SiFive E20 Core Complex Manual

v19.02

SiFive E20 Core Complex Manual

Proprietary Notice

Release Information

Version	Date	Changes
v19.02	February 28, 2019	• Changed the date based release numbering system • Top-level module name [E20_CoreIPSubsystem] • SiFive Insight [enabled] • External MEIP interrupt [distinguished from local CLIC interrupts]
v1p0	June 29, 2018	• Updates to support the intitial release of the E20 Standard Core • Interrupts chapter now supports CLIC modes and CSRs • Added CLIC chapter (removed PLIC and CLINT chapters)
v0p1	February 28, 2018	• Initial release • Describes the functionality of the SiFive E20 Core Complex

Contents

Chapter 1

Introduction

SiFive's E20 Core Complex is an efficient implementation of the RISC-V RV32IMAC 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.
E20 Core	1× E20 RISC-V core.
Local Interrupts	32 Local Interrupt signals per hart which can be connected to off core complex devices.
Hardware Breakpoints	4 hardware breakpoints.
Physical Memory Protection Unit	PMP with 0 x regions and a minimum granularity of 0 bytes.

Table 1: E20 Core Complex Feature Set

1.1 E20 Core Complex Overview

An overview of the SiFive E20 Core Complex is shown in Figure 1. This RISC-V Core IP includes 1 x 32-bit RISC-V core, including local and global interrupt support, and physical memory protection. The E20 Core Complex also includes a debug unit, and one outgoing Port.

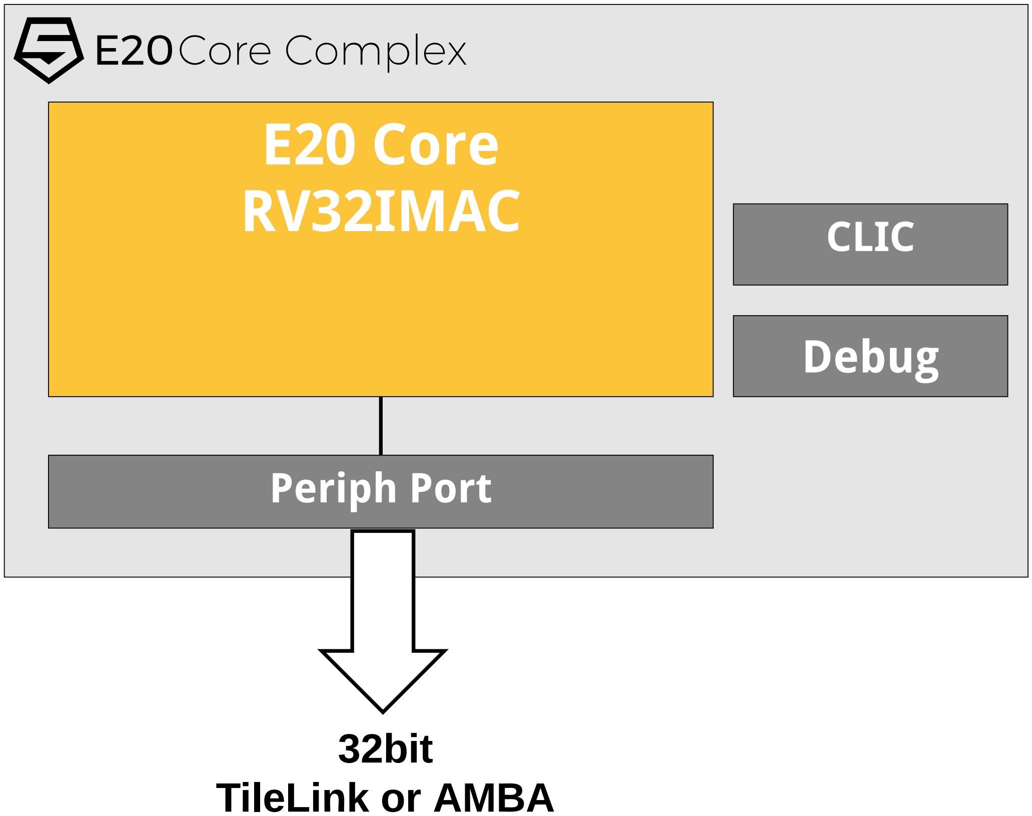

Figure 1: E20 Core Complex Block Diagram

The E20 Core Complex memory map is detailed in Chapter 4, and the interfaces are described in full in the E20 Core Complex User Guide.

