



SiFive E20 Core Complex Manual

21G2.01.00

Copyright © 2021 by SiFive, Inc. All rights reserved.

SiFive E20 Core Complex Manual

Proprietary Notice

Copyright © 2021 by SiFive, Inc. All rights reserved.

SiFive E20 Core Complex Manual by SiFive, Inc. is licensed under Attribution-NonCommercialNoDerivatives 4.0 International. To view a copy of this license, visit: <http://creativecommons.org/licenses/by-nc-nd/4.0>

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 rising 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.

Contents

List of Tables	7
List of Figures	11
1 Introduction	14
1.1 About this Document	14
1.2 About this Release	14
1.3 E20 Core Complex Overview	15
1.4 E2 RISC-V Core	16
1.5 Interrupts	16
1.6 Debug Support	16
1.7 Compliance	16
2 List of Abbreviations and Terms	18
3 E2 RISC-V Core	20
3.1 Supported Modes	20
3.2 Instruction Memory System.....	20
3.2.1 Execution Memory Space	20
3.2.2 Instruction Fetch Unit.....	21
3.3 Execution Pipeline	21
3.4 Data Memory System.....	22
3.5 Local Interrupts.....	22
3.6 Hardware Performance Monitor.....	23
3.6.1 Performance Monitoring Counters Reset Behavior	23
3.6.2 Fixed-Function Performance Monitoring Counters	23
3.7 Ports.....	23
3.7.1 System Port.....	23

4	Physical Memory Attributes and Memory Map	25
4.1	Physical Memory Attributes Overview	25
4.2	Memory Map	26
5	Programmer's Model	27
5.1	Base Instruction Formats	27
5.2	I Extension: Standard Integer Instructions	28
5.2.1	R-Type (Register-Based) Integer Instructions	29
5.2.2	I-Type Integer Instructions	30
5.2.3	I-Type Load Instructions	31
5.2.4	S-Type Store Instructions	32
5.2.5	Unconditional Jumps	33
5.2.6	Conditional Branches	34
5.2.7	Upper-Immediate Instructions	35
5.2.8	Memory Ordering Operations	35
5.2.9	Environment Call and Breakpoints	36
5.2.10	NOP Instruction	36
5.3	M Extension: Multiplication Operations	36
5.3.1	Division Operations	37
5.4	C Extension: Compressed Instructions	37
5.4.1	Compressed 16-bit Instruction Formats	37
5.4.2	Stack-Pointed-Based Loads and Stores	38
5.4.3	Register-Based Loads and Stores	39
5.4.4	Control Transfer Instructions	40
5.4.5	Integer Computational Instructions	41
5.5	B Extension: Bit Manipulation Instructions	44
5.5.1	Zba Extension	44
5.5.2	Zbb Extension	44
5.6	Zicsr Extension: Control and Status Register Instructions	47
5.6.1	Control and Status Registers	49
5.6.2	Defined CSRs	49
5.6.3	CSR Access Ordering	51
5.6.4	SiFive RISC-V Implementation Version Registers	51

5.6.5	Custom CSRs	53
5.7	Base Counters and Timers	53
5.7.1	Timer Register	54
5.7.2	Timer API	54
5.8	Privileged Instructions	55
5.8.1	Machine-Mode Privileged Instructions	55
5.9	ABI - Register File Usage and Calling Conventions	56
5.9.1	RISC-V Assembly.....	58
5.9.2	Assembler to Machine Code.....	58
5.9.3	Calling a Function (Calling Convention)	60
5.10	Memory Ordering - FENCE Instructions	63
5.11	Boot Flow	64
5.12	Linker File	65
5.12.1	Linker File Symbols	66
5.13	RISC-V Compiler Flags	67
5.13.1	arch, abi, and mtune.....	67
5.14	Compilation Process	71
5.15	Large Code Model Workarounds	71
5.15.1	RISC-V Code Model Summary	72
5.15.2	Enabling the Compact Code Model	72
5.16	Pipeline Hazards.....	73
5.16.1	Read-After-Write Hazards	73
5.16.2	Write-After-Write Hazards.....	74
5.17	Reading CSRs.....	74
6	Custom Instructions and CSRs.....	76
6.1	CEASE	76
6.2	Other Custom Instructions	76
7	Interrupts and Exceptions.....	77
7.1	Interrupt Concepts	77
7.2	Exception Concepts	78
7.3	Trap Concepts.....	79

7.4	Interrupt Block Diagram	80
7.5	Local Interrupts	81
7.6	Interrupt Operation	81
7.6.1	Interrupt Entry and Exit	81
7.6.2	Critical Sections in Interrupt Handlers	82
7.7	Interrupt Control and Status Registers	82
7.7.1	Machine Status Register (mstatus)	82
7.7.2	Machine Trap Vector (mtvec)	83
7.7.3	Machine Interrupt Enable (mie)	85
7.7.4	Machine Interrupt Pending (mip)	85
7.7.5	Machine Cause (mcause)	86
7.7.6	Machine Trap Vector Table (mtvt)	88
7.7.7	Handler Address and Interrupt-Enable (mnxti)	89
7.7.8	Machine Interrupt Status (mintstatus)	89
7.7.9	Minimum Interrupt Configuration	90
7.8	Interrupt Latency	90
7.9	Non-Maskable Interrupt	90
7.9.1	Handler Addresses	90
7.9.2	RNMI CSRs	90
7.9.3	MNRET Instruction	91
7.9.4	RNMI Operation	92
8	Core-Local Interrupt Controller (CLIC)	93
8.1	CLIC Interrupt Levels, Priorities, and Preemption	94
8.2	CLIC Vector Table	95
8.2.1	CLIC Vector Table Software Example	95
8.3	CLIC Interrupt Sources	96
8.4	CLIC Interrupt Attribute	96
8.4.1	CLIC Preemption Interrupt Attribute	97
8.5	Details for CLIC Modes of Operation	98
8.6	Memory Map	98
8.7	Register Descriptions	99
8.7.1	Changes to CSRs in CLIC Mode	99

8.7.2	MSIP Register.....	100
8.7.3	Timer Registers.....	100
8.7.4	CLIC Clock Gate Disable Register (disableClicClockGateFeature)	100
8.7.5	CLIC Interrupt Pending Register (clicIntIP)	101
8.7.6	CLIC Interrupt Enable Register (clicIntIE).....	101
8.7.7	CLIC Interrupt Configuration Register (clicIntCfg).....	101
8.7.8	CLIC Configuration Register (cliccfg)	102
9	Power Management	104
9.1	Power Modes	104
9.2	Run Mode	104
9.2.1	Power Control	104
9.3	WFI Clock Gate Mode	105
9.3.1	WFI Wake Up	105
9.4	CEASE Instruction for Power Down	106
9.5	Hardware Reset.....	106
9.6	Early Boot Flow	106
9.7	Interrupt State During Early Boot.....	107
9.8	Other Boot Time Considerations.....	108
9.9	Power-Down Flow.....	108
10	Debug	110
10.1	Debug Module	110
10.2	Debug and Trigger Registers.....	113
10.2.1	Debug Control and Status Register (dcsr)	113
10.2.2	Debug PC (dpc)	114
10.2.3	Debug Scratch (dscratch)	114
10.2.4	Trigger Select Register (tselect)	115
10.2.5	Trigger Data Registers (tdata1-3).....	115
10.3	Breakpoints	116
10.3.1	Breakpoint Match Control Register (mcontrol).....	116
10.3.2	Breakpoint Match Address Register (maddress)	119
10.3.3	Breakpoint Execution	119

10.3.4	Sharing Breakpoints Between Debug and Machine Mode	119
10.4	Debug Memory Map.....	119
10.4.1	Debug RAM and Program Buffer (0x300–0x3FF)	119
10.4.2	Debug ROM (0x800–0xFFF)	120
10.4.3	Debug Flags (0x100–0x110, 0x400–0x7FF)	120
10.4.4	Safe Address	120
10.5	Debug Module Interface.....	120
10.5.1	Debug Module Status Register (dmstatus)	121
10.5.2	Debug Module Control Register (dmcontrol)	122
10.5.3	Hart Info Register (hartinfo)	123
10.5.4	Abstract Control and Status Register (abstractcs)	125
10.5.5	Abstract Command Register (command)	126
10.5.6	Abstract Command Autoexec Register (abstractauto)	126
10.5.7	Debug Module Control and Status 2 Register (dmcs2).....	126
10.5.8	Abstract Commands	127
10.6	Debug Module Operational Sequences	129
10.6.1	Entering Debug Mode	129
10.6.2	Exiting Debug Mode	129
A	SiFive Core Complex Configuration Options	130
A.1	E2 Series.....	130
B	SiFive RISC-V Implementation Registers	134
B.1	Machine Architecture ID Register (machid)	134
B.2	Machine Implementation ID Register (mimpid)	135
C	Revision History	136
	References	137

Tables

Table 1	E20 Core Complex Feature Set	14
Table 2	RISC-V Specification Compliance	17
Table 3	Abbreviations and Terms.....	19
Table 4	E2 Feature Set.....	20
Table 5	Executable Memory Regions for the E20 Core Complex	21
Table 6	E2 Instruction Latency	22
Table 7	Physical Memory Attributes for External Regions	25
Table 8	Physical Memory Attributes for Internal Regions.....	26
Table 9	E20 Core Complex Memory Map. Physical Memory Attributes: R –Read, W –Write, X –Execute, I –Instruction Cacheable, D –Data Cacheable, A –Atomics.....	26
Table 10	Base Instruction Formats	27
Table 11	R-Type Integer Instructions.....	29
Table 12	R-Type Integer Instruction Description	29
Table 13	I-Type Integer Instructions	30
Table 14	I-Type Integer Instruction Description	31
Table 15	I-Type Load Instructions	32
Table 16	I-Type Load Instruction Description	32
Table 17	S-Type Store Instructions	33
Table 18	S-Type Store Instruction Description	33
Table 19	J-Type Instruction Description.....	34
Table 20	B-Type Instructions.....	34
Table 21	B-Type Instruction Description	34
Table 22	RISC-V Base Instruction to Assembly Pseudoinstruction Example	35
Table 23	Multiplication Operation Description	36
Table 24	Division Operation Description	37
Table 25	Stack-Pointed-Based Load Instruction Description.....	38
Table 26	Stack-Pointed-Based Store Instruction Description	39
Table 27	Register-Based Load Instruction Description.....	39
Table 28	Register-Based Store Instruction Description	40

Table 29	Unconditional Jump Instruction Description	40
Table 30	Unconditional Control Transfer Instruction Description	41
Table 31	Conditional Control Transfer Instruction Description	41
Table 32	Integer Constant-Generation Instruction Description	41
Table 33	Integer Register-Immediate Operation Description	42
Table 34	Integer Register-Immediate Operation Description (cont.)	42
Table 35	Integer Register-Immediate Operation Description (cont.)	42
Table 36	Integer Register-Immediate Operation Description (cont.)	42
Table 37	Integer Register-Immediate Operation Description (cont.)	43
Table 38	Integer Register-Register Operation Description	43
Table 39	Integer Register-Register Operation Description (cont.)	43
Table 40	Address Calculation Instructions Description	44
Table 41	Count Leading/Trailing Zeroes Instructions Description	45
Table 42	Count Population Instructions Description	45
Table 43	Logic-With-Negate Instructions Description	45
Table 44	Comparison Instructions Description	46
Table 45	Sign- and Zero-Extend Instructions	46
Table 46	Bitwise Rotation Instructions Description	47
Table 47	OR Combine Instruction Description	47
Table 48	Byte-Reverse Instruction Description	47
Table 49	Control and Status Register Instruction Description	48
Table 50	CSR Reads and Writes	49
Table 51	Machine Mode CSRs	50
Table 52	Debug Mode Registers	50
Table 53	Core Generator Encoding of marchid	52
Table 54	Generator Release Encoding of mimpid	52
Table 55	Timer and Counter Pseudoinstruction Description	53
Table 56	Timer and Counter CSRs	54
Table 57	RISC-V Registers	57
Table 58	RISC-V Assembly and C Examples	58
Table 59	RISC-V Code Model Table	72
Table 60	Exception Priority	78
Table 61	Summary of Exception and Interrupt CSRs	79

Table 62	Machine Status Register (partial)	83
Table 63	Machine Trap Vector Register.....	83
Table 64	Encoding of mtvec.MODE	84
Table 65	Machine Interrupt Enable Register	85
Table 66	Machine Interrupt Pending Register	86
Table 67	Machine Cause Register	87
Table 68	mcause Exception Codes.....	88
Table 69	mtvt Register.....	89
Table 70	mintstatus Register	89
Table 71	RNMI CSRs	91
Table 73	E20 Core Complex Interrupt IDs	96
Table 74	CLIC Base Addresses.....	98
Table 75	CLIC Shared Region Memory Map.....	99
Table 76	CLIC Hart-Specific Region Memory Map	99
Table 77	Changes to CSRs in CLIC Mode.....	100
Table 78	CLIC Clock Gate Disable Register	100
Table 79	CLIC Interrupt Pending Register (partial)	101
Table 80	CLIC Interrupt Enable Register (partial)	101
Table 81	CLIC Interrupt Configuration Register (partial).....	102
Table 82	CLIC Configuration Register	102
Table 83	Example Encoding of cliccfg Bits to clicIntCfg[7:3]	103
Table 84	Power Dial Register	105
Table 85	Debug Module Memory Map Seen from the Debug Module Interface	111
Table 86	Debug Module Memory Map from the Perspective of the Core.....	112
Table 87	Debug and Trigger Registers	113
Table 88	Debug Control and Status Register	114
Table 89	Trigger Select Register.....	115
Table 90	Trigger Data Register 1	115
Table 91	Trigger Data Registers 2 and 3	116
Table 92	Trigger CSRs When Used as Breakpoints.....	116
Table 93	Breakpoint Match Control Register	117
Table 94	NAPOT Size Encoding.....	118
Table 95	Debug Module Interface Signals	121

Table 96	Debug Module Status Register	122
Table 97	Debug Module Control Register	123
Table 98	Hart Info Register	124
Table 99	Abstract Control and Status Register	125
Table 100	Abstract Command Register	126
Table 101	Abstract Command Autoexec Register	126
Table 102	Debug Module Control and Status 2 Register	127
Table 103	Debug Abstract Commands	127
Table 104	Abstract Command Example for 32-bit Block Write	128
Table 105	Core Generator Encoding of marchid	134
Table 106	Generator Release Encoding of mimpid	135
Table 107	E20 Core Complex Manual Revision History	136

Figures

Figure 1	E2 Series Block Diagram.....	15
Figure 2	R-Type	27
Figure 3	I-Type.....	28
Figure 4	S-Type.....	28
Figure 5	B-Type.....	28
Figure 6	U-Type	28
Figure 7	J-Type	28
Figure 8	ADD Instruction Example	29
Figure 9	ADDI Instruction Example.....	31
Figure 10	LW Instruction Example	32
Figure 11	Store Instructions.....	32
Figure 12	SW Instruction Example	33
Figure 13	JAL Instruction.....	33
Figure 14	JALR Instruction	33
Figure 15	Branch Instructions	34
Figure 16	Upper-Immediate Instructions	35
Figure 17	FENCE Instructions	35
Figure 18	NOP Instructions	36
Figure 19	Multiplication Operations	36
Figure 20	Division Operations.....	37
Figure 21	CR Format - Register	37
Figure 22	CI Format - Immediate	37
Figure 23	CSS Format - Stack-relative Store.....	38
Figure 24	CIW Format - Wide Immediate	38
Figure 25	CL Format - Load.....	38
Figure 26	CS Format - Store.....	38
Figure 27	CA Format - Arithmetic.....	38
Figure 28	CJ Format - Jump	38
Figure 29	Stack-Pointed-Based Loads.....	38

Figure 30	Stack-Pointed-Based Stores	39
Figure 31	Register-Based Loads	39
Figure 32	Register-Based Stores	40
Figure 33	Unconditional Jump Instructions	40
Figure 34	Unconditional Control Transfer Instructions	40
Figure 35	Conditional Control Transfer Instructions	41
Figure 36	Integer Constant-Generation Instructions	41
Figure 37	Integer Register-Immediate Operations	41
Figure 38	Integer Register-Immediate Operations (cont.)	42
Figure 39	Integer Register-Immediate Operations (cont.)	42
Figure 40	Integer Register-Immediate Operations (cont.)	42
Figure 41	Integer Register-Immediate Operations (cont.)	43
Figure 42	Integer Register-Register Operations	43
Figure 43	Integer Register-Register Operations (cont.)	43
Figure 44	Defined Illegal Instruction	44
Figure 45	Address Calculation Instructions	44
Figure 46	Count Leading/Trailing Zeroes Instructions	44
Figure 47	Count Population Instruction	45
Figure 48	Logic-With-Negate Instructions	45
Figure 49	Comparison Instructions	46
Figure 50	Sign-Extend Instructions	46
Figure 51	Zero-Extend Instruction	46
Figure 52	Bitwise Rotation Instructions	47
Figure 53	Bitwise Rotation Instructions (cont.)	47
Figure 54	Zicsr Instructions	48
Figure 55	Timer and Counter Pseudoinstructions	53
Figure 56	ECALL and EBREAK Instructions	55
Figure 57	Wait for Interrupt Instruction	56
Figure 58	RISC-V Assembly Example	58
Figure 59	RISC-V Assembly to Machine Code	59
Figure 60	One RISC-V Instruction	60
Figure 61	Stack Memory during Function Calls	62
Figure 62	RV32 Memory Layout	63

Figure 63	E20 Core Complex Interrupt Architecture Block Diagram	80
Figure 64	CLIC Block Diagram.....	93
Figure 65	CLIC Interrupts and Vector Table.....	95
Figure 66	CLIC Interrupt Attribute Example	97
Figure 67	CLIC Preemption Interrupt Attribute Example	98

Chapter 1

Introduction

SiFive's E20 Core Complex is an efficient implementation of the RISC-V RV32IMCB architecture. The SiFive E20 Core Complex is guaranteed to be compatible with all applicable RISC-V standards, and this document should be read together with the official RISC-V user-level, privileged, and external debug architecture specifications.



