` and `mepc` registers updated with the interrupted context and the id of the interrupt.

### 7.7.8 Machine Interrupt Status (mintstatus)

`mintstatus` holds the active interrupt level for each supported privilege mode. These fields are read-only.

| Machine Interrupt Status Register |            |       |                                     |
|-----------------------------------|------------|-------|-------------------------------------|
| CSR                               | mintstatus |       |                                     |
| Bits                              | Field Name | Attr. | Description                         |
| [23:0]                            | Reserved   | WIRI  |                                     |
| [31:24]                           | MIL        | WIRL  | Active Machine Mode Interrupt Level |

**Table 25:** E20 `mintstatus` Register

### 7.7.9 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. For CLIC vectored mode, configure `mtvt`. The CSR number for `mtvt` is `0x307`.
- Enable interrupts in memory mapped PLIC or CLIC 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.

#### Note

`mie` register is disabled when CLIC modes are used. Use `clicIntIE` to enable interrupts in CLIC modes of operation.

- Write `mstatus` to enable interrupts globally for each supported privilege mode.

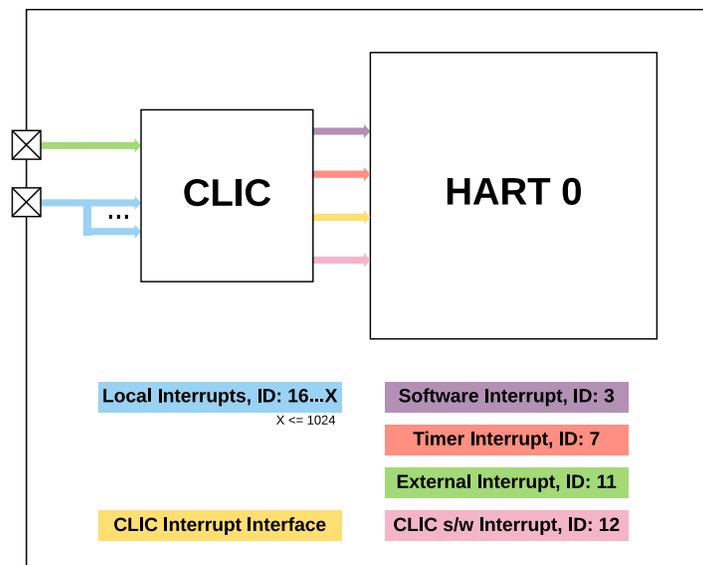
## 7.8 Interrupt Latency

Interrupt latency for the E20 is six clock cycles in CLIC Vectored Mode, as counted by the number of cycles it takes from signaling of the interrupt to the hart to the first instruction of the handler executed. In CLIC Direct Mode, the interrupt latency is four clock cycles.

## Chapter 8

# Core-Local Interrupt Controller (CLIC)

This chapter describes the operation of the Core-Local Interrupt Controller (CLIC). The E20 implements the *RISC-V Core-Local Interrupt Controller (CLIC) specification, Version 20180831*.



**Figure 45:** CLIC Block Diagram

The CLIC is a fully-featured interrupt controller that supports nested interrupts (pre-emption), and programmable interrupt levels and priorities. The CLIC supports software, timer, and external interrupts. In addition to the first 16 local interrupts as defined by the RISC-V Specification, the CLIC also provides 32 additional local external interrupts.

The CLIC provides flexibility for embedded systems with a large number of interrupt sources that require low-latency handling. The CLIC is backwards compatible with the Core-Local Interruptor (CLINT) modes of operation—CLINT direct and CLINT vectored—for software, timer, and external interrupts.

When a CLIC is programmed for CLINT modes of operation, the local external interrupts are not available. The CLIC offers two additional modes of operation, CLIC Direct and CLIC Vectored.

In CLIC direct mode, all interrupts route to the `mtvec.BASE` address, except those that are programmed for vectored mode of operation. These interrupts use the vector table entry with base address `mtvt.BASE`. CLIC vectored mode is a similar concept to CLINT vectored mode, but the CLIC vector table format is slightly different in both the alignment requirements and the actual contents of the vector table itself.

## 8.1 CLIC Interrupt Levels, Priorities, and Preemption

The CLIC allows programmable interrupt levels and priorities for all supported interrupts. The interrupt level is the first step to determine which interrupt gets serviced first, whereas the priority is used to break the tie in the event two interrupts of the same level are received by the hart at the same time.