1.2 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 7, and the debug interface is described in the E20 Core Complex User Guide.

1.3 Interrupts

The E20 Core Complex supports 32 high-priority, low-latency local vectored interrupts per-hart.

Interrupts are described in Chapter 5. The CLIC is described in Chapter 6.

Chapter 2

List of Abbreviations and Terms

Term	Definition
BHT	Branch History Table
BTB	Branch Target Buffer
RAS	Return-Address Stack
CLINT	Core-Local Interruptor. Generates per-hart software interrupts and timer interrupts.
CLIC	Core-Local Interrupt Controller. Configures priorities and levels for core local interrupts.
hart	HARdware Thread
DTIM	Data Tightly Integrated Memory
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 but not tightly integrated to a CPU core.
PMP	Physical Memory Protection
PLIC	Platform-Level Interrupt Controller. The global interrupt controller in a RISC-V system.
TileLink	A free and open interconnect standard originally developed at UC Berkeley.
RO	Used to describe a Read Only register field.
RW	Used to describe a Read/Write register field.
WO	Used to describe a Write Only registers 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.
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.

Chapter 3

E20 RISC-V Core

This chapter describes the 32-bit E20 RISC-V processor core. The processor core comprises a decoupled instruction-fetch unit and a latency-optimized instruction-execution pipeline.

3.1 Instruction-Fetch Unit

The E20 instruction-fetch unit issues TileLink bus requests to supply instructions to the execution pipeline. When executing straight-line code from an instruction source with single-cycle latency, the instruction-fetch unit can supply instructions at a sustained rate of one per cycle, regardless of instruction alignment.

The instruction-fetch unit issues only aligned 32-bit TileLink requests. If the execution pipeline consumes only part of the 32-bit instruction-fetch parcel, the instruction-fetch unit queues the remainder. Hence, when executing programs comprised mostly of compressed 16-bit instructions, the instruction-fetch unit is often idle, thereby reducing bus occupancy and power consumption.

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, provided the target is naturally aligned (i.e., if the target is any 16-bit instruction, or is a 32-bit instruction whose address is divisible by 4). Taken branches and unconditional jumps to misaligned targets incur an additional one-cycle penalty.

3.2 Execution Pipeline

The E20 execution unit is a single-issue, in-order pipeline. The pipeline comprises two stages: Instruction Fetch, described in the previous section, and Execute.

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

- The number of stall cycles between a load instruction and the use of its result is equal to the number of cycles between the bus request and bus response. In particular, if a load is satisfied the cycle after it is demanded, then there is one stall cycle between the load and its use. In this special case, the stall can be obviated by scheduling an independent instruction between the load and its use.
- Integer multiplication instructions have a latency of 4 cycles. Note that in-flight multiplication operations can be interrupted, so they have no effect on worst-case interrupt latency.
- Integer division instructions have variable latency of at most 32 cycles. Note that in-flight division operations can be interrupted, so they have no effect on worst-case interrupt latency.

The E20 pipeline operates as follows. In the Execute stage, 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 TileLink bus. Exceptions and interrupts are also detected in this stage: exceptional instructions do not proceed.

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

3.3 Data Memory System

The E20 processor has a TileLink bus interface used by all loads, stores, and AMOs. Only one data memory access is permitted to be in flight.