A summary of features in the E20 Core Complex can be found in Table 1.

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

Table 1: E20 Core Complex Feature Set

The E20 Core Complex also has a number of on-core-complex configurability options, allowing one to tune the design to a specific application. The configurable options are described in Appendix A.

1.1 About this Document

This document describes the functionality of the E20 Core Complex 21G2.01.00. To learn more about the Evaluation RTL deliverables of the E20 Core Complex, consult the E20 Core Complex User Guide.

1.2 About this Release

This release of E20 Core Complex 21G2.01.00 is intended for evaluation purposes only. As such, the RTL source code has been intentionally obfuscated, and its use is governed by your Evaluation License.

The full list of technical limitations of the Evaluation E20 Core Complex can be found in the E20 Core Complex User Guide.

1.3 E20 Core Complex Overview

The E20 Core Complex 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 E2 Series is shown in Figure 1. Refer to the docs/core_complex_configuration.txt file for a comprehensive summary of the E20 Core Complex configuration.

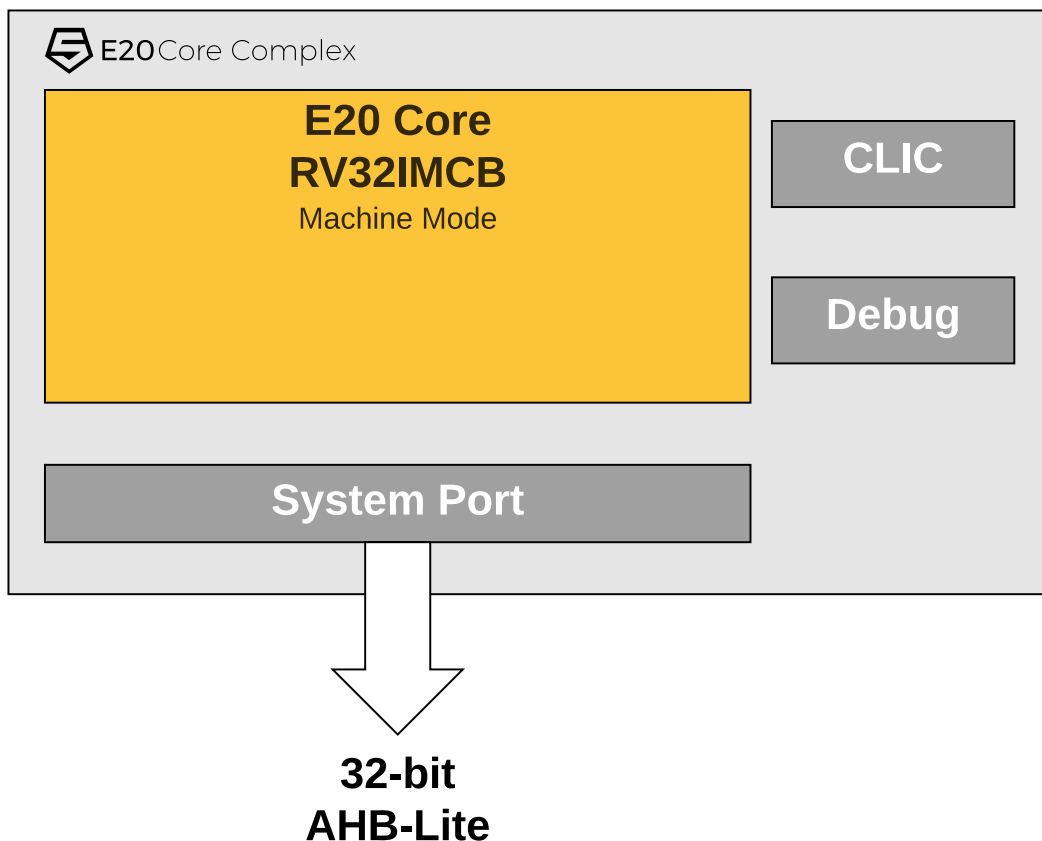


Figure 1: E2 Series Block Diagram

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

1.4 E2 RISC-V Core

The E20 Core Complex 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 SiFive E2 core is guaranteed to be compatible with all applicable RISC-V standards.

The E2 core is configured to support the RV32I base ISA, as well as the Multiply (M), Compressed (C), and Bit Manipulation (B) RISC-V extensions. This is captured by the RISC-V extension string: RV32IMCB. The base ISA and instruction extensions are described in Chapter 5.

The E2 also supports machine mode only, in conjunction with Physical Memory Protection (PMP), thereby allowing System-on-Chip (SoC) implementations to make the right area, power, and feature trade-offs.

The E2 core provides a flexible memory system that includes standards-based configurable bus interfaces, and memory maps that provide a lot of flexibility for SoC integration.

The E2 core is described in more detail in Chapter 3.

1.5 Interrupts

The E20 Core Complex 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 Core Complex 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 Core Complex User Guide.

1.7 Compliance

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

ISA	Version	Status
RV32I Base Integer Instruction Set	2.0	Frozen
Extensions	Version	Status
M Standard Extension for Integer Multiplication and Division	2.0	Ratified
C Standard Extension for Compressed Instruction	2.0	Ratified
Zba Standard Extension for Bit Manipulation	0.94	
Zbb Standard Extension for Bit Manipulation	0.94	
Privilege Mode	Version	Status
Machine-Level ISA	1.11	
Devices	Version	Status
The RISC-V Debug Specification	1.0	
RISC-V Core-Local Interrupt Controller (CLIC) Specification	20180831	

Table 2: RISC-V Specification Compliance

Chapter 2

List of Abbreviations and Terms

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

Table 3: Abbreviations and Terms

Chapter 3

E2 RISC-V Core

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

The E2 feature set is summarized in Table 4.

Feature	Description
ISA	RV32IMCB
SiFive Custom Instruction Extension (SCIE)	Not Present
Modes	Machine mode
Core Interfaces	1 core interface
Physical Memory Protection	Not Present

Table 4: E2 Feature Set

3.1 Supported Modes

The E2 supports RISC-V machine mode only.

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

3.2 Instruction Memory System

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

3.2.1 Execution Memory Space

The regions of executable memory consist of all directly addressable memory in the system. The memory includes any volatile or non-volatile memory located off the Core Complex ports.

Table 5 shows the executable regions of the E20 Core Complex.

Base	Top	Description
0x2000_0000	0x3FFF_FFFF	System Port (512 MiB)

Table 5: Executable Memory Regions for the E20 Core Complex

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

3.2.2 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 can be idle when executing programs comprised mostly of compressed 16-bit instructions. This reduces memory accesses and power consumption.

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

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.

3.3 Execution Pipeline

The E2 execution unit is a single-issue, in-order pipeline. The pipeline comprises: 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 35 cycles. Division operations can be interrupted, so they have no effect on worst-case interrupt latency.

Instruction	Latency
LW	Two-cycle latency, assuming cache hit ¹
LH, LHU, LB, LBU	Two-cycle latency, assuming cache hit ¹
CSR Reads	One-cycle latency
MUL, MULH, MULHU, MULHSU	One-cycle latency
DIV, DIVU, REM, REMU	Between five-cycle and 35-cycle latency, depending on operand values ²
¹ TIM has two-cycle access latency ² The latency of DIV, DIVU, REM, and REMU instructions can be determined by calculating: Latency = 2 cycles + $\log_2(\text{dividend}) - \log_2(\text{divisor}) + 1$ cycle if the input is negative + 1 cycle if the output is negative	

Table 6: E2 Instruction 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.

3.4 Data Memory System

The data memory system consists of the ports in the E20 Core Complex memory map, shown 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.

3.5 Local Interrupts

The E2 supports up to 32 local interrupt sources that are routed directly to the core. See Chapter 7 for a detailed description of Local Interrupts.

3.6 Hardware Performance Monitor

The E2 processor core supports a basic hardware performance monitoring (HPM) facility. The performance monitoring facility 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.

3.6.1 Performance Monitoring Counters Reset Behavior

The `instret` and `cycle` counters are initialized to zero on system reset. Users can write desired values to the counter control and status registers (CSRs) to start counting at a given, known value.

3.6.2 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 (`mcycle`)

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

3.7 Ports

This section describes the Port interfaces to the E2 core.

3.7.1 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 Core Complex User Guide for a description of the System Port implementation in the E20 Core Complex.

Chapter 4

Physical Memory Attributes and Memory Map

This chapter describes the E20 Core Complex 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 Core Complex memory regions are shown in Table 7 and Table 8 for external and internal regions, respectively.

The configured memory regions of the E20 Core Complex are listed with their attributes in Table 9.

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

Table 7: Physical Memory Attributes for External Regions

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

Table 8: Physical Memory Attributes for Internal Regions

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

The E20 Core Complex does not support the RISC-V standard Atomic (A) extension. Any atomic operation executed will generate an illegal instruction exception.

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

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

Note

The Debug region has special behavior. The Debug region is reserved for use from a Debugger, and all accesses to it from the core in non-Debug mode will trap.

4.2 Memory Map

The memory map of the E20 Core Complex is shown in Table 9.

Base	Top	PMA	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 9: E20 Core Complex 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 Core Complex implements the 32-bit RISC-V architecture. The following chapter provides a reference for programmers and an explanation of the extensions supported by RV32IMCB.

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 Core Complex is an implementation of the RISC-V RV32IMCB architecture, and is guaranteed to be compatible with all applicable RISC-V standards. RV32IMCB 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 10 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 10: Base Instruction Formats

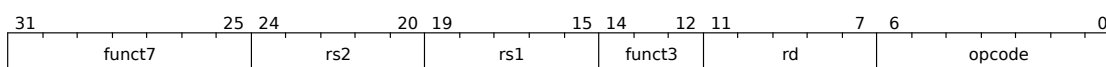


Figure 2: R-Type

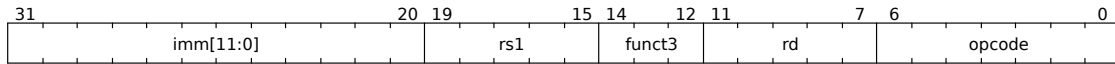


Figure 3: I-Type

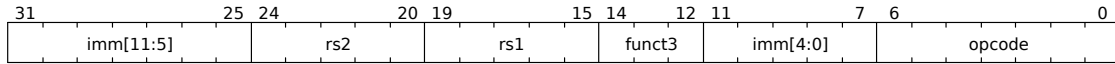


Figure 4: S-Type

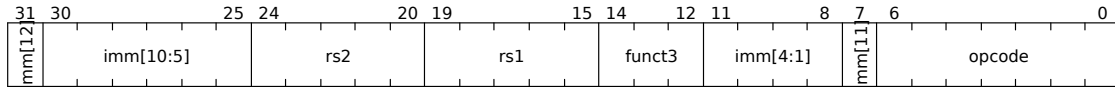


Figure 5: B-Type

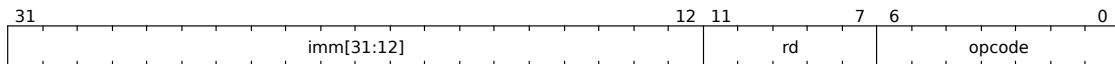


Figure 6: U-Type

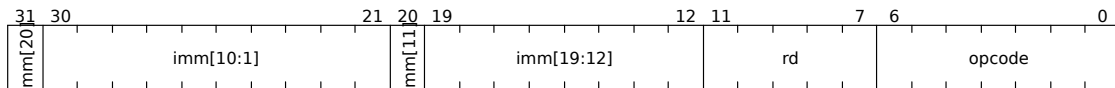


Figure 7: J-Type

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

Table 11: R-Type Integer Instructions

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

Table 12: R-Type Integer Instruction Description

Below is an example of an ADD instruction.

add x18, x19, x10

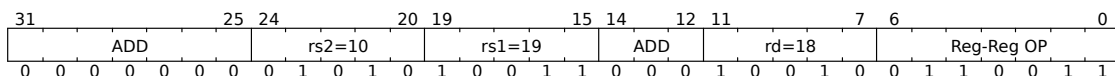


Figure 8: ADD Instruction Example

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

Table 13: I-Type Integer Instructions

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

Instruction	Description
ADDI	Adds the sign-extended 12-bit immediate to register <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).

Table 14: I-Type Integer Instruction Description

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

Below is an example of an ADDI instruction.

addi x15, x1, -50

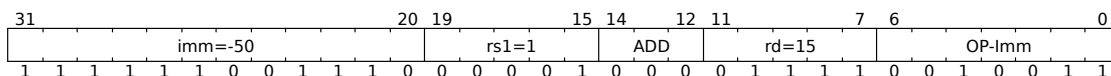


Figure 9: ADDI Instruction Example

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 15 below, **funct3** field encodes size and signedness of load data.

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

Table 15: I-Type Load Instructions

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

Table 16: I-Type Load Instruction Description

Below is an example of a LW instruction.

lw x14, 8(x2)

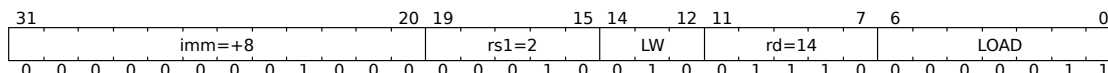


Figure 10: LW Instruction Example

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.

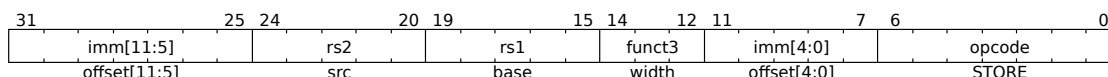


Figure 11: Store Instructions

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 17: S-Type Store Instructions

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

Table 18: S-Type Store Instruction Description

Below is an example SW instruction.

sw x14, 8(x2)

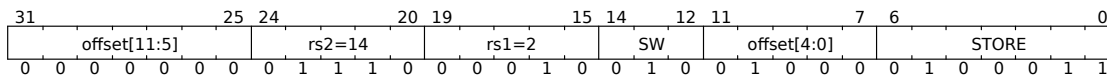


Figure 12: SW Instruction Example

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 ± 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.

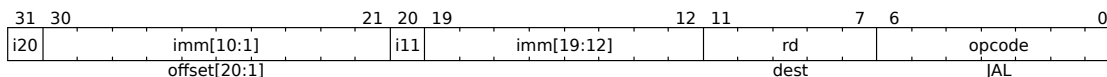


Figure 13: JAL Instruction

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.

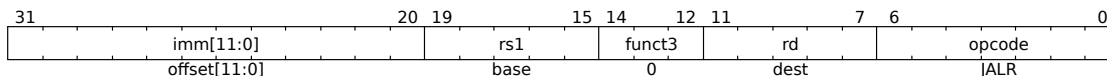


Figure 14: JALR Instruction

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

Table 19: J-Type Instruction Description

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 ± 4 KiB.

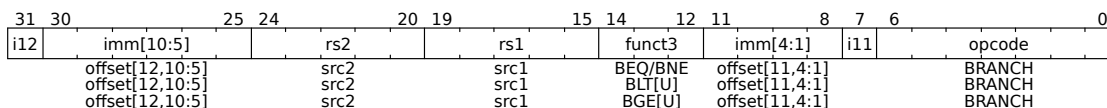


Figure 15: Branch Instructions

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

Table 20: B-Type Instructions

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

Table 21: B-Type Instruction Description

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

Table 22: RISC-V Base Instruction to Assembly Pseudoinstruction Example

Note

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

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

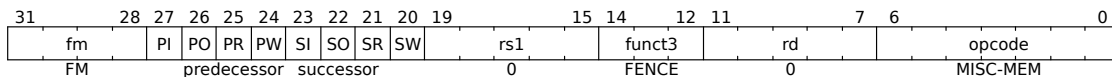


Figure 17: FENCE Instructions

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 5.10.

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

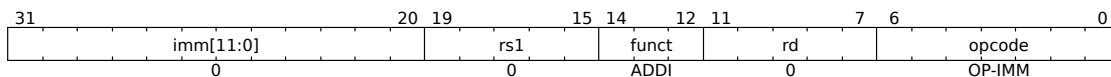


Figure 18: NOP Instructions

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

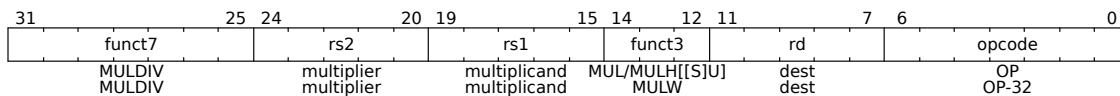


Figure 19: Multiplication Operations

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.

Table 23: Multiplication Operation Description

Combining MUL and MULH together creates one multiplication operation.