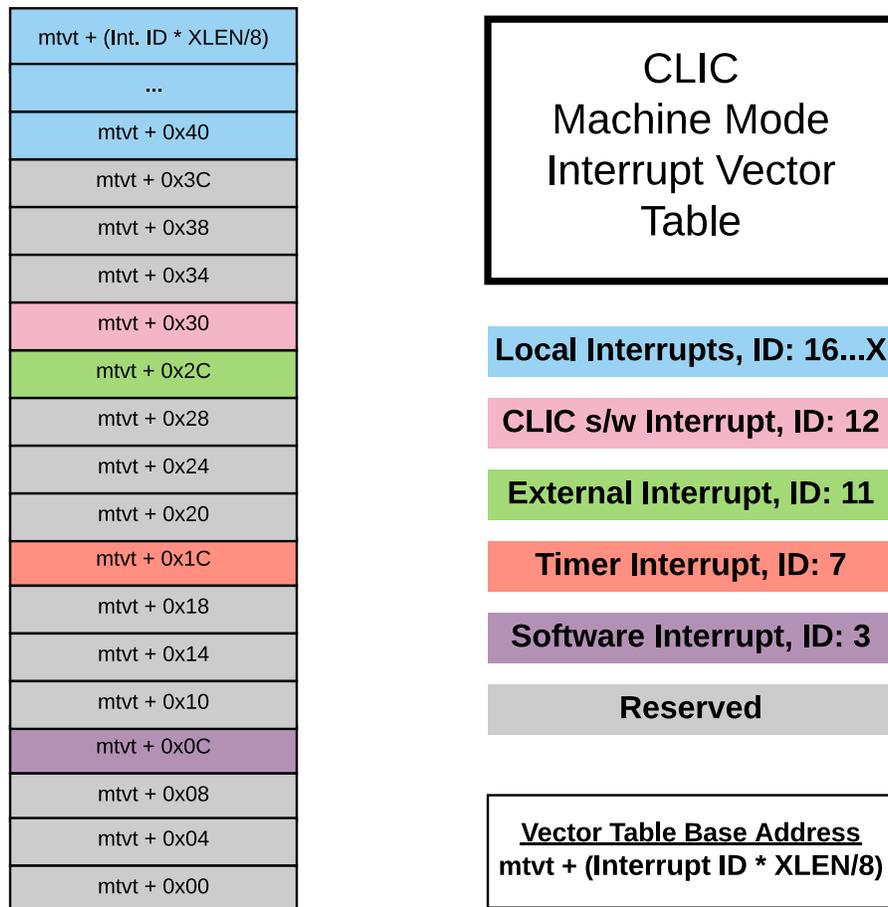
For an interrupt to preempt another active interrupt, the level setting of the non-active interrupt is required to be higher than that of the active interrupt. If two interrupts have the same level setting, preemption will not occur even if one has a higher priority. There are up to 16 level values available.

At any time, a hart is running in some privilege mode with some interrupt level. The hart's current interrupt level is made visible in the `mintstatus` register (Section 7.7.8); however, the current privilege mode is not visible to software running on a hart.

The CLIC supports 49 interrupts, where the first 16 are reserved for software, timer, external, and CLIC software interrupts for all privilege modes, and 32 additional local external interrupts.

The number of preemption levels, and priorities within each level, is determined by the number of configuration bits in the CLIC's `cllicIntCfg` register and the value of the CLIC's `cllicfg.nlBits` register.

## 8.2 CLIC Vector Table



**Figure 46:** CLIC Interrupts and Vector Table

CLIC vectored mode of operation provides the ability to use a vector table for interrupts, shown above. The CLIC vector table is populated with the address of interrupt handlers, not the jump opcode like the CLINT. The software implementation is slightly different for CLIC, since the address of the handler is loaded by hardware directly.

### 8.2.1 CLIC Vector Table Software Example

The example below shows an implementation of the CLIC vector table, in C:

```
#define write_csr(reg, val) ({ \
    asm volatile ("csrw " #reg ", %0" :: "rK"(val)); })

__attribute__((aligned(64))) uintptr_t
__mtvt_clic_vector_table[CLIC_VECTOR_TABLE_SIZE_MAX];

uint32_t mode = MTVEC_MODE_CLIC_VECTORED; /* value of 0x3 */
```

```

/* Setup mtvec to always handle exceptions - same as CLINT vector table */
mtvec_base = (uintptr_t)&__mtvec_clint_vector_table;
write_csr (mtvec, (mtvec_base | mode));

/* Write base address into vector table used for mtvt.BASE for interrupts */
__mtvt_clic_vector_table[INT_ID_SOFTWARE] = (uintptr_t)&software_handler;
__mtvt_clic_vector_table[INT_ID_TIMER] = (uintptr_t)&timer_handler;
__mtvt_clic_vector_table[INT_ID_EXTERNAL] = (uintptr_t)&external_handler;

/* Setup mtvt which is CLIC specific, to hold base address for interrupt
handlers */
mtvt_base = (uintptr_t)&__mtvt_clic_vector_table;
write_csr (0x307, (mtvt_base)); /* 0x307 is CLIC CSR number */