Store instructions incur no stalls if acknowledged by the TileLink bus on the cycle after they are sent. Otherwise, the pipeline will interlock on the next memory-access instruction until the store is acknowledged.

3.4 Local Interrupts

The E20 supports up to 32 local interrupt sources that are routed directly to the core. See Chapter 5 for a detailed description of Local Interrupts.

3.5 Supported Modes

The E20 supports RISC-V machine mode only.

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

3.6 Hardware Performance Monitor

The E20 Core Complex supports a basic hardware performance monitoring facility compliant with *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. The `mcycle` CSR holds a count of the number of clock cycles the hart has executed since some arbitrary time in the past. The `minstret` CSR holds a count of the number of instructions the hart has retired since some arbitrary time in the past. Both are 64-bit counters. The `mcycle` and `minstret` CSRs hold the 32 least-significant bits of the corresponding counter, and the `mcycleh` and `minstreth` CSRs hold the most-significant 32 bits.

Chapter 4

Memory Map

The memory map of the E20 Core Complex is shown in Table 2.

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX	Debug	Debug Address Space
0x0000_1000	0x01FF_FFFF		Reserved	
0x0200_0000	0x02FF_FFFF	RW	CLIC	On Core Complex Devices
0x0300_0000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	RWX	System Port (512 MiB)	Off Core Complex Address Space for External I/O
0x4000_0000	0x7FFF_FFFF		Reserved	

Table 2: E20 Core Complex Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

Chapter 5

Interrupts

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

Specifically the E20 Core Complex implements 20180709 of the RISC-V Core-Local Interrupt Controller (CLIC) specification. The CLIC represents a new RISC-V interrupt specification which differs from the *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. As of June 2018, the CLIC is currently a RISC-V draft proposal of the RISC-V foundation's Fast Interrupts Task Group. Future versions of this core may implement later versions of the CLIC specification.

5.1 Interrupt Concepts

The E20 Core Complex supports Machine Mode interrupts. It also has support for the following types of RISC-V interrupts: local and global.

Local interrupts are signaled directly to an individual hart with a dedicated interrupt value. This allows for reduced interrupt latency as no arbitration is required to determine which hart will service a given request and no additional memory accesses are required to determine the cause of the interrupt.

The E20 Core Complex has 32 interrupts which are delivered to the core via the Core-Local Interrupt Controller (CLIC) along with the software and timer interrupts.

Global interrupts, by contrast, 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(s) 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.

This chapter describes the E20 Core Complex interrupt architecture.

Chapter 6 describes the Core-Local Interrupt Controller (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.

The E20 Core Complex interrupt architecture is depicted in Figure 2.

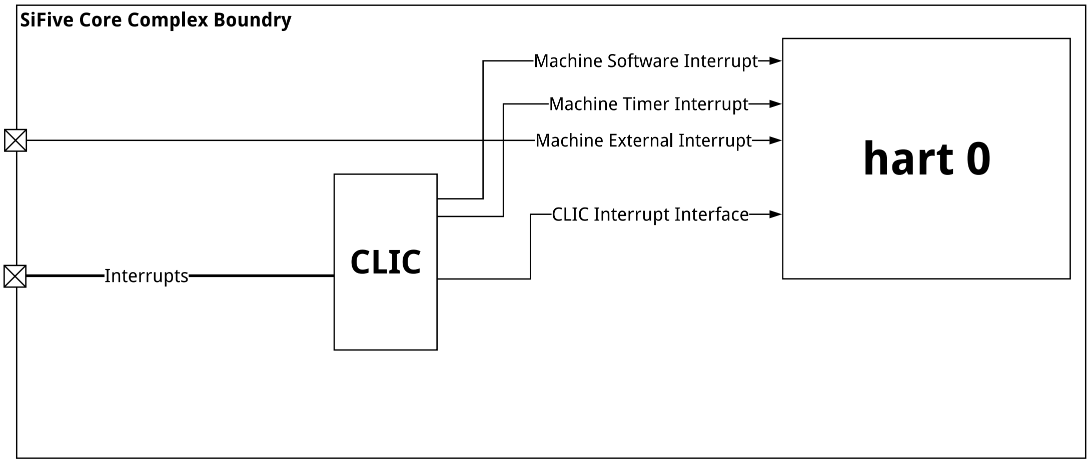

Figure 2: E20 Core Complex Interrupt Architecture Block Diagram.