5.3.1 Division Operations

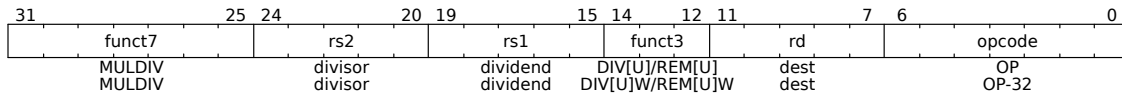


Figure 20: Division Operations

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.

Table 24: Division Operation Description

Combining DIV and REM together creates one 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

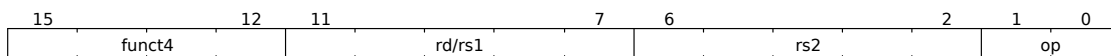


Figure 21: CR Format - Register

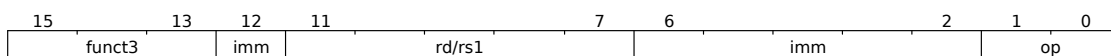


Figure 22: CI Format - Immediate

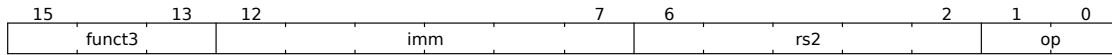


Figure 23: CSS Format - Stack-relative Store

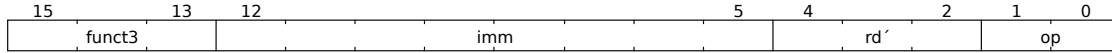


Figure 24: CIW Format - Wide Immediate

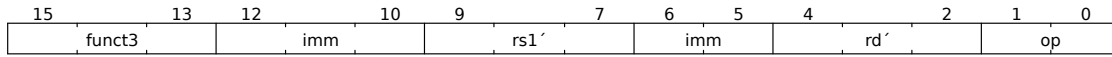


Figure 25: CL Format - Load

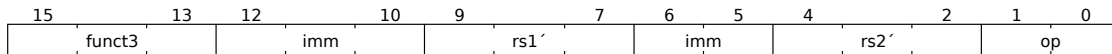


Figure 26: CS Format - Store

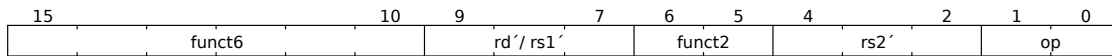


Figure 27: CA Format - Arithmetic

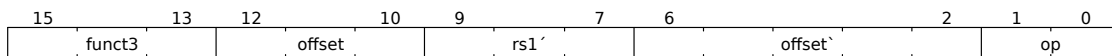


Figure 28: CJ Format - Jump

5.4.2 Stack-Pointed-Based Loads and Stores

The compressed load instructions are expressed in CI format.

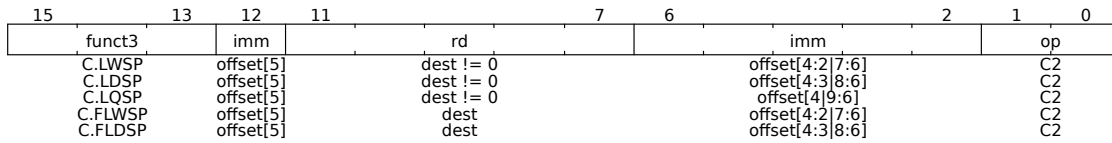


Figure 29: Stack-Pointed-Based Loads

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.

Table 25: Stack-Pointed-Based Load Instruction Description

The compressed store instructions are expressed in CSS format.

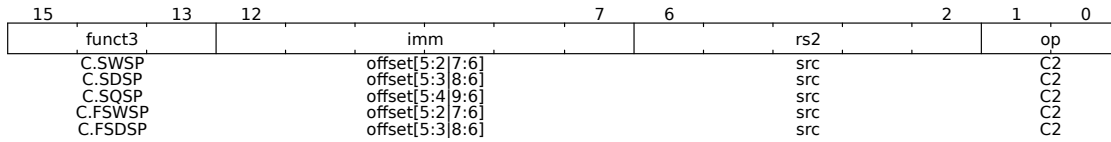


Figure 30: Stack-Pointed-Based Stores

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.

Table 26: Stack-Pointed-Based Store Instruction Description

5.4.3 Register-Based Loads and Stores

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

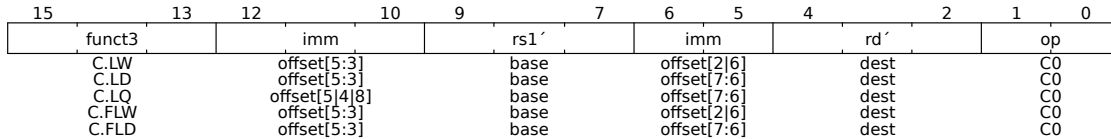


Figure 31: Register-Based Loads

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.

Table 27: Register-Based Load Instruction Description

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

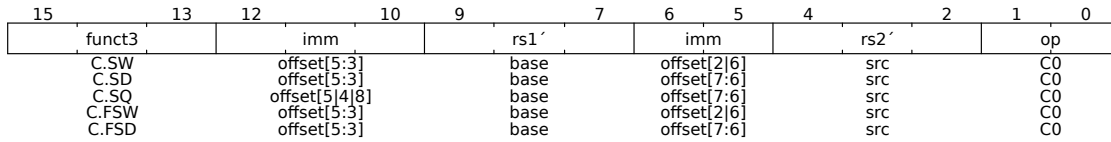


Figure 32: Register-Based Stores

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.

Table 28: Register-Based Store Instruction Description

5.4.4 Control Transfer Instructions

RVC provides unconditional jump instructions and conditional branch instructions.

The unconditional jump instructions are expressed in CJ format.

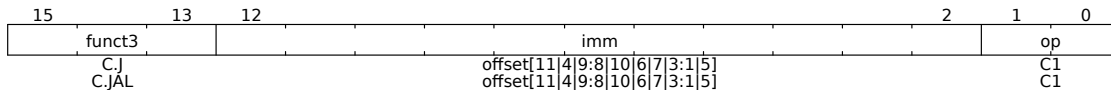


Figure 33: Unconditional Jump Instructions

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.

Table 29: Unconditional Jump Instruction Description

The unconditional control transfer instructions are expressed in CR format.

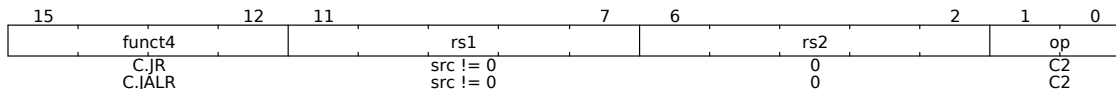


Figure 34: Unconditional Control Transfer Instructions

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> .

Table 30: Unconditional Control Transfer Instruction Description

The conditional control transfer instructions are expressed in CB format.

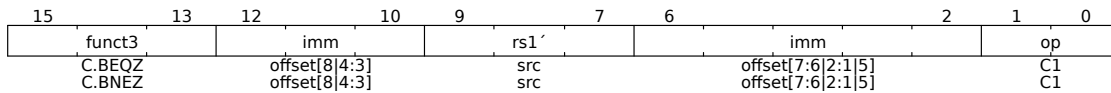


Figure 35: Conditional Control Transfer Instructions

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.

Table 31: Conditional Control Transfer Instruction Description

5.4.5 Integer Computational Instructions

Integer Constant-Generation Instructions

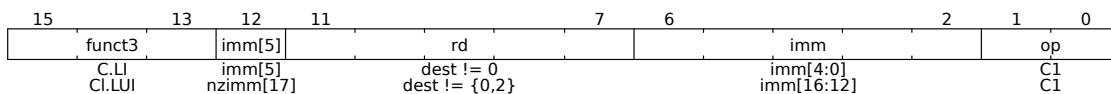


Figure 36: Integer Constant-Generation Instructions

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

Table 32: Integer Constant-Generation Instruction Description

Integer Register-Immediate Operations

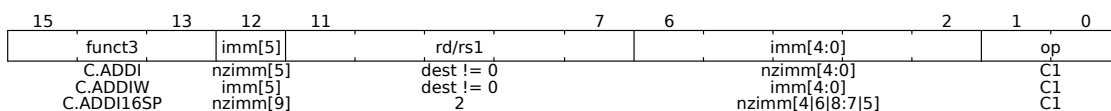


Figure 37: Integer Register-Immediate Operations

Instruction	Description
C.ADDI	Adds the non-zero sign-extended 6-bit immediate to the value in register <i>rd</i> then writes the result to <i>rd</i> .
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 (<i>sp=x2</i>), where the immediate is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues.

Table 33: Integer Register-Immediate Operation Description

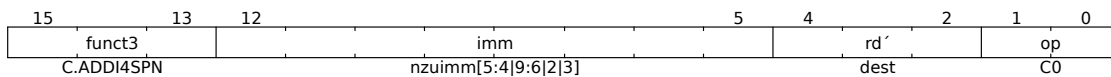


Figure 38: Integer Register-Immediate Operations (cont.)

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

Table 34: Integer Register-Immediate Operation Description (cont.)

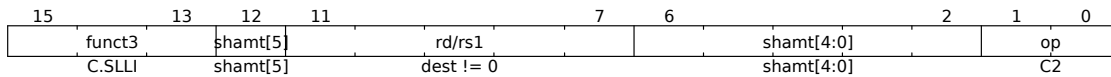


Figure 39: Integer Register-Immediate Operations (cont.)

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

Table 35: Integer Register-Immediate Operation Description (cont.)

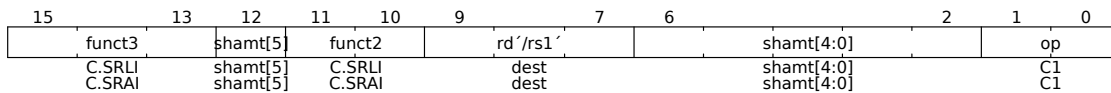


Figure 40: Integer Register-Immediate Operations (cont.)

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

Table 36: Integer Register-Immediate Operation Description (cont.)

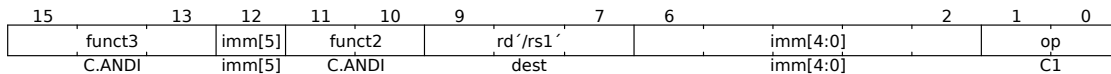


Figure 41: Integer Register-Immediate Operations (cont.)

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

Table 37: Integer Register-Immediate Operation Description (cont.)

Integer Register-Register Operations

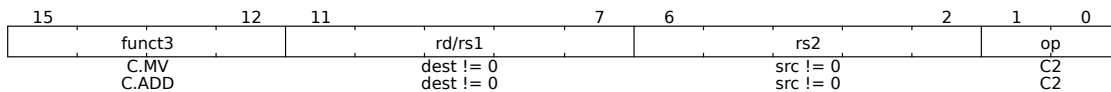


Figure 42: Integer Register-Register Operations

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

Table 38: Integer Register-Register Operation Description

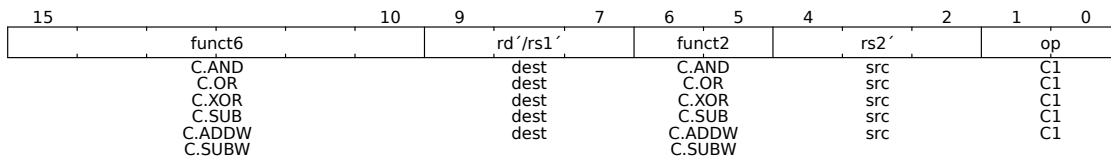


Figure 43: Integer Register-Register Operations (cont.)

Instruction	Description
C.AND	Computes the bitwise AND of the values in registers rd' and rs2'.
C.OR	Computes the bitwise OR of the values in registers rd' and rs2'.
C.XOR	Computes the bitwise XOR of the values in registers rd' and 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.

Table 39: Integer Register-Register Operation Description (cont.)

Defined Illegal Instruction

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

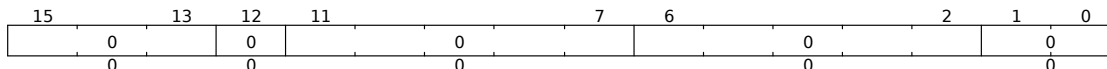


Figure 44: Defined Illegal Instruction

5.5 B Extension: Bit Manipulation Instructions

This section discusses the bit manipulation instructions supported by RISC-V. SiFive implements the Zba and Zbb instructions of the B extension.

5.5.1 Zba Extension

The Zba instructions are used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and 32-sized indices; that is, a shifted index is added to a base address.

Address Calculation Instructions

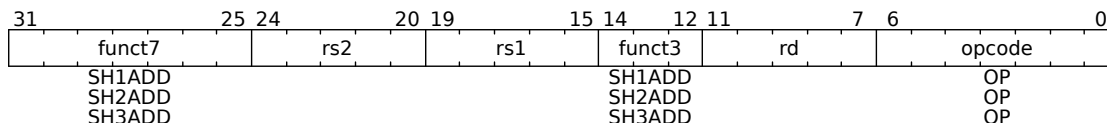


Figure 45: Address Calculation Instructions

Instruction	Description
SH1ADD rd, rs1, rs2	Shifts rs1 by 1 bit, then adds the result to rs2
SH2ADD rd, rs1, rs2	Shifts rs1 by 2 bits, then adds the result to rs2
SH3ADD rd, rs1, rs2	Shifts rs1 by 3 bits, then adds the result to rs2

Table 40: Address Calculation Instructions Description

5.5.2 Zbb Extension

The Zbb instructions are used for basic bit manipulation.

Count Leading/Trailing Zeros Instructions

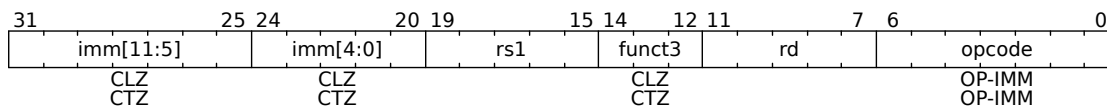


Figure 46: Count Leading/Trailing Zeros Instructions

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

Table 41: Count Leading/Trailing Zeroes Instructions Description

Count Population Instructions

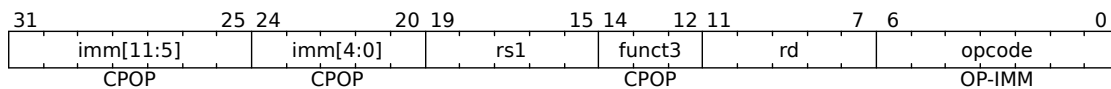


Figure 47: Count Population Instruction

Instruction	Description
CPOP rd, rs	Counts the number of 1 bits in the source register

Table 42: Count Population Instructions Description

Logic-With-Negate Instructions

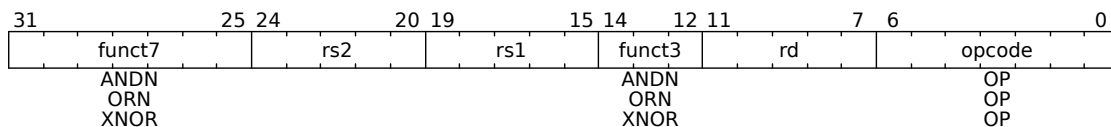


Figure 48: Logic-With-Negate Instructions

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

Table 43: Logic-With-Negate Instructions Description

Comparison Instructions

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

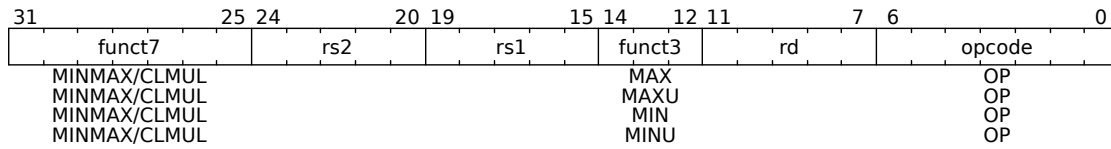


Figure 49: Comparison Instructions

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

Table 44: Comparison Instructions Description

Sign-Extend and Zero-Extend Instructions

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

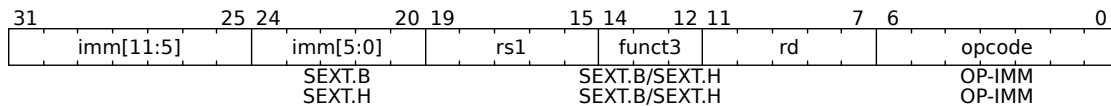


Figure 50: Sign-Extend Instructions

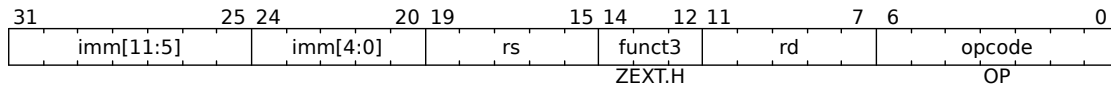


Figure 51: Zero-Extend Instruction

Instruction	Description
SEXT.B rd, rs	Sign-extends the least-significant byte in the source to 32 by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits
SEXT.H rd, rs	Sign-extends the least-significant halfword in <i>rs</i> to 32 by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits
ZEXT.H rd, rs	Zero-extends the least-significant halfword of the source to 32 by inserting 0's into all of the bits more significant than 15

Table 45: Sign- and Zero-Extend Instructions

Bitwise Rotation Instructions

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

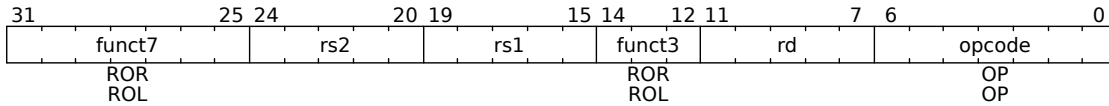


Figure 52: Bitwise Rotation Instructions

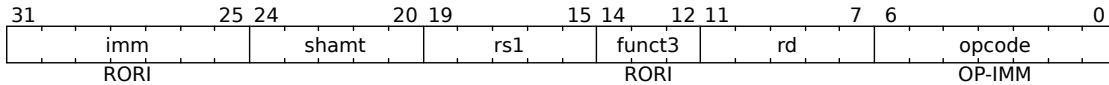


Figure 53: Bitwise Rotation Instructions (cont.)