```

### 8.3 CLIC Interrupt Sources

The E20 has 32 interrupt sources that can be connected to peripheral devices, in addition to the standard RISC-V software, timer, and external interrupts. These interrupt inputs are exposed at the top-level via the `local_interrupts` signals. Any unused `local_interrupts` inputs should be tied to logic 0. These signals are positive-level triggered.

The E20 does not include a PLIC, which is used to signal External Interrupts. A Machine External Interrupt signal, `meip`, is exposed at the top-level and can be used to integrate the E20 with an external PLIC.

See the E20 User Manual for a description of these interrupt signals.

CLIC Interrupt IDs are provided in Table 26.

| E20 Interrupt IDs |           |                            |
|-------------------|-----------|----------------------------|
| ID                | Interrupt | Notes                      |
| 0–2               | Reserved  |                            |
| 3                 | msip      | Machine Software Interrupt |
| 4–6               | Reserved  |                            |
| 7                 | mtip      | Machine Timer Interrupt    |
| 8–10              | Reserved  |                            |
| 11                | meip      | Machine External Interrupt |
| 12                | csip      | CLIC Software Interrupt    |
| 13–15             | Reserved  |                            |
| 16                | lint0     | Local Interrupt 0          |
| 17                | lint1     | Local Interrupt 1          |
| ...               | lintX     | Local Interrupt X          |
| 48                | lint31    | Local Interrupt 31         |

**Table 26:** E20 Interrupt IDs

## 8.4 CLIC 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
}
```

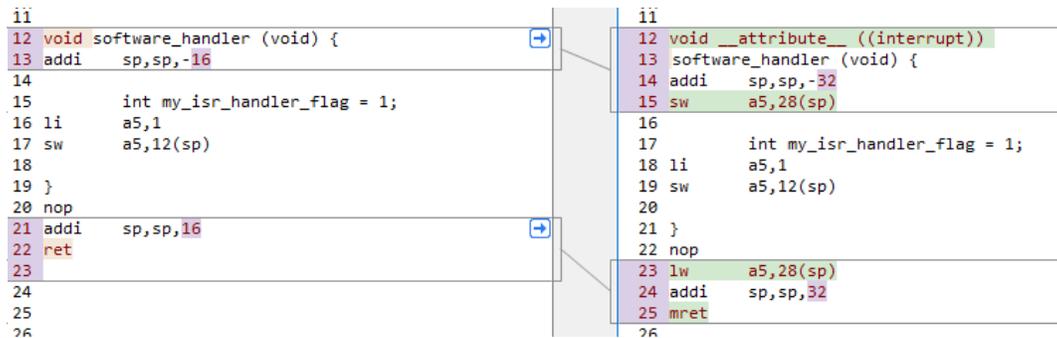


Figure 47: CLIC Interrupt Attribute Example

This attribute will save and restore registers that are used within the handler, and insert an `mret` instruction at the end of the handler.

### 8.4.1 CLIC Preemption Interrupt Attribute

In order for an interrupt of a higher level to preempt an active interrupt of a lower level, `mstatus.mie` needs to be enabled (non-zero) within the handler, since it is disabled by hardware automatically upon entry. Prior to re-enabling interrupts through `mstatus.mie`, `mepc` and `mcause` must first be saved and subsequently restored before `mret` is executed at the end of the handler. There is a CLIC-specific interrupt attribute that will do these steps automatically.

```
void __attribute__((interrupt("SiFive-CLIC-preemptible")))
software_handler (void) {
    // handler code
}
```

#### Note

Using the `SiFive-CLIC-preemptible` attribute requires the addition of the `-fomit-frame-pointer` compiler flag.

The functionality of this CLIC-specific attribute can be demonstrated by comparing the list output of functions with and without the attribute applied.