5.2 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.

5.2.1 Interrupt Entry and Exit

When an interrupt occurs:

- The value of `mstatus.MIE` is copied into `mcause.MPIE`, and then `mstatus.MIE` is cleared, effectively disabling interrupts.
- When in CLIC mode, the interrupted interrupt level is copied into `mcause.MPIL`, and the interrupt level is set to that of the incoming interrupt as defined in its `clicintcfg` register.
- The privilege mode prior to the interrupt is encoded in `mstatus.MPP`.

- The current `pc` is copied into the `mepc` register, and then `pc` is set to the value specified by `mtvec` as defined by the `mtvec.MODE` described in Table 5.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. Interrupts can be re-enabled by explicitly setting `mstatus.MIE` or by executing an `MRET` instruction to exit the handler. 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 involved in handling RISC-V interrupts are described in Section 5.3.

5.2.2 Interrupt Levels and Priorities

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 5.3.8), however, the current privilege mode is not visible to software running on a hart.

The CLIC architecture supports pre-emption of up to 256 interrupt levels for each privilege mode, where higher-numbered interrupt levels can preempt lower-numbered interrupt levels. Interrupt level 0 corresponds to regular execution outside of an interrupt handler. The CLIC also supports programmable priorities within a given level which are used to prioritize among interrupts pending-and-enabled at the same interrupt level. The highest-priority interrupt at a given interrupt level is taken first. In case there are multiple pending-and-enabled interrupts at the same highest priority, the highest-numbered interrupt ID is taken first.

The number of available pre-emption 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.

See Section 6.3.3 and Section 6.3.4 for details of the E20 Core Complex `clicintcfg` implementation and `cliccfg` registers respectfully.

5.2.3 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.

5.3 Interrupt Control 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 1.10* and the RISC-V Core-Local Interrupt Controller Specification Version xx .

5.3.1 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 3. 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 *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*.

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

Table 3: E20 Core Complex `mstatus` Register (partial)

Interrupts are enabled by setting the MIE bit in `mstatus` and by enabling the desired individual interrupt in the `mie` register, described in Section 5.3.3.

Note that when operating in CLIC mode, `mstatus.MPP` and `mstatus.MPIE` are accessible in the `mcause` register described in Section 5.3.5.

5.3.2 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. The interrupt processing mode is defined in the lower two bits of the `mtvec` register as described in Table 5.

Machine Trap Vector Register			
CSR	`mtvec`		
Bits	**Field Name**	**Attr.**	**Description**
[1:0]	MODE	WARL	`MODE` Sets the interrupt processing mode. The encoding for the E20 Core Complex supported modes is described in Table 5.
[31:2]	BASE[31:2]	WARL	Interrupt Vector Base Address. Requires 64-byte alignment.

Table 4: `mtvec` Register

MODE Field Encoding `mtvec.MODE`		
Value	**Name**	**Description**
0x0	Direct	All exceptions set `pc` to `BASE`
0x1	Vectored	Asynchronous interrupts set `pc` to `BASE` + 4 × `mcause.EXCCODE`.
0x2	CLIC Direct	All exceptions set `pc` to `BASE`.
0x3	CLIC Vectored	Asynchronous interrupts set `pc` to the address in the vector table located at `mtvt` + 4 × `mcause.EXCCODE`

Table 5: 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.

See Table 4 for a description of the `mtvec` register. See Table 5 for a description of the `mtvec.MODE` field. See Table 9 for the E20 Core Complex interrupt exception code values.