Instruction	Description
ROR rd, rs1, rs2	Performs a rotate right shift of rs1 by the amount in the least-significant 5 bits of rs2
ROL rd, rs1, rs2	Performs a rotate left shift of rs1 by the amount in the least-significant 5 bits of rs2
RORI rd, rs1, shamt	Performs a rotate right shift of rs1 by the amount in the least significant 5 bits of shamt. The encodings corresponding to shamt[5]=1 are reserved.

Table 46: Bitwise Rotation Instructions Description

OR Combine Instruction

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

Table 47: OR Combine Instruction Description

Byte-Reverse Instruction

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

Table 48: Byte-Reverse Instruction Description

5.6 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.

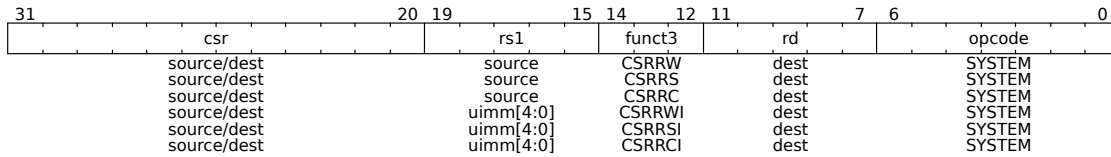


Figure 54: Zicsr Instructions

Instruction	Description
CSRRW rd, rs1 csr	Instruction atomically swaps values in the CSRs and integer registers.
CSRRS rd, rs1 csr	Instruction reads the value of the CSR, zero-extends the value to 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, zero-extends 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 an 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 an 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.

Table 49: Control and Status Register Instruction Description

The CSRRWI, CSRRSI, and CSRRCI instructions are similar in kind to CSRRW, CSRRS, and CSRRC respectively, except in that they update the CSR using an 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 size effects that might otherwise occur on a CSR write. For CSRRWI, if rd = x0, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read. Both CSRRSI and CSRRCI will always read the CSR and cause any read side effects regardless of the rd and rs1 fields.

Table 50 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 50: CSR Reads and Writes

5.6.1 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.

5.6.2 Defined CSRs

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

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	RO	mvendorid	Vendor ID.
0xF12	RO	marchid	Architecture ID.
0xF13	RO	mimpid	Implementation ID.
0xF14	RO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	RW	mstatus	Machine status register.
0x301	RW	misa	ISA and extensions.
0x302	RW	medeleg	Machine exception delegation register.
0x303	RW	mideleg	Machine interrupt delegation register.
0x304	RW	mie	Machine interrupt-enable register.
0x305	RW	mtvec	Machine trap-handler base address.
0x306	RW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	RW	mscratch	Scratch register for machine trap handlers.
0x341	RW	mepc	Machine exception program counter.
0x342	RW	mcause	Machine trap cause.
0x343	RW	mtval	Machine bad address or instruction.
0x344	RW	mip	Machine interrupt pending.
Machine Counter/Timers			
0xB00	RW	mcycle	Machine cycle counter.
0xB02	RW	minstret	Machine instruction-retired counter.
0xB80	RW	mcycleh	Upper 32 bits of mcycle, RV32I only.
0xB82	RW	minstreth	Upper 32 bits of minstret, RV32I only.
Machine Counter Setup			
0x320	RW	mcountinhibit	Machine counter-inhibit register.
Debug/Trace Register (shared with Debug Mode)			
0x7A0	RW	tselect	Debug/Trace trigger register select.
0x7A1	RW	tdata1	First Debug/Trace trigger data register.
0x7A2	RW	tdata2	Second Debug/Trace trigger data register.
0x7A3	RW	tdata3	Third Debug/Trace trigger data register.

Table 51: 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 52: Debug Mode Registers

5.6.3 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 5.10. 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.

5.6.4 SiFive RISC-V Implementation Version Registers

`mvendorid`

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

`marchid`

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

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

Value	Core Generator
0x8000_0002	E2/S2-Series Processor

Table 53: Core Generator Encoding of marchid

mimpid

The value in `mimpid` holds an encoded value that uniquely identifies the version of the generator used to build this implementation. If your release version is not included in Table 54, contact your SiFive account manager for more information.

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

Table 54: Generator Release Encoding of mimpid

Reading Implementation Version Registers

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

In C:

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

In Assembly:

```
csrr a5, mimpid
```

5.6.5 Custom CSRs

SiFive implements some custom CSRs that are specific to the implementation. For these CSRs, including the Feature Disable CSR, consider Chapter 6.

5.7 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 Core Complex implements `mcycle`, `mtime`, and `minstret` counters, which have dedicated functions: cycle count, real-time clock, and instructions-retired, respectively. The timer functionality is based on the `mtime` register. Additionally, the E20 Core Complex implements event counters in the form of `mhpmcounter`, which is used to monitor user requested events.

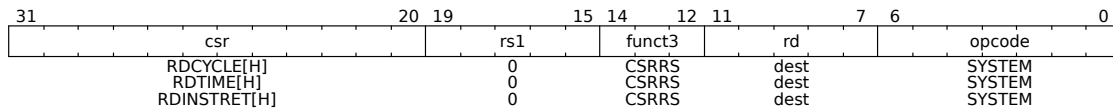


Figure 55: Timer and Counter Pseudoinstructions

Instruction	Description
RDCYCLE rd	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	RV32I instruction that reads bits 63–32 of the same cycle counter.
RDTIME rd	Generates an illegal instruction exception. The <code>mtime</code> register is memory mapped to the CLIC register space and can be read using a regular load instruction.
RDTIMEH rd	RV32I-only instruction. Generates an illegal instruction exception. The <code>mtime</code> register is memory mapped to the CLIC register space and can be read using a regular load instruction.
RDINSTRET rd	Reads the low 32-bits of the <code>instret</code> CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past.
RDINSTRETH rd	RV32I-only instruction that reads bits 63–32 of the same instruction counter.

Table 55: Timer and Counter Pseudoinstruction Description

RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions read the full 64 bits of the `cycle`, `time`, and `instret` counters. The RDCYCLE pseudoinstruction reads the low 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 5.7.2 below.

Number	Privilege	Name	Description
0xC00	RO	<code>cycle</code>	Cycle counter for RDCYCLE instruction
0xC01	RO	<code>time</code>	Timer for RDTIME instruction
0xC02	RO	<code>instret</code>	Instruction-retired counter for RDINSTRET instruction
0xC80	RO	<code>cycleh</code>	Upper 32 bits of cycle, RV32 only.
0xC81	RO	<code>timeh</code>	Upper 32 bits of time, RV32 only.
0xC82	RO	<code>instreth</code>	Upper 32 bits of instret, RV32 only

Table 56: Timer and Counter CSRs

5.7.1 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 Core Complex User Guide. On reset, `mtime` is cleared to zero.

5.7.2 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 meta1_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 meta1_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

5.8 Privileged Instructions

The RISC-V architecture implements privileged instructions that can only be executed when the E20 Core Complex is operating in a privileged mode. The SYSTEM major opcode is used to encode all of the privileged instructions.

5.8.1 Machine-Mode Privileged Instructions

Environment Call and Breakpoint

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

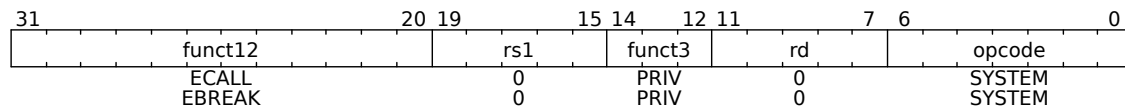


Figure 56: ECALL and EBREAK Instructions

Trap-Return Instructions

To return after handling a trap, there are separate trap return instructions per privilege level: MRET, SRET, and URET. MRET is always provided, while SRET must be provided if the respective privilege mode is supported. URET is only provided if user-mode traps are supported. An xRET instruction can be executed in privilege mode x or higher, where executing a

lower-privilege xRET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack.

Wait for Interrupt

The Wait for Interrupt (WFI) instruction provides a hint to the E20 Core Complex that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart.

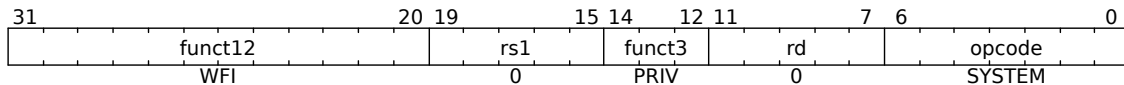


Figure 57: Wait for Interrupt Instruction

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

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

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

The WFI instruction is just a hint, and a legal implementation is to implement WFI as a NOP. In SiFive's implementation of WFI, the WFI instruction is issued and the core goes into internal clock gating state.

5.9 ABI - Register File Usage and Calling Conventions

RV32IMCB 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-pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments / return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller
Floating-Point Registers			
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments / return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fa2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Table 57: 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.

5.9.1 RISC-V Assembly

RISC-V instructions have opcodes and operands.

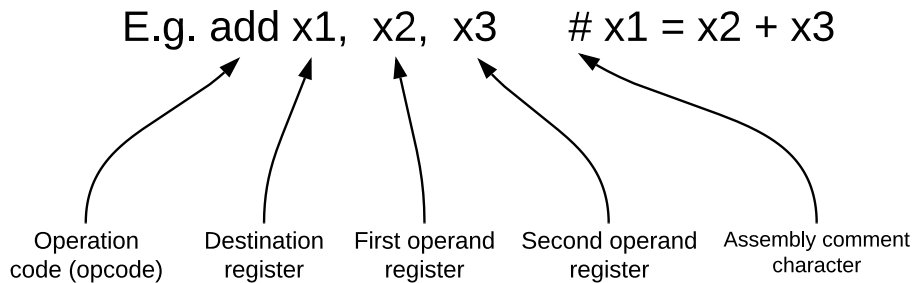


Figure 58: RISC-V Assembly Example

Assembly	C	Description
add x1, x2, x3	a = b + c	a=x1, b=x2, c=x3
sub x3, x4, x5	d = e - f	d=x3, e=x4, f=x5
add x0, x0, x0	NOP	Writes to x0 are always ignored
add x3, x4, x0	f = g	f=x3, g=x4
addi x3, x4, -10	f = g - 10	f=x3, g=x4
lw x10, 12(x13) # 12 = 3x4 add x11, x12, x10	int A[100]; g = h + A[3];	Reg x10 gets A[3] g=x11, h=x12
lw x10, 12(x13) # 12 = 3x4 add x10, x12, x10 sw x10, 40(x13) # 40 = 10x4	int A[100]; A[10] = h + A[3];	Reg x10 gets A[3] h=x12 Reg x10 gets h + A[3]
bne x13, x14, done add x10, x11, x12 done:	if (i == j) f = g + h;	f=x10, g=x11, h=x12, i=x13, j=x14
bne x10, x14, else add x10, x11, x12 j done else: sub x10, x11, x12 done:	if (i == j) f = g + h; else f = g - h;	f=x10, g=x11, h=x12, i=x13, j=x14