```

8
9
10
11
12 void software_handler (void) {
13     addi    sp,sp,-16
14
15     int my_isr_handler_flag = 1;
16     li     a5,1
17     sw     a5,12(sp)
18
19 }
20 nop
21 addi    sp,sp,16
22 ret
23
24
25
26
27
28
29
30
31
12 void __attribute__((interrupt("SiFive-CLIC-preemptible")))
13 software_handler (void) {
14     addi    sp,sp,-32
15     sw     s0,28(sp)
16     sw     s1,24(sp)
17     csrr  s0,mcause
18     csrr  s1,mepc
19     csrsi mstatus,8
20     sw     a5,20(sp)
21
22     int my_isr_handler_flag = 1;
23     li     a5,1
24     sw     a5,12(sp)
25
26 }
27 nop
28 lw     a5,20(sp)
29 csrci  mstatus,8
30 csrw   mepc,s1
31 csrw   mcause,s0
32 lw     s1,24(sp)
33 lw     s0,28(sp)
34 addi    sp,sp,32
35 mret

```

**Figure 48:** CLIC Preemption Interrupt Attribute Example

Note that this attribute applies to vectored interrupts. To support preemption for non-vectored interrupts, refer to the CLIC Specification example, here:

<https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc#c-abi-trampoline-code>

Also, refer to the CLIC section on how to manage interrupt stacks across privilege modes, here: <https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc#managing-interrupt-stacks-across-privilege-modes>

## 8.5 Details for CLIC Modes of Operation

In CLIC modes of operation, both the Machine Interrupt Enable (mie) and Machine Interrupt Pending (mip) registers are hard wired to zero, and their functionality moves to the clicIntIE and clicIntIP registers.

## 8.6 Memory Map

The CLIC memory map is separated into multiple regions depending on the number of harts that implement a CLIC; one shared region, and as many hart-specific regions. This allows for backwards compatibility with the Core-Local Interruptor (CLINT) and its msip, mtimecmp, and mtime memory-mapped registers, as well as compatibility between CLIC and non-CLIC harts. The base address for all regions are provided below in Table 27.

| Base Addresses for CLIC Regions |        |  |
|---------------------------------|--------|--|
| Address                         | Region | Notes  |
| 0x0200_0000                     | Shared | RISC-V Standard CLINT Base. The specific implementation of this region is described in detail in Table 28. |
| 0x0280_0000                     | Hart 0 | Hart 0 CLIC Base. The specific implementation of this region is described in detail in Table 29.           |

Table 27: CLIC Base Addresses

| CLIC Shared Region |       |       |                     |                            |
|--------------------|-------|-------|---------------------|----------------------------|
| Offset             | Width | Attr. | Description         | Notes                      |
| 0x0000             | 4B    | RW    | msip for hart 0     | MSIP Register (1 bit wide) |
| 0x0004             |       |       | Reserved            |                            |
| ...                |       |       |                     |                            |
| 0x3FFF             |       |       |                     |                            |
| 0x4000             | 8B    | RW    | mtimecmp for hart 0 | MTIMEECMP Register         |
| 0x4008             |       |       | Reserved            |                            |
| ...                |       |       |                     |                            |
| 0xBFF7             |       |       |                     |                            |
| 0xBFF8             | 8B    | RW    | mtime               | Timer Register             |
| 0xC000             |       |       | Reserved            |                            |

Table 28: CLIC Shared Register Map

| CLIC Hart-Specific Region |                     |            |  |
|---------------------------|---------------------|------------|--|
| Offset                    | Width               | Name       | Description                            |
| 0x000                     | 1B per Interrupt ID | clicIntIP  | CLIC Interrupt Pending Registers       |
| 0x400                     | 1B per Interrupt ID | clicIntIE  | CLIC Interrupt Enable Registers        |
| 0x800                     | 1B per Interrupt ID | clicIntCfg | CLIC Interrupt Configuration Registers |
| 0xC00                     | 1B                  | clicCfg    | CLIC Configuration Register            |

Table 29: CLIC Hart-Specific Region Map

## 8.7 Register Descriptions

This section describes the changes made to interrupt CSRs while in CLIC mode, as well as additional CLIC mode registers.

### 8.7.1 Changes to CSRs in CLIC Mode

This section describes the differences to CSRs when a CLIC is implemented, compared to a design including a CLINT. See Section 7.7 for further description of these CSRs.

**mstatus**

mstatus.mpp and mstatus.mpie are accessible via fields in the mcause register.

**mie and mip**

mie and mip are hardwired to zero and replaced with memory-mapped clicIntIE and clicIntIP registers.

**mtvec**

Additional modes that enable CLIC modes of operation.

**mcause**

Stores previous privilege mode and previous interrupt enable.

**8.7.2 CLIC Interrupt Pending Register (clicIntIP)**

| clicIntIP Register |            |                                   |      |  |