Mode Direct

When operating in direct mode all synchronous exceptions and asynchronous interrupts trap to the `mtvec.BASE` address. Inside the trap handler, software must read the `mcause` register to determine what triggered the trap.

Mode Vectored

While operating in vectored mode, interrupts set the `pc` to `mtvec.BASE` + 4 × exception code. For example, if a machine timer interrupt is taken, the `pc` is set to `mtvec.BASE` + 0x1C. Typically, the trap vector table is populated with jump instructions to transfer control to interrupt-specific trap handlers.

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

All machine external interrupts (global interrupts) are mapped to exception code of 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 XLEN-6 bits of mtvec for all exceptions and interrupts.

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

Mode CLIC Vectored

In vectored CLIC mode, on an interrupt, the processor switches to the handler's privilege mode and sets the hardware vectoring bit `mcause.MINHV`, then fetches an XLEN-bit handler address from the in-memory vector table pointed to by `mtvt` described in Section 5.3.6. The address fetched is defined in the following formula:
`mtvt+4×mcause.EXCCODE`.

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

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

5.3.3 Machine Interrupt Enable (`mie`)

Individual interrupts are enabled by setting the appropriate bit in the `mie` register. The `mie` register is described in Table 6.

Machine Interrupt Enable Register			
CSR	`mie`		
Bits	**Field Name**	**Attr.**	**Description**
[2:0]	Reserved	WPRI	
3	MSIE	RW	Machine Software Interrupt Enable
[6:4]	Reserved	WPRI	
7	MTIE	RW	Machine Timer Interrupt Enable
[10:8]	Reserved	WPRI	
11	MEIE	RW	Machine External Interrupt Enable
[15:12]	Reserved	WPRI	
16	LIE0	RW	Local Interrupt 0 Enable
17	LIE1	RW	Local Interrupt 1 Enable
18	LIE2	RW	Local Interrupt 2 Enable
…			
31	LIE15	RW	Local Interrupt 15 Enable

Table 6: `mie` Register

When in either of the CLIC modes, the `mie` register is hardwired to zero and individual interrupt enables are controlled by the `clicintie[i]` CLIC memory-mapped registers. See Chapter 6 for a detailed description of `clicintie`.

5.3.4 Machine Interrupt Pending (mip)

The machine interrupt pending (`mip`) register indicates which interrupts are currently pending. The `mip` register is described in Table 7.

Machine Interrupt Pending Register			
CSR	`mip`		
Bits	**Field Name**	**Attr.**	**Description**
[2:0]	Reserved	WIRI	
3	MSIP	RO	Machine Software Interrupt Pending
[6:4]	Reserved	WIRI	
7	MTIP	RO	Machine Timer Interrupt Pending
[10:8]	Reserved	WIRI	
11	MEIP	RO	Machine External Interrupt Pending
[15:12]	Reserved	WIRI	
16	LIP0	RO	Local Interrupt 0 Pending
17	LIP1	RO	Local Interrupt 1 Pending
18	LIP2	RO	Local Interrupt 2 Pending
...			
31	LIP15	RO	Local Interrupt 15 Pending

Table 7: `mip` Register

In CLIC mode, the `mip` register is hardwired to zero and individual interrupt enables are controlled by the `clicintip[i]` CLIC memory-mapped registers. See Chapter 6 for a detailed description of `clicintip`.

5.3.5 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 8 for more details about the `mcause` register. Refer to Table 9 for a list of synchronous exception codes.