Table 58: RISC-V Assembly and C Examples

5.9.2 Assembler to Machine Code

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

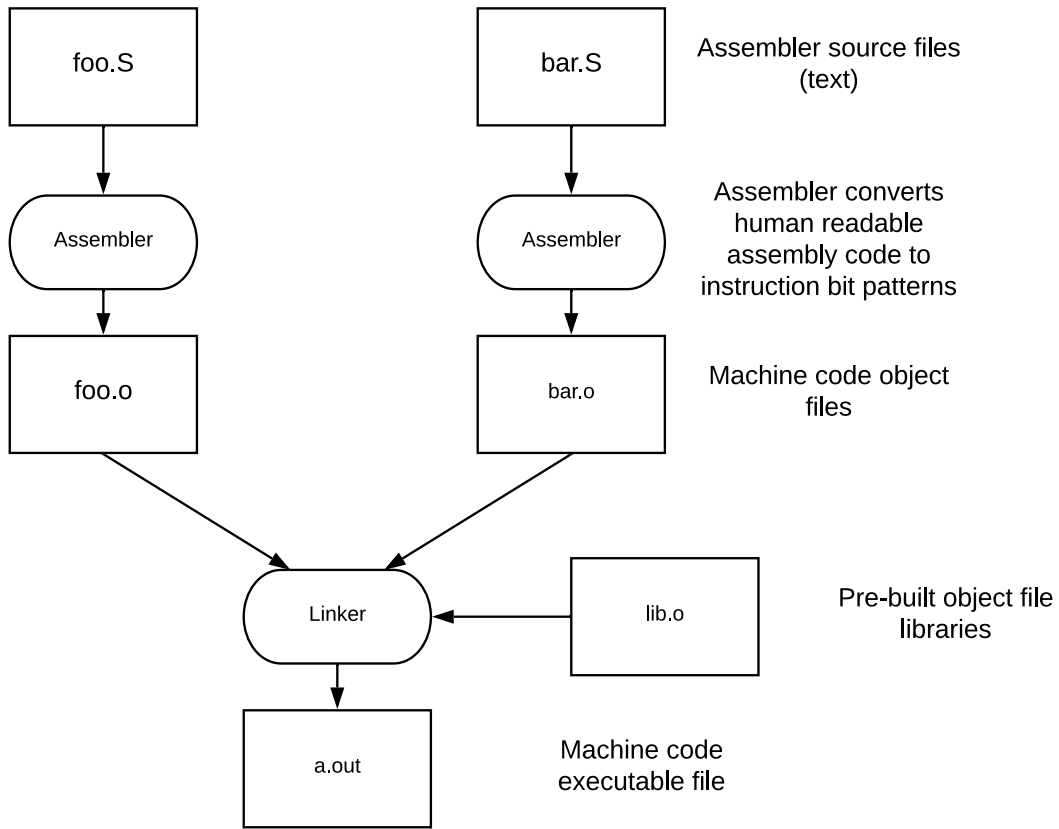


Figure 59: RISC-V Assembly to Machine Code

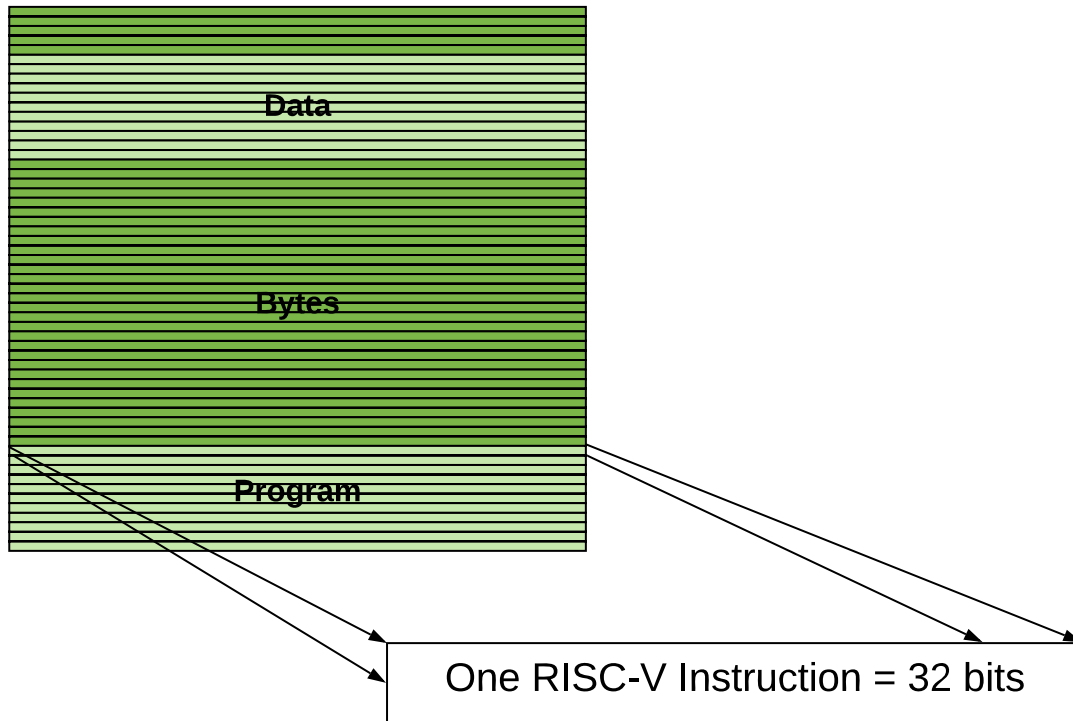


Figure 60: One RISC-V Instruction

5.9.3 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 s1, 4(sp)    # adjust stack to delete 2 items
jr ra             # jump back to calling routine
```

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

Nested Functions

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

Take, for example, the following function:

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

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

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

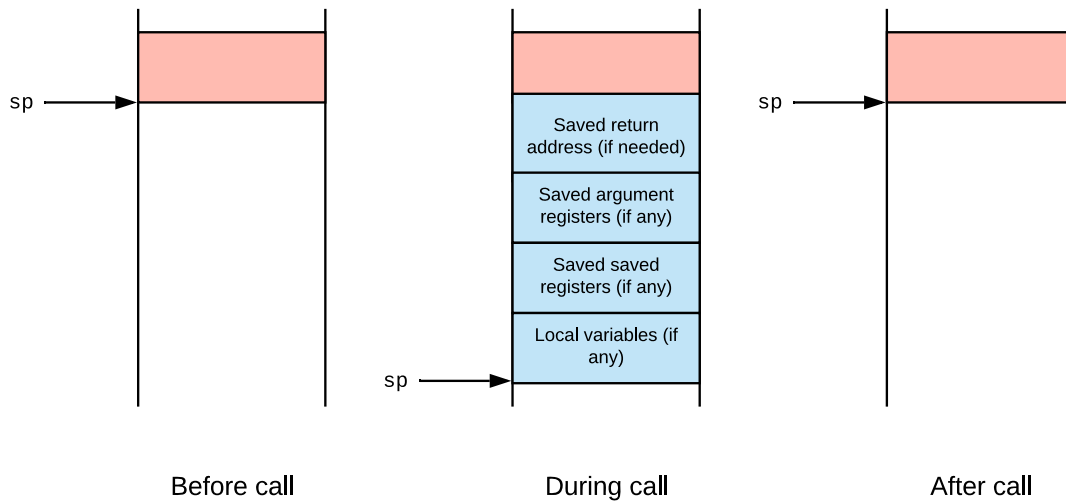


Figure 61: Stack Memory during Function Calls

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

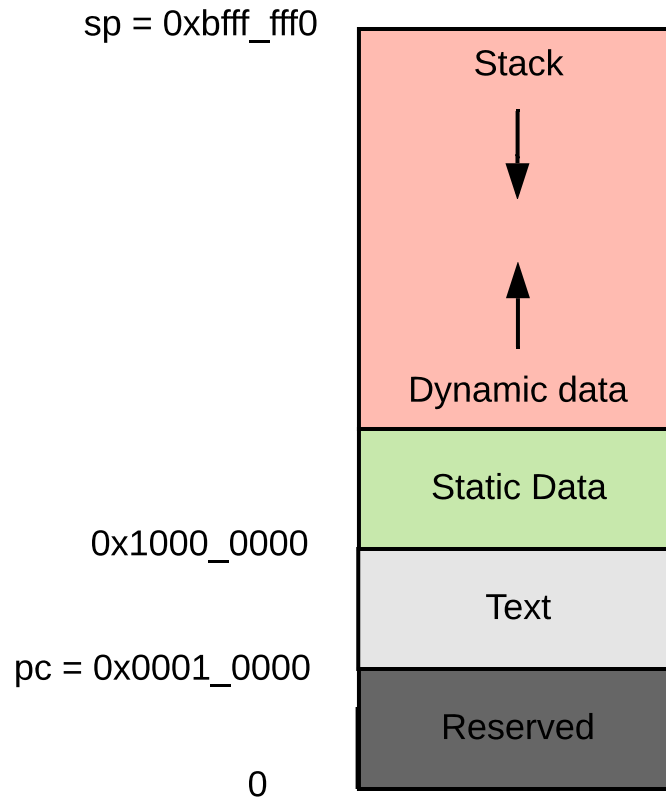


Figure 62: RV32 Memory Layout

5.10 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.

5.11 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 feature disable CSR 0x7c1.
5. Read mhartid into register a0 and call _start, which exists in crt0.S.
6. We now transition to File: freedom-metal/gloss/crt0.S, label: _start.
7. Initialize stack pointer, sp, with _sp generated symbol. Harts with mhartid of one or larger are offset by ($_sp + _stack_size \times mhartid$). 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 `crto.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.

5.12 Linker File

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

There are usually three different linker file options:

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

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

For example:

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

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

5.12.1 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

5.13 RISC-V Compiler Flags

5.13.1 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 (ilp32 vs. ilp32f vs. ilp32d). In order for objects to be linked together, they must follow the same ABI.

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

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

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

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

arch/abi Combinations

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

Example:

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

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

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imac -mabi=ilp32 -o- -S -O3  
dmul:  
    mv      a4,a2
```

```
mv    a5,a3
add   sp,sp,-16
mv    a2,a0
mv    a3,a1
mv    a0,a4
mv    a1,a5
sw    ra,12(sp)
call  __muldf3
lw    ra,12(sp)
add   sp,sp,16
jr    ra
```

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

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32d -o- -S -03
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 -03
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
```


5.14 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).

5.15 Large Code Model Workarounds

RISC-V software currently requires that linked symbols reside within a 32-bit range. There are two types of code models defined for RISC-V, **medlow** and **medany**. The **medany** code model generates `auipc/ld` pairs to refer to global symbols, which allows the code to be linked at any address, while **medlow** generates `lui/ld` pairs to refer to global symbols, which restricts the code to be linked around address zero. They both generate 32-bit signed offsets for referring to symbols, so they both restrict the generated code to being linked within a 2 GiB window. When building software, the code model parameter is passed into the RISC-V toolsuite and it defines a method to generate the necessary instruction combinations to access global symbols within the software program. This is done using `-mcmode1=medany/medlow`. For 32-bit architectures, we use the **medlow** code model, while **medany** is used for 64-bit architectures. This is controlled within the 'setting.mk' file in the `freedom-e-sdk/bsp` folder.

The real problem occurs when:

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

Example for symbols within 32-bit address space:

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

Example for symbols outside 32-bit address space:

```
MEMORY
```

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

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

relocation truncated to fit:

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

5.15.1 RISC-V Code Model Summary

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

Table 59: RISC-V Code Model Table

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

5.15.2 Enabling the Compact Code Model

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

1. Enable compact code model in settings.mk file to use `RISCV_CMODEL = compact` instead of `RISCV_CMODEL = medany`.
2. Update assembly files to use new instructions if `__riscv_cmodel_compact` is defined by the toolsuite.
3. Update linker alignment.
4. Use the latest GCC+LLVM toolsuite from SiFive, starting with 2021.06.2.

Makefile Update

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

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

The `RISCV_CMODEL` definition gets passed into the toolsuite using the `-mcmode1` switch.

Note

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

This `-mcmode1=compact` option will enable the symbol `__riscv_cmodel_compact` to be visible within the code, and can be used to determine the correct code sequences to use within assembly files.

Assembly File Updates

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

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

Linker Alignment

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

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

5.16 Pipeline Hazards

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

5.16.1 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.

5.16.2 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.

5.17 Reading CSRs

There are several methods for reading the CSRs that are implemented in the E20 Core Complex. A full list of the defined RISC-V CSRs are described in Section 5.6.2.

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

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

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

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

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

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

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

Note

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

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

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

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

Chapter 6

Custom Instructions and CSRs

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_N` signal to the outside of the Core Complex, indicating that it is safe to power down.
- Debug `haltreq` will not work after a CEASE instruction has retired.

6.2 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 Core Complex.

Chapter 7

Interrupts and Exceptions

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

Specifically, the E20 Core Complex 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 1.11*. 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 Core Complex 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 Core Complex 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 Core Complex 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 60 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 60: Exception Priority

Refer to Table 68 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.

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

CSR	Description
exception	SiFive Scope signal. Indicates the moment that an exception occurs in the write-back (commit) stage.
mcause	Contains the cause value of the exception/interrupt. See Section 7.7.5 for more description.
mepc	Contains the pc where the exception occurs.
mtval	If the cause is a load/store fault, this register has the value of the problematic address. If it is an invalid instruction, it provides the instruction that the core tried to execute.
mstatus	Contains the interrupt enables, privilege modes, and general status of execution. See Section 7.7.1 for more description.
mtvec	Contains the vector that the core will jump to when an exception occurs. If this is not a valid executable value, you may get a double exception when jumping to the exception handler, so it is important to look at all these registers when the exception FIRST occurs. See Section 7.7.2 for more description.

Table 61: Summary of Exception and Interrupt CSRs

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 Core Complex interrupt architecture is depicted in Figure 63.

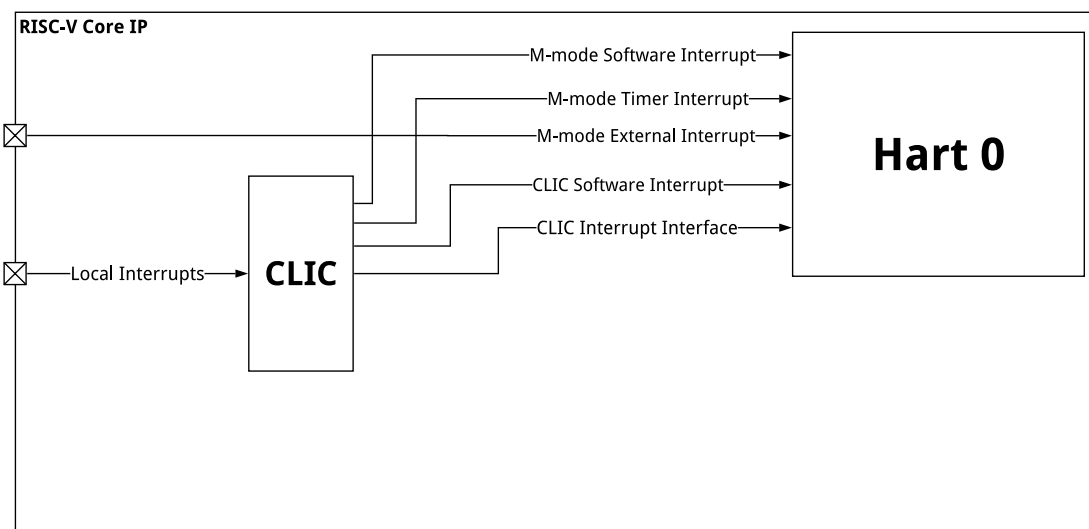


Figure 63: E20 Core Complex Interrupt Architecture Block Diagram

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 `cllicIntIP` register when in CLIC modes of operation. The `msip` register is described in Table 66 and `cllicIntIP` is described in Table 79.

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 Core Complex 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. When this interrupt is triggered, such as by writing to the `msip` register, it is expected that the interrupt is not always taken on the instruction that follows the write to the register. To stall execution until the interrupt is taken, use a branch to self instruction.

Local external interrupts (Interrupt ID #16–47) may connect directly to an interrupt source. The E20 Core Complex 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 `cllicIntcfg` 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 64.

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 Core Complex specific implementation of interrupt CSRs is described below. For a complete description of RISC-V interrupt behavior and how to access CSRs, please consult *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* 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 Core Complex is provided in Table 62. Note that this is not a complete description of `mstatus` as it contains fields unrelated to interrupts. For the full description of `mstatus`, please consult *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

Machine Status Register (mstatus)			
CSR	0x300		
Bits	Field Name	Attr.	Description
[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 62: Machine Status Register (partial)

Interrupts are enabled by setting the MIE bit in `mstatus`. Prior to writing `mstatus.MIE=1`, it is recommended to first enable interrupts in `mie` 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 Core Complex will process interrupts. For Direct and Vectored modes, the interrupt processing mode is defined in the `MODE` field of the `mtvec` register. The `mtvec` register is described in Table 63, and the `mtvec.MODE` field is described in Table 64.

Machine Trap Vector Register (mtvec)			
CSR	0x305		
Bits	Field Name	Attr.	Description
[1:0]	MODE	WARL	MODE Sets the interrupt processing mode. The encoding for the E20 Core Complex supported modes is described in Table 64.
[31:2]	BASE[31:2]	WARL	Interrupt Vector Base Address. Operating in CLINT Direct Mode requires 4-byte alignment. Operating in CLINT Vectored Mode requires 128-byte alignment. Operating in CLIC mode requires minimum 64-byte alignment.

Table 63: Machine Trap Vector Register

MODE Field Encoding <code>mtvec.MODE</code>		
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 × <code>mcause.EXCCODE</code> .
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 <code>mtvt + (mcause.EXCCODE × 4)</code> .

Table 64: 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 67.

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.MINH_V`, 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.MINH_V`. 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 65.

Machine Interrupt Enable Register (<code>mie</code>)			
CSR	0x304		
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 65: Machine Interrupt Enable Register

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

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 66.

Machine Interrupt Pending Register (mip)			
CSR	0x344		
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 66: Machine Interrupt Pending 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 67 for more details about the mcause register. Refer to Table 68 for a list of synchronous exception codes.

Machine Cause Register (mcause)			
CSR	0x342		
Bits	Field Name	Attr.	Description
[9:0]	EXCCODE	WLRL	A code identifying the last exception.
[15:10]	Reserved	WLRL	
[23:16]	MPIL	WLRL	Previous interrupt level. CLIC mode only.
[26:24]	Reserved	WLRL	
27	MPIE	WLRL	Previous interrupt enable, same as <code>mstatus.MPIE</code> . 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 67: Machine Cause Register

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	Reserved
1	14	Debug interrupt
1	15	Reserved
1	16	CLIC Local Interrupt 0
1	17	CLIC Local Interrupt 1
1	18–46	...
1	47	CLIC Local Interrupt 31
1	≥48	Reserved
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8–10	Reserved
0	11	Environment call from M-mode
0	12–13	Reserved
0	14	Debug
0	≥15	Reserved

Table 68: mcause Exception Codes

Note that there are scenarios where a misaligned load or store will generate an access exception instead of an address-misaligned exception. The access exception is raised when the misaligned access should not be emulated in a trap handler, e.g., emulating an access in an I/O region, as such emulation could cause undesirable side-effects.

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			
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 69: 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			
Bits	Field Name	Attr.	Description
[23:0]	Reserved	WIRI	
[31:24]	MIL	WIRL	Active Machine Mode Interrupt Level

Table 70: 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 `cllicIntiE` 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 Core Complex 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.

7.9 Non-Maskable Interrupt

The `rnmi` (resumable non-maskable interrupt) interrupt signal is a level-sensitive input to the hart. Non-maskable interrupts have higher priority than any other interrupt or exception on the hart and cannot be disabled by software. Specifically, they are not disabled by clearing the `mstatus.mie` register.

7.9.1 Handler Addresses

The NMI has an associated exception trap handler address. This address is set by external input signals, described in the E20 Core Complex User Guide.

7.9.2 RNMI CSRs

These M-mode CSRs enable a resumable non-maskable interrupt (RNMI).

Number	Name	Description
0x350	mnscratch	Resumable Non-maskable scratch register
0x351	mnepc	Resumable Non-maskable EPC value
0x352	mncause	Resumable Non-maskable cause value
0x353	mnstatus	Resumable Non-maskable status

Table 71: RNMI CSRs

- The `mnscratch` CSR holds a 32-bit read-write register, which enables the NMI trap handler to save and restore the context that was interrupted.
- The `mnepc` CSR is a 32-bit read-write register, which, on entry to the NMI trap handler, holds the PC of the instruction that took the interrupt. The lowest bit of `mnepc` is hardwired to zero.
- The `mncause` CSR holds the reason for the NMI, with bit 31 set to 1, and the NMI cause encoded in the least-significant bits, or zero if NMI causes are not supported. The lower bits of `mncause`, defined as the `exception_code`, are as follows:

mncause	NMI Cause	Function
1	Reserved	Reserved
2	RNMI input pin	External <code>rnmi_N</code> input
3	Reserved	Reserved

Table 72: `mncause.exception_code` Fields

- The `mnstatus` CSR holds a two-bit field, which, on entry to the trap handler, holds the privilege mode of the interrupted context encoded in the same manner as `mstatus.mpp`.

7.9.3 MNRET Instruction

This M-mode only instruction uses the values in `mnepc` and `mnstatus` to return to the program counter and privileged mode of the interrupted context, respectively. This instruction also sets the internal `rnmie` state bits.

Encoding is same as MRET except with bit 30 set (i.e., `funct7=0111000`). For example:

```
.word 0x70200073 // opcode for MNRET (return from RNMI)
```

7.9.4 RNMI Operation

When an RNMI interrupt is detected, the interrupted PC is written to the `mnepc` CSR, the type of RNMI to the `mncause` CSR, and the privilege mode of the interrupted context to the `mnstatus` CSR. An internal microarchitectural state bit, `rnmie`, is cleared to indicate that the processor is in an RNMI handler and cannot take a new RNMI interrupt. When clear, the internal `rnmie` bit also disables all other interrupts.

Note

These interrupts are called non-maskable because software cannot mask the interrupts. However, for correct operation, other instances of the same interrupt must be held off until the handler is completed, hence the internal state bit.

The RNMI handler can resume original execution using the `MNRET` instruction (described in Section 7.9.3), which restores the PC from `mnepc`, the privilege mode from `mnstatus`, and also sets the internal `rnmie` state bit, which re-enables other interrupts.

If the hart encounters an exception while the `rnmie` bit is clear, the exception state is written to `mepc` and `mcause`, `mstatus.mpp` is set to M-mode, and the hart jumps to the RNMI exception handler address.

Note

Traps in the RNMI handler can only be resumed if they occur while the handler was servicing an interrupt that occurred outside of machine mode.

Chapter 8

Core-Local Interrupt Controller (CLIC)

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

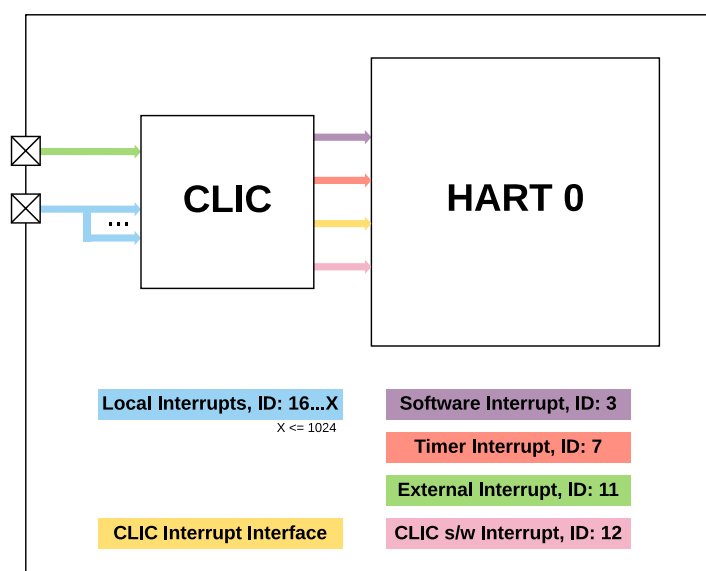


Figure 64: CLIC Block Diagram

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 `clicIntCfg` register and the value of the CLIC's `cliccfg.nlBits` register.

8.2 CLIC Vector Table

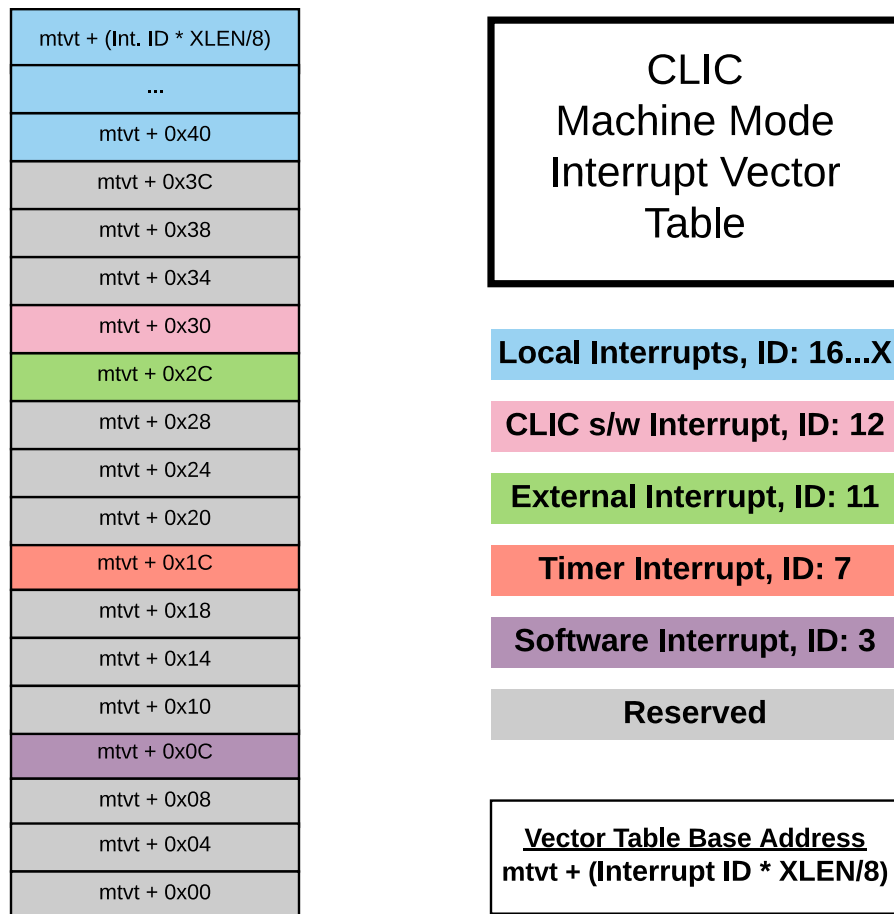


Figure 65: CLIC Interrupts and Vector Table

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 Core Complex 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 Core Complex 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 Core Complex with an external PLIC.

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

CLIC Interrupt IDs are provided in Table 73.

ID	Interrupt Source
0–2	Reserved
3	Machine Software Interrupt (<code>msip</code>)
4–6	Reserved
7	Machine Timer Interrupt (<code>mtip</code>)
8–10	Reserved
11	Machine External Interrupt (<code>meip</code>)
12	CLIC Software Interrupt (<code>csip</code>)
13–15	Reserved
16–47	Local Interrupt 0–31

Table 73: E20 Core Complex 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
}
```

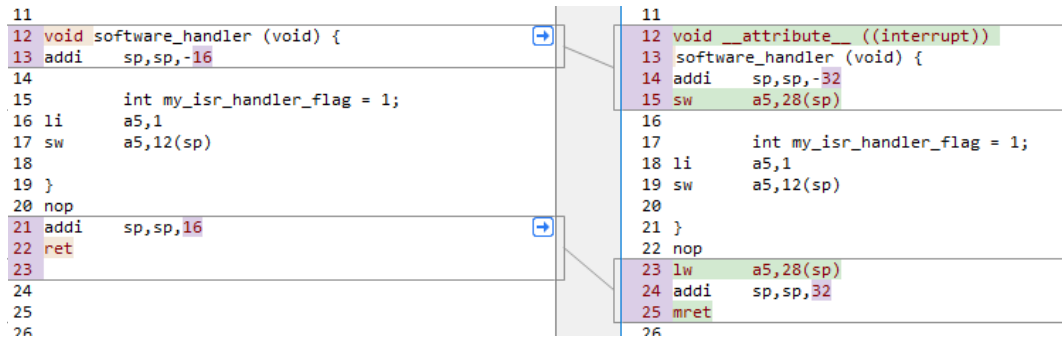


Figure 66: CLIC Interrupt Attribute Example

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 67: 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 74.

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 75.
0x0280_0000	Hart 0	Hart 0 CLIC Base. The specific implementation of this region is described in detail in Table 76.

Table 74: 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	MTIMECMP Register
0x4008			Reserved	
...				
0xBFF7				
0xBFF8	8B	RW	mtime	Timer Register
0xC000			Reserved	
...				
0x7F_EFFF				
0x7F_F000	1B	RW	disableClicClockGateFeature	CLIC global clock gating disable feature (1 bit wide)
0x7F_F001			Reserved	

Table 75: CLIC Shared Region Memory 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 76: CLIC Hart-Specific Region Memory 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.

CSR	Description
mstatus	mstatus.MPP and mstatus.MPIE are accessible via fields in the mcause register.
mie, 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.

Table 77: Changes to CSRs in CLIC Mode

8.7.2 MSIP Register

Machine mode software interrupts are generated by writing to the memory-mapped control register `msip`. The `msip` register is a 32-bit wide **WARL** register, where the upper 31 bits are tied to 0. The least-significant bit is reflected in the MSIP bit of the `mip` CSR. Other bits in the `msip` register are hardwired to zero. On reset, each `msip` register is cleared to zero.

Software interrupts are most useful for interprocessor communication in multi-hart systems, as harts may write each other's `msip` bits to effect interprocessor interrupts.

8.7.3 Timer Registers

`mtime` is a 64-bit read-write register that contains the number of cycles counted from the `rtc_toggle` signal, which is described in the E20 Core Complex User Guide.

A timer interrupt is pending whenever `mtime` is greater than or equal to the value in the `mtimecmp` register. The timer interrupt is reflected in the MTIP bit of the `mip` register, described in Chapter 7.

On reset, `mtime` is cleared to zero. The `mtimecmp` registers are not reset.

8.7.4 CLIC Clock Gate Disable Register (`disableClicClockGateFeature`)

The CLIC implements a clock gating feature to gate the module clock node when not active. CLIC clock gating is disabled out of reset and should be enabled in startup code, unless otherwise specified by SiFive erratum. Once enabled, clock is only available when there is activity on the CLIC control bus, activity on any interrupt line, or an RTCTICK event. Clock gating is further described in the E20 Core Complex User Guide.

CLIC Clock Gate Disable Register (<code>disableClicClockGateFeature</code>)				
Register Address		CLIC Shared Base + 0x7F_F000		
Bits	Field Name	Attr.	Rst.	Description
0	<code>disableClicClockGateFeature</code>	RW	0x1	Used to enable/disable CLIC clock gating feature. Clear to enable.
[7:1]	Reserved	RO	0x0	

Table 78: CLIC Clock Gate Disable Register

8.7.5 CLIC Interrupt Pending Register (clicIntIP)

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.

CLIC Interrupt Pending Register (clicIntIP)				
Register Address		CLIC Hart Base + 1 × Interrupt ID		
Bits	Field Name	Attr.	Rst.	Description
0	clicIntIP	RO*	X	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	0x0	

*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 79: CLIC Interrupt Pending Register (partial)

8.7.6 CLIC Interrupt Enable Register (clicIntIE)

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.

CLIC Interrupt Enable Register (clicIntIE)				
Register Address		CLIC Hart Base + 0x400 + 1 × Interrupt ID		
Bits	Field Name	Attr.	Rst.	Description
0	clicIntIE	RW	X	When clicIntIE is set, the corresponding Interrupt ID is enabled
[7:1]	Reserved	RO	0x0	

Table 80: CLIC Interrupt Enable Register (partial)

8.7.7 CLIC Interrupt Configuration Register (clicIntCfg)

The E20 Core Complex has a total of 2 bits in clicIntCfg that specify how to encode a given interrupt's preemption level, priority, and/or hardware vectoring setting.

CLIC Interrupt Configuration Register (<code>cllcIntCfg</code>)				
Register Address		CLIC Hart Base + 0x800 + 1 × Interrupt ID		
Bits	Field Name	Attr.	Rst.	Description
[5:0]	<code>cllcIntCfgPad</code>	RO	0x3F	Padding of <code>cllcIntCfg</code>
[7:6]	<code>cllcIntCfg</code>	WARL	X	<p><code>cllcIntCfg</code> sets the preemption level and priority of a given interrupt.</p> <p>When selective hardware vectoring is enabled, the least-significant bit is used to control vectoring of a given interrupt.</p> <p>The actual number of bits used to control these settings are determined by <code>cllcCfg</code>.</p>

Table 81: CLIC Interrupt Configuration Register (partial)

8.7.8 CLIC Configuration Register (`cllcCfg`)

The `cllcCfg` register is used to configure the operation of the CLIC, primarily by determining the function of the bits implemented in `cllcIntCfg`; that is, the number of levels and priorities set by `cllcIntCfg`. It also contains the selective hardware vector configuration, which enables CLIC direct mode or vectored mode on a per-interrupt basis.

CLIC Configuration Register (<code>cllcCfg</code>)				
Register Address		CLIC Hart Base + 0xC00		
Bits	Field Name	Attr.	Rst.	Description
0	<code>nvBits</code>	WARL	0x0	When set, selective hardware vectoring is enabled
[4:1]	<code>nlBits</code>	WARL	0x0	Determines the number of Level bits available in <code>cllcIntCfg</code>
[6:5]	<code>nmBits</code>	WARL	0x0	Determines the number of Mode bits available in <code>cllcIntCfg</code>
7	Reserved	WARL	0x0	

Table 82: CLIC Configuration Register

The E20 Core Complex only supports machine mode interrupts, therefore `cllcCfg.nmBits` is set to zero.

`cllcCfg.nlBits` is used to determine the number of `cllcIntCfg` bits used for levels versus priorities. The CLIC supports a maximum of 256 preemption levels, which requires 8 `cllcIntCfg` bits to encode all 256 levels. If `cllcCfg.nlBits` is < 2, then the remaining least-significant implemented bits of `cllcIntCfg` are used to encode priorities within a given preemption level. If `cllcCfg.nlBits` is set to zero, then all interrupts are treated as level 255 and all 2 `cllcIntCfg` bits are used to set priorities.

`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 Core Complex) 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.

For a given design with 5 implemented `clicIntCfg` bits, the encoding of `cliccfg` settings to `clicIntCfg` bits is shown in Table 83.

<code>nmBits</code>	<code>nlBits</code>	<code>nvBits</code>	<code>clicIntCfg</code> Bit Format	Total Modes	Total Levels	Total Priorities	Selective Vectoring
2'b00=0	4'b0000=0	1'b0=0	pppppxxx	1	1	32	Disabled
2'b00=0	4'b0001=1	1'b0=0	lppppxxx	1	2	16	Disabled
2'b00=0	4'b0010=2	1'b0=0	llpppxxx	1	4	8	Disabled
2'b00=0	4'b0011=3	1'b0=0	lllppxxx	1	8	4	Disabled
2'b00=0	4'b0100=4	1'b0=0	llllpxxx	1	16	2	Disabled
2'b00=0	4'b0101=5	1'b0=0	lllllxxx	1	32	1	Disabled
2'b00=0	4'b0010=2	1'b1=1	llppvxxx	1	4	4	Enabled

Note: l=level, p=priority, v=selective vectoring, and x=unimplemented bits.

Table 83: Example Encoding of `cliccfg` Bits to `clicIntCfg`[7:3]

Chapter 9

Power Management

The following chapter describes power modes and establishes flows for powering up, powering down, and resetting the hardware of the E20 Core Complex.

9.1 Power Modes

Power modes include normal run mode with the Power Dial option and wait-for-interrupt clock gating mode using the `wFI` instruction. Additionally, there is a full power down mode supported via the `CEASE` instruction. These modes are covered in detail below.

9.2 Run Mode

The hart is fully operational in run mode, and SiFive designs include the option to include coarse-grained architectural clock gating. When this feature is enabled in the hart, the I-Cache, D-Cache, integer pipeline, Debug Logic, and Floating-Point Unit (FPU) each contain their own clock gate module. The clock gating feature will enable automatic clock gating of functional units when they are inactive and allow the hart to gate its own clock(s) based on activity.

9.2.1 Power Control

To further reduce power while in run mode, users may choose to reduce `external_source_for_core_N_clock`, which is required to be changed synchronously to the rest of the clocks in the system. It is important to note that the clock relationships with the rest of the system must still be maintained if `external_source_for_core_N_clock` is reduced.

To limit maximum power without frequency changes, Power Dial provides a method of scaling down dynamic power in a core. Power is reduced by restricting cycles allowed to advance instructions into the execution pipeline. This feature can typically be used for throttling high CPU usage applications.

Power Dial Register (PowerDia1)			
Register Address		0x7C8	
Bits	Field Name	Rst.	Description
[3:0]	IssueRate	0x0	(1 - value/16) portion of the peak instruction throughput (i.e., value=0 is no reduction)
[31:4]	Reserved	R0	

Table 84: Power Dial Register

The Power Dial Register may only be programmed in M-mode. When written with a non-zero value, the register restricts peak instruction throughput to the indicated rate. A value of 0 has no effect on instruction throughput. The rate is calculated per 256-cycle period, so a Power Dial value of 1 restricts instruction throughput at a common point in the pipeline to 240 cycles of each 256-cycle period. Reducing the peak rate reduces the worst-case power while minimizing the impact on performance.

9.3 WFI Clock Gate Mode

WFI clock gating mode can be entered by executing the `wfi` instruction. The assembly-level instruction is simply `wfi` and executing the C method using the GCC compiler can be accomplished with `asm("wfi")`.

9.3.1 WFI Wake Up

Wake up from a WFI occurs when the hart receives any interrupt. Depending on the software configuration, the hart will either immediately enter the interrupt handler, or resume execution on the instruction immediately after the WFI.

If interrupts are enabled and `mstatus.MIE=1`, then the hart will wake when an interrupt is enabled and becomes pending, and immediately enter the interrupt handler. Upon exit from the interrupt handler, program execution will resume at the instruction following the WFI.

If interrupts are enabled but `mstatus.MIE=0`, then the hart will wake when an interrupt is enabled and becomes pending but will not enter the interrupt handler. It will simply resume at the instruction immediately after the WFI in this case.

To prevent an interrupt source from waking a hart, the enable bit for that interrupt must be written to 0 prior to executing the `wfi` instruction. If any interrupts are pending upon executing a WFI instruction, then the WFI is effectively treated as a NOP instruction.

When the CLIC is operating in CLINT modes of operation (`mtvec.MODE=0` or `mtvec.MODE=1`), any CLIC interrupt source that is enabled and becomes pending still has the ability to wake the hart from a WFI. This includes enable bits in `clicIntIE` and pending bits in `clicIntIP`, in addition to enable bits in `mie` and pending bits in `mip`.

Refer to Chapter 7 for more detail on interrupt configuration.

9.4 CEASE Instruction for Power Down

To fully power down, follow the steps described in Section 9.9, where the last step is to execute a CEASE instruction. Once the CEASE instruction is executed, the core will not retire another instruction until reset. The CEASE opcode is 0x30500073 and can be implemented in either assembly or C. To create an assembly-level function using GCC, consider the following example.

```
.global _cease
.type      _cease, @function
_ceil:
    .word 0x30500073
    ret
```

The next example demonstrates how to implement the CEASE instruction within a function in C.

```
static inline void cease()
{
    __asm__ __volatile__ (".word 0x30500073" : : : "memory"); // CEASE
}
```

9.5 Hardware Reset

The following list summarizes the hardware reset values required by *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* and applies to all SiFive designs.