|--------------------|------------|-----------------------------------|------|--|
| Register Address   |            | CLIC Hart Base + 1 × Interrupt ID |      |  |
| Bits               | Field Name | Attr.                             | Rst. | Description  |
| 0                  | clicIntIP  | RO*                               | 0    | When clicIntIP is set, the corresponding Interrupt ID is pending. Only the software interrupt bits are writable. For all other interrupts these are read-only registers connected directly to input pins or logic. |
| [7:1]              | Reserved   | RO                                | 0    |  |

\*clicIntIP bits for msip (Interrupt ID #3) and csip (Interrupt ID #12) are RW registers that enables software to set these interrupts to pending.

**Table 30:** CLIC Interrupt Pending Register (partial)

When in CLIC mode, the Machine Interrupt Pending (mip) CSR is hardwired to zero and interrupt pending status is instead presented in the clicIntIP memory-mapped registers.

**8.7.3 CLIC Interrupt Enable Register (clicIntIE)**

| clicIntIE Register |            |   |      |   |
|--------------------|------------|---|------|---|
| Register Address   |            | CLIC Hart Base + 0x400 + 1 × Interrupt ID |      |   |
| Bits               | Field Name | Attr.                                     | Rst. | Description   |
| 0                  | clicIntIE  | RW  | 0    | When clicIntIE is set, the corresponding Interrupt ID is enabled. |
| [7:1]              | Reserved   | RO  | 0    |   |

**Table 31:** CLIC Interrupt Enable Register (partial)

When in CLIC mode, the Machine Interrupt Enable (mie) CSR is hardwired to zero and interrupt enables are instead presented in the clicIntIE memory-mapped registers.

#### 8.7.4 CLIC Interrupt Configuration Register (clicIntCfg)

| clicIntCfg Register |               |   |      |   |
|---------------------|---------------|---|------|---|
| Register Address    |               | CLIC Hart Base + 0x800 + 1 × Interrupt ID |      |   |
| Bits                | Field Name    | Attr.                                     | Rst. | Description   |
| [5:0]               | clicIntCfgPad | RO  | 0    | Padding of clicIntCfg.  |
| [7:6]               | clicIntCfg    | RW  | 0    | clicIntCfg sets the pre-emption level and priority of a given interrupt. When selective hardware vectoring is enabled, the least-significant bit is used to control vectoring of a given interrupt. |

**Table 32:** CLIC Interrupt Configuration Register (partial)

The E20 has a total of 2 bits in clicIntCfg which specify how to encode a given interrupt's pre-emption level and/or priority. The actual number of bits which determine the pre-emption level is determined by clicCfg.NLBITS. If clicCfg.NLBITS is < 2, then the remaining least significant implemented bits are used to encode priorities within a given pre-emption level. If clicCfg.NLBITS is set to zero, then all interrupts are treated as level 255 and all 2 bits are used to set priorities.

#### 8.7.5 CLIC Configuration Register (clicCfg)

This register determines the number of levels and priorities set by clicIntCfg. It also contains the selective hardware vector configuration, which allows CLIC direct mode or vectored mode on a per-interrupt basis.

| clicCfg Register |            |                        |      |   |
|------------------|------------|------------------------|------|---|
| Register Address |            | CLIC Hart Base + 0xC00 |      |   |
| Bits             | Field Name | Attr.                  | Rst. | Description   |
| 0                | nvBits     | RW                     | 0    | When set, selective hardware vectoring is enabled.          |
| [4:1]            | nlBits     | RW                     | 0    | Determines the number of Level bits available in clicIntCfg |
| [6:5]            | nmBits     | RW                     | 0    | Determines the number of Mode bits available in clicIntCfg. |
| 7                | Reserved   | WARL                   | 0    |   |

**Table 33:** CLIC Configuration Register (partial)

The `cliccfg` register is used to configure the operation of the CLIC primarily by determining the function of the bits implemented in `clicIntCfg` bits. The E20 only supports machine mode interrupts, therefore `cliccfg.nmBits` is set to zero.

`cliccfg.nlBits` is used to determine the number of `clicIntCfg` bits used for levels versus priorities. The CLIC supports a maximum of 256 pre-emption levels, which requires 8 bits to encode all 256 levels. For values of `cliccfg.nlBits` less than 8, the lower bits are assumed to be all 1s. The resulting encoding of `cliccfg.nlBits` to interrupt levels is shown below:

| Value  | Encoding | Interrupt Levels  |
|--|----------|---|
| 0  | 1111     | 255   |
| 1  | x111     | 127,255   |
| 2  | xx11     | 63,127,191,255  |
| 3  | xxx1     | 31,63,65,127,159,191,223,255                              |
| 4  | xxxx     | 15,31,47,63,79,95,111,127,143,159,175,191,207,223,239,255 |
| Note: x bits are available <code>clicIntCfg</code> bits. |          |   |

**Table 34:** Encoding of `cliccfg.nlBits`

See Section 8.7.4 for a description of the effects of `cliccfg.nlBits` on `clicIntCfg`.

`cliccfg.nvBits` allows for certain, selected, interrupts to be vectored while in CLIC Direct mode. If in CLIC Direct mode and `cliccfg.nvBits` is set to 1, then selective interrupt vectoring is turned on. The least-significant implemented bit of `clicIntCfg` (bit 6 in the E20) controls the vectoring behavior of a given interrupt. When in CLIC Direct mode, and both `cliccfg.nvBits` and the relevant bit of `clicIntCfg` are set to 1, then the interrupt is vectored using the vector table pointed to by the `mtvt` CSR. This allows some interrupts to all jump to a common base address held in `mtvec`, while the others are vectored in hardware.

## Chapter 9

# Power Management

The following chapter establishes flows for powering up, powering down, and resetting the hardware of the E20.

### 9.1 Hardware Reset

The following list summarizes the hardware reset values required by the RISC-V Privileged Specification 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 `misalr` register holds the full set of supported extensions for that implementation, and `misalr.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 a value indicating the cause of the 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.

### 9.2 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.

### 9.3 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 CLIC exists, ensure memory-mapped CLIC interrupt enable register `clicIntIE` contents reflect the requirements of the system, and that no unexpected CLIC pending `clicIntIP` bits are set.

The `clicIntIP` bits are read-only with the exception of the software interrupt (`clicIntIP[0]`, bit 3) and the CLIC software interrupt pending (`clicIntIP[0]`, bit 12). If any of the non-software CLIC pending bits are set, check the source of the interrupt.

Note that `mip` and `mie` are hardwired to 0 when using CLIC modes of operation, and all enable and pending status reside in memory mapped `clicIntIE` and `clicIntIP` registers.

## 9.4 Other Boot Time Considerations

- 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.
- 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.

## 9.5 Power-Down Flow

Designate one core as master and all others as slaves. For our Core IP product, coordination with an External Agent is required.

1. External Agent: Wait for communication from master 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_X` is high for the master core and all slave cores.
  - d. Once `cease_from_tile_X` is high for master core and all slave cores, apply reset to the whole core complex.

2. Master 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. Master core: if there is an L2 cache, flush it (all addresses at which cacheable physical memory exists).
  - iii. 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
- e. Disable all interrupts.
- f. Execute CEASE instruction.

# Chapter 10

## Debug

This chapter describes the operation of SiFive debug hardware, which follows *The RISC-V Debug Specification, Version 0.13*. Currently only interactive debug and hardware breakpoints are supported.

### 10.1 Debug CSRs

This section describes the per hart Trace and Debug Registers (TDRs), which are mapped into the CSR space as follows:

| CSR Name | Description                       | Allowed Access Modes |
|----------|-----------------------------------|----------------------|
| tselect  | Trace and debug register select   | Debug, Machine       |
| tdata1   | First field of selected TDR       | Debug, Machine       |
| tdata2   | Second field of selected TDR      | Debug, Machine       |
| tdata3   | Third field of selected TDR       | Debug, Machine       |
| dcsr     | Debug control and status register | Debug                |
| dpc      | Debug PC                          | Debug                |
| dscratch | Debug scratch register            | Debug                |

**Table 35:** Debug Control and Status Registers

The dcsr, dpc, and dscratch registers are only accessible in debug mode, while the tselect and tdata1-3 registers are accessible from either debug mode or machine mode.

#### 10.1.1 Trace and Debug Register Select (tselect)

To support a large and variable number of TDRs 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:

| Trace and Debug Select Register |            |       |  |
|---------------------------------|------------|-------|--|
| CSR                             | tselect    |       |  |
| Bits                            | Field Name | Attr. | Description                                  |
| [31:0]                          | index      | WARL  | Selection index of trace and debug registers |

Table 36: tselect CSR

The index field is a **WARL** field that does not hold indices of unimplemented TDRs. Even if index can hold a TDR index, it does not guarantee the TDR exists. The type field of tdata1 must be inspected to determine whether the TDR exists.

### 10.1.2 Trace and Debug Data Registers (tdata1-3)

The tdata1-3 registers are 32-bit read/write registers selected from a larger underlying bank of TDR registers by the tselect register.