1. Privilege mode is set to machine mode.
2. `mstatus.MIE` and `mstatus.MPRV` are required to be 0.
3. The `misa` register holds the full set of supported extensions for that implementation, and `misa.MXL` defaults to the widest supported ISA available, referred to as MXLEN.
4. The `pc` is set to the implementation specific reset vector.
5. The `mcause` register is set to 0x0 at reset.
6. The PMP configuration fields for address matching mode (A) and Lock (L) are set to 0, which defaults to no protection for any privilege level.

The internal state of the rest of the system should be completed by software early in the boot flow.

9.6 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.

9.7 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 `cllicIntIE` contents reflect the requirements of the system, and that no unexpected CLIC pending `cllicIntIP` bits are set.

The `cllicIntIP` bits are read-only with the exception of the software interrupt (`cllicIntIP[0]`, bit 3) and the CLIC software interrupt pending (`cllicIntIP[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 `cllicIntIE` and `cllicIntIP` registers.

9.8 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 `mhpmpcounterX`. At any point, the `mhpmpcounterX` 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.

9.9 Power-Down Flow

For SiFive Core IP, coordination with an "External Agent" is required.

1. External Agent: Wait for communication from the core to initiate the following steps:
 - a. Stop sending inbound traffic (both transactions and interrupts) into the Core Complex.
 - b. Wait until all outstanding requests to the Core Complex are completed, then
 - c. Wait until `cease_from_tile_N` is high for the core.
 - d. Once `cease_from_tile_N` is high for the core, apply reset to the entire Core Complex.
2. 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.
 - i. Copy contents of any TIMs/LIMs into external memory.
 - ii. 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 1.0*. Currently, only interactive debug and hardware breakpoints are supported.

10.1 Debug Module

The Debug Module (DM) handles nearly all of the functions related to debugging. It is a slave to both the Debug Module Interface (DMI) coming from the probe, and a TileLink bus coming from the core. From the perspective of the core, the DM appears as a 4K block in the memory map. The DM memory map as seen from the perspective of the core is shown in Table 86, and the register map from the perspective of the DMI is shown in Table 85.

Most of the DM is clocked by `debug_clock`. The `dmcontrol` register is accessible when `debug_clock` is not running, mainly to be able to write to `haltreq` while the core is in reset due to `ndreset`. Doing so generates a debug interrupt and will interrupt the selected core immediately once it is out of reset or during a WFI instruction.

DMI Address	Name	Description
0x11	dmstatus	Debug Module Status. See Table 96 for more information.
0x10	dmcontrol	Debug Module Control. See Table 97 for more information.
0x12	hartinfo	Hart Information. See Table 98 for more information.
0x40	haltsum0	Read-only. Halt Summary 0. Bit n reads 1 if hart n is halted.
0x13	haltsum1	Read-only. Halt Summary 1. Only present on systems with >32 harts. Not used by SiFive.
0x16	abstractcs	Abstract Control and Status. See Table 99 for more information.
0x18	abstractauto	Selects whether access to particular DATA or PROGBUF locations will re-execute the last command. Used for block transfers or other repeating commands. See Table 101 for more information.
0x17	command	Initiate abstract command. See Table 100 for more information.
0x04-0x0F	data0-data11	Read/Write DATA registers. 32-bit SiFive cores have 1 data register, 64-bit cores have 2.
0x20-0x2F	progbuf0-progbuf15	Read/Write PROGBUF registers.
0x32	dmcs2	Fields to set up and read back Halt Group or Resume Group configuration. Present by default on systems with more than 1 hart or with any external triggers. See Table 102 for more information.

Table 85: Debug Module Memory Map Seen from the Debug Module Interface

From the point of view of the core, the DM appears as a 4K block of memory. It is mapped into low memory so that memory references can use addresses relative to the \$zero register.

TL Address	Name	Attr.	Description
0x100	HALTED	WO	Written with hartid by ROM code when hart gets a debug interrupt or reenters ROM due to EBREAK. Sets halted[hartid]. If an abstract command was running, writing this also clears busy.
0x104	GOING	WO	Written by ROM code when it begins executing a command started by FLAGS[hartid].go. Clears FLAGS[hartid].go.
0x108	RESUMING	WO	Written with hartid by hart when it is about to resume. Sets resumeack[hartid] and clears halted[hartid] and FLAGS[hartid].resume.
0x10C	EXCEPTION	WO	Written by hart when it encounters an exception in debug mode. Sets cmderr to "exception".
0x300	WHERE TO	RO	JAL to ABSTRACT. This opcode is constructed by DM hardware and is needed because ABSTRACT is not a fixed address (depends on number of PROGBUF words selected in the configuration).
contiguous	ABSTRACT	RO	2 words constructed by DM hardware based on abstract command written from DTM. +0: If transfer set, construct instruction to load/store specific register to/from DATA[0] (32 bits) or DATA[1:0] (64 bits), else NOP. +4: If postexec set, then NOP to fall thru and execute PROGBUF, else EBREAK to return to ROM park loop.
contiguous	PROGBUF	RW	Configurable number (typically 16, max 16) of R/W words to be filled in by debugger and executed by hart.
contiguous	IMPEBREAK	RO	Reads as EBREAK to return to ROM park loop when execution runs off the end of PROGBUF. In E2, default is 2-word PROGBUF and IMPEBREAK present. Most others have 16-word PROGBUF and no IMPEBREAK.
0x380-0x3BF	DATA	RW	Configurable number (1 for 32-bit or 2 for 64-bit, max 12) of R/W words intended for use for data transfer between debugger and hart. Since it is contiguous with PROGBUF, the debugger may use DATA as an extension of PROGBUF.
0x400-0x7FF	FLAGS	RO	One byte flag per hart. Bit 0 (go): Set by writing an abstract command, cleared by ROM write to GOING. ROM will jump to WHERE TO. Bit 1 (resume): Set by writing 1 to resumereq[hartid]. Cleared by ROM write of hartid to RESUMING. ROM restores s0 then executes dret.

Table 86: Debug Module Memory Map from the Perspective of the Core

TL Address	Name	Attr.	Description
0x800-0xFFFF	ROM	RO	<p>Debug interrupt or EBREAK enters at 0x800, saves s0, writes hartid to HALTED, then busy-waits for <code>FLAGS[hartid] > 0</code>.</p> <p>If <code>FLAGS[hartid].go</code>, write 0 to GOING, then jump to WHERETO.</p> <p>Else write hartid to RESUMING, then execute dret to return to user program.</p> <p>ROM Source Code: https://github.com/chipsalliance/rocket-chip/blob/master/scripts/debug_rom/debug_rom.S</p>

Table 86: Debug Module Memory Map from the Perspective of the Core

10.2 Debug and Trigger Registers

This section describes the per hart debug and trigger registers, which are mapped into the CSR space as follows:

CSR	Name	Allowed Access Mode	Description
0x7B0	dcsr	Debug	Debug Control and Status Register
0x7B1	dpc	Debug	Debug PC. Stores execution address just before debug exception and to return to at dret.
0x7B2	dscratch0	Debug	Debug Scratch Register 0
0x7A0	tselect	Debug, Machine	Trigger Select. Most configs implement 2, 4, or 8 triggers. Triggers are all type 2 (address/data).
0x7A1	tdata1	Debug, Machine	Trigger Data 1, mcontrol
0x7A2	tdata2	Debug, Machine	Trigger Data 2, the address for comparison
0x7A3	tdata3	Debug, Machine	Trigger Data 3

Table 87: Debug and Trigger Registers

10.2.1 Debug Control and Status Register (dcsr)

This register gives information about debug capabilities and status. Its detailed functionality is described in *The RISC-V Debug Specification, Version 1.0*.

Debug Control and Status Register (dcsr)			
CSR	0x7B0		
Bits	Field Name	Attr.	Description
[1:0]	prv	RW	Privilege level of processor prior to debug exception and to return to at dret.
2	step	RW	Set to 0x1 to single-step.
3	nmip	RO	Non-maskable interrupt pending. Not used by SiFive.
4	mprven	WARL	Not used by SiFive.
5	Reserved		
[8:6]	cause	RO	Indicates cause of most recent debug exception.
9	stoptime	WARL	0x1 will stop timers in debug mode. Not used by SiFive (timers continue).
10	stopcount	WARL	0x1 will stop counters in debug mode. Not used by SiFive (counters continue).
11	stepie	WARL	Enable interrupts when stepping. Not used by SiFive (interrupts disabled).
12	ebreaku	RW	EBREAK instructions in U-mode enter debug mode (vs. breakpoint exception).
13	ebreaks	RW	EBREAK instructions in S-mode enter debug mode.
14	Reserved		
15	ebreakm	RW	EBREAK instructions in M-mode enter debug mode.
[27:16]	Reserved		
[31:28]	xdebugver	RO	Version

Table 88: Debug Control and Status Register

10.2.2 Debug PC (dpc)

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

10.2.3 Debug Scratch (dscratch)

This register is generally reserved for use by Debug ROM in order to save registers needed by the code in Debug ROM. The debugger may use it as described in *The RISC-V Debug Specification, Version 1.0*.

10.2.4 Trigger Select Register (tselect)

To support a large and variable number of triggers for tracing and breakpoints, they are accessed through one level of indirection where the tselect register selects which bank of three tdata1-3 registers are accessed via the other three addresses.

The tselect register has the format shown below:

Trigger Select Register (tselect)			
CSR	0x7A0		
Bits	Field Name	Attr.	Description
[31:0]	index	WARL	Selection index of triggers

Table 89: Trigger Select Register

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

10.2.5 Trigger Data Registers (tdata1-3)

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

Trigger Data Register 1 (tdata1)			
CSR	0x7A1		
Bits	Field Name	Attr.	Description
[26:0]	Trigger-Specific Data		
27	dmode	WARL	Selects between debug mode (dmode=1) and machine mode (dmode=0) views of the registers, where only debug mode code can access the debug mode view of the triggers
[31:28]	type	WARL	The type of trigger selected by tselect <ul style="list-style-type: none"> • 0x0 - No such trigger • 0x1 - Reserved • 0x2 - Address/Data Match Trigger • ≥0x3 - Reserved

Table 90: Trigger Data Register 1

Trigger Data Registers 2 and 3 (tdata2/3)			
CSR	0x7A2 - 0x7A3		
Bits	Field Name	Attr.	Description
[31:0]	Trigger-Specific Data		

Table 91: Trigger Data Registers 2 and 3

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

10.3 Breakpoints

The E20 Core Complex 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
tselect	tselect	Breakpoint selection index
tdata1	mcontrol	Breakpoint match control
tdata2	maddress	Breakpoint match address
tdata3	N/A	Reserved

Table 92: Trigger CSRs When Used as Breakpoints

10.3.1 Breakpoint Match Control Register (mcontrol)

Each breakpoint control register is a read/write register laid out in Table 93. This register is accessible as tdata1 when type is 0x2.

Breakpoint Match Control Register (mcontrol1)				
CSR	0x7A1			
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	
6	M	WARL	X	Address match on machine mode
[10:7]	match	WARL	X	Breakpoint Address Match type <ul style="list-style-type: none"> • 0x0 - Single address • 0x1 - Power-of-2 range, limited to 64 bytes in SiFive implementations • 0x2 - ≥ address • 0x3 - < address • Others not supported by SiFive
11	chain	WARL	0x0	Chain adjacent conditions. When set, this trigger and the next must match at the same time to fire. Typically used for a range breakpoint using 2 triggers, one with match=0x2 and one with match=0x3. This is not a sequential trigger.
[15:12]	action	WARL	0x0	Breakpoint action to take
[17:16]	size0	WARL	0x0	Size of the breakpoint. Fixed at 0, meaning accesses of any size that cover any part of the trigger address range will fire.
18	timing	WARL	0x0	Timing of the breakpoint. Fixed at 0, meaning breaks happen just before the event.
19	select	WARL	0x0	Perform match on address or data. Fixed at 0, meaning all triggers compare addresses only (no data value).
20	Reserved	WPRI	X	
[26:21]	maskmax	RO	0x4	Largest supported NAPOT range
27	dmode	RW	0x0	Debug-Only access mode
[31:28]	type	RO	0x2	Address/Data match type, always 0x2

Table 93: Breakpoint Match Control Register

The type field is a 4-bit read-only field holding the value 0x2 to indicate this is a breakpoint containing address match logic.

The action field is a 4-bit read-write **WARL** field that specifies the available actions when the address match is successful. The value 0 generates a breakpoint exception. The value 1 enters debug mode. Other actions are not implemented.

The R/W/X bits are individual **WARL** fields, and if set, indicate an address match should only be successful for loads, stores, and instruction fetches, respectively. All combinations of implemented bits must be supported.

The M/S/U bits are individual **WARL** fields, and if set, indicate that an address match should only be successful in the machine, supervisor, and user modes, respectively. All combinations of implemented bits must be supported.

The match field is a 4-bit read-write **WARL** field that encodes the type of address range for breakpoint address matching. Three different match settings are currently supported: exact, NAPOT, and arbitrary range. A single breakpoint register supports both exact address matches and matches with address ranges that are naturally aligned powers-of-two (NAPOT) in size. Breakpoint registers can be paired to specify arbitrary exact ranges, with the lower-numbered breakpoint register giving the byte address at the bottom of the range and the higher-numbered breakpoint register giving the address 1 byte above the breakpoint range and using the chain bit to indicate both must match for the action to be taken.

NAPOT ranges make use of low-order bits of the associated breakpoint address register to encode the size of the range as follows:

address	Match Type and Size
a...aaaaaa	Exactly 1 byte
a...aaaaa0	2-byte NAPOT range
a...aaaa01	4-byte NAPOT range
a...aaa011	8-byte NAPOT range
a...aa0111	16-byte NAPOT range
a...a01111	32-byte NAPOT range
	...
a01...1111	2^{31} -byte NAPOT range

Table 94: 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.

10.3.2 Breakpoint Match Address Register (`maddress`)

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.

10.3.3 Breakpoint Execution

Breakpoint traps are taken precisely. Implementations that emulate misaligned accesses in software will generate a breakpoint trap when either half of the emulated access falls within the address range. Implementations that support misaligned accesses in hardware must trap if any byte of an access falls within the matching range.

Debug mode breakpoint traps jump to the debug trap vector without altering machine mode registers.

Machine mode breakpoint traps jump to the exception vector with "Breakpoint" set in the `mcause` register and with `badaddr` holding the instruction or data address that caused the trap.

10.3.4 Sharing Breakpoints Between Debug and Machine Mode

When debug mode uses a breakpoint register, it is no longer visible to machine mode (that is, the `tdrtype` will be 0). Typically, a debugger will leave the breakpoints alone until it needs them, either because a user explicitly requested one or because the user is debugging code in ROM.

10.4 Debug Memory Map

This section describes the Debug Module's memory map when accessed via the regular system interconnect. The Debug Module is only accessible to debug code running in debug mode on a hart (or via a Debug Transport Module). The following addresses are offsets from the base address of the Debug Module. Note that the PMP must allow M-mode access to the Debug Module address range for debugging to be possible.

10.4.1 Debug RAM and Program Buffer (0x300–0x3FF)

The E20 Core Complex 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 Core Complex has one 32-bit words of debug data RAM. Its location can be determined by reading the `DM.hartinfo` register, as described in *The RISC-V Debug Specification, Version 1.0*. This RAM space is used to pass data for the Access Register abstract command, as described in *The RISC-V Debug Specification, Version 1.0*. The E20 Core Complex supports only general-purpose register access when harts are halted. All other commands must be implemented by executing from the debug program buffer.

In the E20 Core Complex, both the program buffer and debug data RAM are general-purpose RAM and are mapped contiguously in the Core Complex memory space. Therefore, additional data can be passed in the program buffer, and additional instructions can be stored in the debug data RAM.

Debuggers must not execute program buffer programs that access any Debug Module memory except defined program buffer and debug data addresses.

10.4.2 Debug ROM (0x800–0xFFF)

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

10.4.3 Debug Flags (0x100–0x110, 0x400–0x7FF)

The flag registers in the Debug Module are used for the Debug Module to communicate with each hart. These flags are set and read used by the debug ROM and should not be accessed by any program buffer code. The specific behavior of the flags is not further documented here.

10.4.4 Safe Address

In the E20 Core Complex, the Debug Module contains the Debug Module address range in the memory map. Memory accesses to these addresses raise access exceptions, unless the hart is in debug mode. This property allows a "safe" location for unprogrammed parts, as the default `mtvec` location is `0x0`.

10.5 Debug Module Interface

The SiFive Debug Module (DM) conforms to *The RISC-V Debug Specification, Version 1.0*. A debug probe or agent connects to the Debug Module through the Debug Module Interface (DMI). The following sections describe notable spec options used in the implementation and should be read in conjunction with *The RISC-V Debug Specification, Version 1.0*.

DMI is a simple read/write bus whose master is the DTM (if it exists, otherwise DMI passes through to customer logic) and whose slave is the Debug Module. The master sends a request to the slave and the slave responds with a response. A request is considered sent if `req_ready=1` indicating the master is sending a request and `req_valid=1` indicating the slave is accepting the request on this cycle. Similarly, the response is sent when both `resp_valid=1`

indicating the slave is sending a response and resp_ready=1 indicating the master is accepting it.

Note

It is the responsibility of the debugger to simulate virtual address accesses by accessing the page tables directly, then sending the translated physical address to hardware when doing the access.

Note

The Debug Module registers are not directly accessible from the core.