| Trace and Debug Data Register 1 |                   |       |  |
|---------------------------------|-------------------|-------|--|
| CSR                             | tdata1            |       |  |
| Bits                            | Field Name        | Attr. | Description  |
| [27:0]                          | TDR-Specific Data |       |  |
| [31:28]                         | type              | RO    | Type of the trace & debug register selected by tselect |

Table 37: tdata1 CSR

| Trace and Debug Data Registers 2 and 3 |                   |       |             |
|--|-------------------|-------|-------------|
| CSR                                    | tdata2/3          |       |             |
| Bits                                   | Field Name        | Attr. | Description |
| [31:0]                                 | TDR-Specific Data |       |             |

Table 38: tdata2/3 CSRs

The high nibble of tdata1 contains a 4-bit type code that is used to identify the type of TDR selected by tselect. The currently defined types are shown below:

| Type | Description                |
|------|----------------------------|
| 0    | No such TDR register       |
| 1    | Reserved                   |
| 2    | Address/Data Match Trigger |
| ≥3   | Reserved                   |

Table 39: tdata Types

The dmode bit 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 TDRs. Any

attempt to read/write the `tdata1-3` registers in machine mode when `dmode=1` raises an illegal instruction exception.

### 10.1.3 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 0.13*.

### 10.1.4 Debug PC (`dpc`)

When entering debug mode, the current PC is copied here. When leaving debug mode, execution resumes at this PC.

### 10.1.5 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 0.13*.

## 10.2 Breakpoints

The E20 supports four 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                |
|----------------------|-----------------------|----------------------------|
| <code>tselect</code> | <code>tselect</code>  | Breakpoint selection index |
| <code>tdata1</code>  | <code>mcontrol</code> | Breakpoint match control   |
| <code>tdata2</code>  | <code>maddress</code> | Breakpoint match address   |
| <code>tdata3</code>  | N/A                   | Reserved                   |

**Table 40:** TDR CSRs when used as Breakpoints

### 10.2.1 Breakpoint Match Control Register (`mcontrol`)

Each breakpoint control register is a read/write register laid out in Table 41.

| Breakpoint Control Register |            |       |      |   |
|-----------------------------|------------|-------|------|---|
| CSR                         | mcontrol   |       |      |   |
| Bits                        | Field Name | Attr. | Rst. | Description                                 |
| 0                           | R          | WARL  | X    | Address match on LOAD                       |
| 1                           | W          | WARL  | X    | Address match on STORE                      |
| 2                           | X          | WARL  | X    | 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    | Reserved                                    |
| 6                           | M          | WARL  | X    | Address match on machine mode               |
| [10:7]                      | match      | WARL  | X    | Breakpoint Address Match type               |
| 11                          | chain      | WARL  | 0    | Chain adjacent conditions.                  |
| [15:12]                     | action     | WARL  | 0    | Breakpoint action to take.                  |
| [17:16]                     | sizeLo     | WARL  | 0    | Size of the breakpoint. Always 0.           |
| 18                          | timing     | WARL  | 0    | Timing of the breakpoint. Always 0.         |
| 19                          | select     | WARL  | 0    | Perform match on address or data. Always 0. |
| 20                          | Reserved   | WPRI  | X    | Reserved                                    |
| [26:21]                     | maskmax    | RO    | 4    | Largest supported NAPOT range               |
| 27                          | dmode      | RW    | 0    | Debug-Only access mode                      |
| [31:28]                     | type       | RO    | 2    | Address/Data match type, always 2           |

Table 41: Test and Debug Data Register 3

The type field is a 4-bit read-only field holding the value 2 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:

| <b>address</b> | <b>Match type and size</b> |
|----------------|----------------------------|
| a...aaaaaa     | Exact 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 42:** NAPOT Size Encoding

The `maskmax` field is a 6-bit read-only field that specifies the largest supported NAPOT range. The value 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 `address` 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.

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.

### 10.2.2 Breakpoint Match Address Register (`address`)

Each breakpoint match address register is a 32-bit read/write register used to hold significant address bits for address matching and also the unary-encoded address masking information for NAPOT ranges.