Group	Signal	Source	Description
System	clock	system	All signals timed to this clock. With JTAG DTM, this clock is the JTAG TCK.
	reset	system	Synchronous reset. Generated by power-on reset circuit.
Request Bus	req_ready	slave	Slave ready to receive request.
	req_valid	master	Master's request valid.
	req_addr	master	Configurable width address bus. 0x7 for SiFive.
	req_data	master	32-bit write data bus.
	req_op	master	<ul style="list-style-type: none"> • 0x0 = None • 0x1 = Read • 0x2 = Write • 0x3 = Reserved
Response Bus	resp_ready	master	Master is ready to receive response.
	resp_valid	slave	Slave response is valid.
	resp_data	slave	32-bit read data bus.
	resp_op	slave	<ul style="list-style-type: none"> • 0x0 = Success • 0x1 = Failure • 0x2 = Not used • 0x3 = Reserved

Table 95: Debug Module Interface Signals

10.5.1 Debug Module Status Register (dmstatus)

dmstatus holds the DM version number and other implementation information. Most importantly, it contains status bits that indicate the current state of the selected hart(s).

Debug Module Status Register (dmstatus)				
DMI Address		0x11		
Bits	Field Name	Attr.	Rst.	Description
[3:0]	version	RO	0x3	Implementation version number
4	Reserved	RO	0x0	
5	hasresethaltreq	RO	0x1	1 if resethaltreq exists
[7:6]	Reserved	RO	0x0	
8	anyhalted	RO	0x0	Any currently selected hart is halted
9	allhalted	RO	0x0	All currently selected harts are halted
10	anyrunning	RO	0x1	Any currently selected hart is running
11	allrunning	RO	0x1	All currently selected harts are running
12	anyunavail	RO	0x0	Any currently selected hart is not available (i.e., is powered down). DM supports it, but not currently used by SiFive cores.
13	allunavail	RO	0x0	All currently selected harts are not available (i.e., is powered down). DM supports it, but not currently used by SiFive cores.
14	anynonexistent	RO	0x0	Any currently selected hart does not exist in the system
15	allnnonexistent	RO	0x0	All currently selected harts do not exist in the system
16	anyresumeack	RO	0x1	Any currently selected hart has resumed execution
17	allresumeack	RO	0x1	All currently selected harts have resumed execution
18	anyhavereset	RO	0x0	Any currently selected hart has been reset, but reset has not been acknowledged
19	allhavereset	RO	0x0	All currently selected harts have been reset, but reset has not been acknowledged
[21:20]	Reserved	RO	0x0	
22	impebreak	RO	0x1	1 if PROGBUF is followed by implicit EBREAK. Generally, 1 for E2 cores, 0 otherwise.
[31:23]	Reserved	RO	0x0	

Table 96: Debug Module Status Register

10.5.2 Debug Module Control Register (dmcontrol)

A debugger performs most hart controls through the `dmcontrol` register.

Debug Module Control Register (dmcontrol)				
DMI Address		0x10		
Bits	Field Name	Attr.	Rst.	Description
0	dmactive	RW	0x0	0 disables the DM and sets DMI registers to their reset state, 1 puts the DM in operational mode. Drives dmactive output that could be used by a system power controller to maintain power to the DM while it is being used. When 1, dmcontrol should be read back until dmactive=1, which indicates that the Debug Module is fully operational. When 0, the DM TileLink clock is gated off to save power.
1	ndmreset	RW	0x0	Write 1 to reset system (assert ndreset output). Write 0 to operate normally.
2	clrresethaltreq	WO	0x0	Write 1 to clear the reset-halt-request bit
3	setresethaltreq	WO	0x0	When written to 1, the core will halt upon the next deassertion of its reset
[27:4]	Reserved	RW	0x0	
28	ackhavereset	WO	0x0	Write 1 to acknowledge that a reset occurred on the selected hart
29	Reserved	RO	0x0	
30	resumereq	WO	0x0	Write 1 to request selected hart to resume, cleared to 0 automatically when hart resumes
31	haltreq	RW	0x0	Write 1 to request selected hart to halt. Generates debug interrupt to the core. Write 0 once halted has been set by the DM.

Table 97: Debug Module Control Register

10.5.3 Hart Info Register (hartinfo)

hartinfo contains information about the currently selected hart.

Hart Info Register (hart.info)				
DMI Address		0x12		
Bits	Field Name	Attr.	Rst.	Description
[11:0]	dataaddr	RO	0x380	Address of DATA registers in hart memory map. 0x380 for SiFive.
[15:12]	datasize	RO	0x1	Number of DATA registers. 0x1 for 32-bit, 0x2 for 64-bit SiFive cores.
16	dataaccess	RO	0x1	DATA registers are shadowed in the hart memory map. 1 for SiFive.
[19:17]	Reserved	RO	0x0	
[23:20]	nscratch	RO	0x1	Number of dscratch registers available for debugger. 1 for SiFive.
[31:24]	Reserved	RO	0x0	

Table 98: Hart Info Register

10.5.4 Abstract Control and Status Register (abstractcs)

Abstract Control and Status Register (abstractcs)				
DMI Address		0x16		
Bits	Field Name	Attr.	Rst.	Description
[3:0]	datacount	RO	0x1	Number of DATA registers. 0x1 for 32-bit, 0x2 for 64-bit SiFive cores.
[7:4]	Reserved	RO	0x0	
[10:8]	cmderr	RW1C	0x0	<p>Non-zero value indicates an abstract command error. Remains set until cleared by writing all ones. If set, no abstract commands are accepted.</p> <ul style="list-style-type: none"> • 0x0 - No error • 0x1 - Busy. Abstract command or register was accessed while command was running. • 0x2 - Not supported. Abstract command type not supported by hardware was attempted. • 0x3 - Exception. An exception occurred during execution of an abstract command. • 0x4 - Halt/resume. Abstract command attempted while hart was running or unavailable. • 0x5 - Bus. Bus error occurred during abstract command. Not used by SiFive. • 0x7 - Other. Abstract command failed for another reason. Not used by SiFive.
11	Reserved	RO	0x0	
12	busy	RO	0x0	Reads as 1 while Abstract command is running, 0 if not.
[23:13]	Reserved	RO	0x0	
[28:24]	progbufsize	RO	0x2	Number of 32-bit words in PROGBUF. E20 Core Complex has 2 words.
[31:29]	Reserved	RO	0x0	

Table 99: Abstract Control and Status Register

10.5.5 Abstract Command Register (command)

Abstract Command Register (command)			
DMI Address		0x17	
Bits	Field Name	Attr.	Description
[15:0]	regno	RW	Select which register to read/write. SiFive only supports GPRs: 0x1000-0x101F.
16	write	RW	1=write register, 0=read register. Only done if transfer=1.
17	transfer	RW	1=do the register read/write, 0=don't.
18	postexec	RW	1=execute PROGBUF after the command, 0=don't.
19	aarpostincrement	RW	Not supported by SiFive.
[22:20]	aarsize	RW	0x2, 0x3, 0x4 select 32, 64, 128 bits, respectively.
23	Reserved	RO	0x0
	[31:24]	cmdtype	RW

Table 100: Abstract Command Register

10.5.6 Abstract Command Autoexec Register (abstractauto)

Abstract Command Autoexec Register (abstractauto)				
DMI Address		0x18		
Bits	Field Name	Attr.	Rst.	Description
[11:0]	autoexecdata	RW	0x0	Bitmap of DATA registers [11:0]. 1 indicates DATA access initiates command.
[15:12]	Reserved	RO	0x0	
[31:16]	autoexecprogbuf	RW	0x0	Bitmap of PROGBUF words [15:0]. 1 indicates PROGBUF access initiates command.

Table 101: Abstract Command Autoexec Register

10.5.7 Debug Module Control and Status 2 Register (dmcs2)

Table 102 describes the Debug Module Control and Status 2 Register dmcs2. If halt/resume groups are not implemented, then group will always read back as 0. The Debug Module external triggers may be allocated as needed between halt and resume groups.

Debug Module Control and Status 2 Register (dmcs2)				
DMI Address		0x32		
Bits	Field Name	Attr.	Rst.	Description
0	hgselect	RW	0x0	0=operate on harts, 1=operate on external triggers.
1	hgwrite	WO	X	When written with 1, the selected harts or external trigger is assigned to group group.
[6:2]	group	RW	0x0	Specify the halt group or resume group number that the selected harts or external triggers will be assigned to.
[10:7]	Reserved	RO	0x0	
11	grouptype	RW	0x0	0=operate on Halt Group configuration, 1=operate on Resume Group configuration.
[31:12]	Reserved	RO	0x0	

Table 102: Debug Module Control and Status 2 Register

10.5.8 Abstract Commands

Abstract commands provide a debugger with a path to read and write processor state and are used for extracting and modifying processor state such as registers and memory. Register `s0` is saved by the ROM and is available for use by the abstract command code. An abstract command is started by the debugger writing to `command`. In `command`, the debugger selects whether to load/store a register, execute `PROGBUF`, or both. Only GPR register transfers are supported currently. Many aspects of Abstract Commands are optional in *The RISC-V Debug Specification, Version 1.0* and are implemented as described below.

cmdtype	Feature	Support
Access Register	GPR registers	Access Register command, register number 0x1000 - 0x101F
	CSR registers	Not supported. CSRs are accessed using the Program Buffer.
	FPU registers	Not supported. FPU registers are accessed using the Program Buffer.
	Autoexec	Both <code>autoexecprogbuf</code> and <code>autoexecdata</code> 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 103: Debug Abstract Commands

The use of abstract commands is outlined in the following example, describing how to read a word of target memory:

1. The debugger writes opcodes to PROGBUF to accomplish the desired function.
2. The debugger writes the desired memory address to DATA[0].
3. The debugger requests an abstract command specifying to load s0 from DATA[0], then execute PROGBUF. Writing to command while hart n is selected has the side effect of setting FLAGS[n].go. Writing to command also sets busy which is readable from the debugger, and indicates that an abstract command is in progress.
4. The ROM busy-wait loop being executed by hart n sees FLAGS[n].go set.
5. ROM code writes 0 to GOING which has the effect of clearing FLAGS[n].go.
6. ROM code jumps to WHERETO, then ABSTRACT which contains the opcode lw s0, 0(DATA) to load s0 from DATA[0]. Opcodes in ABSTRACT are constructed by DM hardware from command. If command.transfer=0, no register transfer is done and instead ABSTRACT[0] reads as NOP.
7. If a register read/write is all that is needed, the debugger would set command.postexec to 0. ABSTRACT[1] would then read as EBREAK.
8. If command.postexec=1, ABSTRACT[1] reads as NOP and execution falls through to PROGBUF which will have been previously written by the debugger with the opcodes lw s0, 0(s0), then sw s0, DATA(zero), then EBREAK.
9. EBREAK reenters ROM at address 0x800. ROM writes hartid to HALTED which has the side effect of clearing busy, telling the debugger that the abstract command is finished.
10. The debugger reads the result from DATA[0].

The autoexec feature of Abstract Commands is supported by SiFive hardware (and is used by OpenOCD for memory block read and write). Once an abstract command has been completed, the debugger can read or write a particular DATA or PROGBUF location to run the command again. For example, fast download can be accomplished by setting up PROGBUF for memory write, then repeatedly writing words to DATA[0]. Each write re-executes the register transfer and PROGBUF to store the word into memory. For a 32-bit block write, the abstract command would be set up like this:

ABSTRACT	regno=s1, write=1, transfer=1, postexec=1. DM constructs the instructions lw s1,0(DATA) // load s1 from debugger NOP // fall thru to PROGBUF
PROGBUF	sw s1, 0(s0) // store s1 to memory addi s0, s0, 4 // increment memory pointer ebreak // done

Table 104: Abstract Command Example for 32-bit Block Write

10.6 Debug Module Operational Sequences

The sections below describe the flow for entering into and exiting from debug mode. The user can halt and resume more than one hart at a time using the hart array mask.

10.6.1 Entering Debug Mode

To use debug mode, the DM must be enabled by writing `0x0000_0001` to `dmcontrol`.

The debugger can request a halt by writing `0x8000_0001` to `dmcontrol` to set `haltreq`. This generates a debug interrupt to the core.

The core enters debug mode and jumps to the debug interrupt handler located at `0x800` and serviced from the DM.

ROM code at `0x800` writes `hartid` into the `HALTED` register which has the effect of setting the halted bit for this hart. Halted bits are readable from the debugger and generally will be continually polled to check for breakpoints when a hart is running.

ROM code then busy-waits checking its hart-specific `FLAGS` register.

10.6.2 Exiting Debug Mode

The debugger writes `1` to `resumereq` in the `dmcontrol` register to restart execution. This clears `resumeack` and sets bit `1` of the `FLAGS` register for the selected hart.

The ROM busy-wait loop being executed by hart `n` sees `FLAGS[n].resume` set.

ROM code writes `hartid` to `RESUMING`, which has the effect of clearing `FLAGS[n].resume`, setting `resumeack`, and clearing `halted` for the hart.

ROM code then executes `dret` which returns to user code at the address currently in `dpc`.

The debugger sees `resumeack` and knows the resume was successful.

Appendix A

SiFive Core Complex Configuration Options

This section provides a reference of the key configuration options of the SiFive E2 Series cores and the larger Core Complex. The file `docs/core_complex_configuration.txt` lists the features and options configured in the E20 Core Complex.

A.1 E2 Series

The E2 Series comes with the following set of configuration options. Note that the configuration may be limited to a fixed set of discrete options.

Modes and ISA:

- Optional support for RISC-V user mode
- Optional M, A, F, D, B, and Zfh extensions
 - If M extension, configurable 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) with the following options:
 - Configurable size (4 KiB to 8 MiB) and base address
 - Optional AMO support
 - Configurable pipeline depth (0, 1, or 3 additional stages)
 - Configurable number of banks (1 to 64)
- Optional μ Instruction Cache with configurable size (up to 16 KiB) and line size (32 B or 64 B)

- Optional Address Remapper with the following options:
 - Configurable number of entries (4, 8, 16, 32, or 64)
 - "From" region with configurable size (4 to 4294967296) and base address
 - "To" region with configurable size (4 to 4294967296) and base address
 - Configurable maximum remap region size (4 to 4294967296)

Error Handling:

- Optional Bus-Error Unit (BEU)
- Optional ECC support

Ports:

- Optional second System Port, Peripheral Port, and Front Port
 - Each port has a configurable base address, size, and protocol (AHB, AHB-Lite, APB, or AXI4)
 - If AXI4 protocol, configurable AXI ID width (4, 8, or 16). Front, Memory, and System Ports only.

Security:

- Optional Physical Memory Protection (PMP), configurable up to 16 regions
- Optional Disable Debug Input
- Optional Password-protected Debug
- Optional Hardware Cryptographic Accelerator (HCA) with the following options:
 - Configurable base address
 - Optional AES-128/192/256
 - Optional AES-MAC
 - Optional SHA-224/256/384/512
 - Optional True Random Number Generator (TRNG)
 - Optional Public Key Accelerator (PKA) with the following parameters:
 - Configurable PKA operation maximum width (256- or 384-bits)

SiFive Insight Debug and Trace:

- Optional Debug Module with the following options:
 - Configurable base address

- Configurable debug interface (JTAG, cJTAG, APB)
- Configurable 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:
 - Configurable Trace Encoder Format (BTM or HTM)
 - Trace Sink (SRAM, ATB Bridge, SWT, System Memory, and/or PIB)
 - If SRAM Sink, configurable Trace Buffer size (256 B to 64 KiB)
 - If PIB Sink, configurable width (1-, 2-, 3-, 5-, or 9-bit) and optional PIB clock input
 - Optional Timestamp capabilities with configurable width (40, 48, or 56 bits) and source (Bus Clock, Core Clock, or External)
 - External Trigger Inputs (0 to 8) and Outputs (0 to 8)
 - Optional Instrumentation Trace Component (ITC)
 - Optional PC Sampling

Interrupts:

- Optional 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:

- Configurable SRAM user-defined inputs (0 to 1024)
- Configurable SRAM user-defined outputs (0 to 1024)
- Optional SRAM Macro Extraction
- Optional Clock Gate Extraction
- Optional Grouping and Wrapping of extracted macros

Note that the SRAM user-defined feature is mutually exclusive to the macro extraction features.

Clocks and Reset:

- Optional Clock Gating
- Optional Separate Reset for Core and Uncore
- Configurable Reset Scheme (Synchronous, Asynchronous, Full Asynchronous with separate GPR reset)

RTL Options:

- Optional custom RTL module name prefix

Appendix B

SiFive RISC-V Implementation Registers

This section provides a reference to the SiFive RISC-V implementation version registers `marchid` and `mimpid`.

B.1 Machine Architecture ID Register (`marchid`)

Value	Core Generator
0x8000_0002	E2/S2-Series Processor

Table 105: Core Generator Encoding of `marchid`

B.2 Machine Implementation ID Register (mimpid)

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

Table 106: Generator Release Encoding of mimpid

Appendix C

Revision History

This section describes the changes in this document between release versions.

Version	Date	Document Changes
21G2.01.00	July 21, 2021	<ul style="list-style-type: none">• Initial release

Table 107: E20 Core Complex Manual Revision History

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), Version 20180831, August 2018. [Online]. Available: <https://github.com/riscv/riscv-fast-interrupt>

[5] E. Edgar and T. Newsome, Eds., RISC-V Debug Support, Version 1.0, May 2021. [Online]. Available: <https://github.com/riscv/riscv-debug-spec>

[6] C. Wolf, Ed., RISC-V Bitmanip Extension, Version 0.94, December 2020. [Online]. Available: <https://github.com/riscv/riscv-bitmanip>