### 10.2.3 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.

### 10.2.4 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.

## 10.3 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).

### 10.3.1 Debug RAM and Program Buffer (0x300–0x3FF)

The E20 has two 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 E20 has one 32-bit words of debug data RAM. Its location can be determined by reading the `DMHARTINFO` register as described in the RISC-V Debug Specification. This RAM space is used to pass data for the Access Register abstract command described in the RISC-V Debug Specification. The E20 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 E20, 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.

The E20 does not implement the `DMSTATUS.anyhavereset` or `DMSTATUS.allhavereset` bits.

### 10.3.2 Debug ROM (0x800–0xFFF)

This ROM region holds the debug routines on SiFive systems. The actual total size may vary between implementations.

### 10.3.3 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.

### 10.3.4 Safe Zero Address

In the E20, the debug module contains the addresses 0x0 through 0xFFF 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.

## 10.4 Debug Module Interface

The SiFive Debug Module (DM) conforms to *The RISC-V Debug Specification, Version 0.13*. 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.

### 10.4.1 DM Registers

#### dmstatus register

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).

#### dmcontrol register

A debugger performs most hart control through the dmcontrol register.

| Control      | Function   |
|--------------|--|
| dmactive     | This bit enables the DM and is reflected in the dmactive output signal. When dmactive=0, the clock to the DM is gated off. |
| ndmreset     | This is a read/write bit that drives the ndreset output signal.  |
| resethaltreq | When set, the DM will halt the hart when it emerges from reset.  |
| hartreset    | Not Supported  |
| hartsel      | This field selects the hart to operate on  |
| hase1        | Not Supported  |

**Table 43:** Debug Control Register

### 10.4.2 Abstract Commands

Abstract commands provide a debugger with a path to read and write processor state. Many aspects of Abstract Commands are optional in the RISC-V Debug Spec and are implemented as described below.

| <b>cmdtype</b>  | <b>Feature</b>       | <b>Support</b>   |
|-----------------|----------------------|--|
| 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 44:** Debug Abstract Commands

# Chapter 11

## Appendix

### 11.1 Appendix A

This section lists the key configuration options of the SiFive E2 Series core. The configuration for the E20 is listed in `docs/core_complex_configuration.txt`.

#### 11.1.1 E2 Series

The E2 Series comes with the following set of configuration options:

##### Modes and ISA

- Optional support for RISC-V user mode
- Optional M, A, and F extensions
- Configurable Multiplication performance (1-cycle or 4-cycle)
- Shared or Separate Core Instruction and Data Interface(s)
- Configurable base ISA (RV32I or RV32E)
- Optional SiFive Custom Instruction Extension (SCIE)

##### On-Chip Memory

- 1 or 2 optional Tightly-Integrated Memories (TIMs), configurable up to 512 KiB
- Optional  $\mu$ Instruction Cache, configurable up to 16 KiB

##### Ports

- Optional second System Port, Peripheral Port, and Front Port
  - Each port has a configurable base address, size, and protocol (AHB, AHB-Lite, APB, AXI4)

##### Security

- Number of Physical Memory Protection registers (2 to 16)

## **Debug**

- Configurable debug interface (JTAG, cJTAG, APB)
- Number of Hardware Breakpoints (0 to 16) and External Triggers (0 to 16)
- Optional System Bus Access
- Optional Core Register Access
- Configurable number of performance counters (0 to 8)
- Optional Raw Instruction Trace Port
- Optional Nexus Trace Encoder with the following options:
  - Trace Sink (SRAM, ATB Bridge, SWT)
  - Optional Timestamp capabilities with configurable width and source
  - External Trigger Inputs (0 to 8) and Outputs (0 to 8)
  - Trace Buffer size (256 KiB to 64 KiB)
  - Optional Instrumented Trace

## **Interrupts**

- Optional Core-Local Interrupt Controller (CLIC) with the following parameters:
  - Priority Bits (2 to 8)
  - Number of interrupts (1 to 511)
- If no CLIC, then a configurable number of Core-Local Interruptor (CLINT) interrupts (0 to 16)

## **Design For Test**

- Optional SRAM Macro Extraction
- Optional Clock Gate Extraction
- Optional Grouping and Wrapping of extracted macros

## **Power Management**

- Optional Clock Gating
- Separate Reset for Core and Uncore

## **Clock and Reset**

- Configurable Reset Scheme (Synchronous, Asynchronous, Full Asynchronous)
- Optional Separate GPR Reset

Note that the configuration may be limited to a fixed set of discrete options.

## Chapter 12

# 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] K. Asanovic, Eds., SiFive Proposal for a RISC-V Core-Local Interrupt Controller (CLIC). [Online]. Available: <https://github.com/sifive/clic-spec>