Machine Cause Register			
CSR	`mcause`		
Bits	**Field Name**	**Attr.**	**Description**
[9:0]	Exception Code	WLRL	A code identifying the last exception.
[22:10]	Reserved	WLRL	
23	mpie	WLRL	Previous interrupt enable, same as `mstatus.mpie`. CLIC mode only.
[27:24]	mpil	WLRL	Previous interrupt level. CLIC mode only.
[29:28]	mpp	WLRL	Previous interrupt privilege mode, same as `mstatus.mpp`. 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 8: `mcause` Register

Interrupt Exception Codes		
Interrupt	**Exception Code**	**Description**
1	0–2	Reserved
1	3	Machine software interrupt
1	4–6	Reserved
1	7	Machine timer interrupt
1	8–10	Reserved
1	11	Machine external interrupt
1	12	CLIC Software Interrupt Pending (CSIP)
1	13–15	Reserved
1	16	CLIC Local Interrupt 0
1	17	CLIC Local Interrupt 1
1	18–31	…
1	48	CLIC Local Interrupt 32
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8–10	Reserved
0	11	Environment call from M-mode
0	≥ 12	Reserved

Table 9: `mcause` Exception Codes

5.3.6 Machine Trap Vector Table (`mtvt`)

The `mtvt` register holds the base address of the trap vector table. `mtvt` must be 64-byte aligned and values other than 0 in the low 6 bits of mtvt are reserved.

Machine Trap Vector Table Register			
CSR	`mtvt`		
Bits	**Field Name**	**Attr.**	**Description**
[31:6]	Base	WARL	Base address of the CLIC Vector Table
[5:0]	Reserved	WARL	

Table 10: `mtvt` Register

```
          # Vector table layout for RV32 (4-byte function pointers)
mtvt ->   0x800000 # Interrupt 0 handler function pointer
          0x800004 # Interrupt 1 handler function pointer
          0x800008 # Interrupt 2 handler function pointer
          0x80000c # Interrupt 3 handler function pointer
```


5.3.7 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 incuring 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 acccesses `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.

5.3.8 Machine Interrupt Status (`mintstatus`)

A new M-mode CSR, `mintstatus`, holds the active interrupt level for each supported privilege mode. These fields are read-only. The primary reason to expose these fields is to support debug.

Machine Interrupt Status Register			
CSR	`mintstatus`		
Bits	**Field Name**	**Attr.**	**Description**
[23:0]	Reserved	WIRI	
[31:24]	mil	WIRL	Active Machine Mode Interrupt Level
[32:13]	Reserved	WIRI	

Table 11: E20 Core Complex `mintstatus` Register

5.4 Interrupt Latency

Interrupt latency for the E20 Core Complex is 6 cycles, as counted by the numbers of cycles it takes from signaling of the interrupt to the hart to the first instruction fetch of the handler.

Chapter 6

Core-Local Interrupt Controller (CLIC)

This chapter describes the operation of the Core-Local Interrupt Controller (CLIC). The E20 Core Complex implements 20180709 of the RISC-V CLIC specification.

6.1 Interrupt Sources

The E20 Core Complex has 32 interrupt sources which can be connected to peripheral devices in addition to the standard RISC-V software and timer 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 is exposed at the top level which can be used to integrate the E20 Core Complex with an external PLIC.

CLIC Interrupt IDs are provided in the in Table 12.

E20 Core Complex Interrupt IDs		
ID	**Interrupt**	**Notes**
2 - 0	Reserved	
3	msip	Machine Software Interrupt
6 - 4	Reserved	
7	mtip	Machine Timer Interrupt
10 - 8	Reserved	
11	meip	Machine Extneral Interrupt
12	csip	CLIC Software Interrupt
15 - 13	Reserved	
16	lint0	Local Interrupt 0
17	lint1	Local Interrupt 1
…	lintX	Local Interrupt X
48	lint32	Local Interrupt 32

Table 12: E20 Core Complex Interrupt IDs

6.2 CLIC Memory Map

The CLIC memory map is separated into multiple regions depending on the number of harts which implement a CLIC; one shared region, and 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 13.

Base Address 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 14.
0x0280_0000	Hart 0	Hart 0 CLIC Base. The specific implementation of this region is described in detail in Table 15.

Table 13: CLIC Base Address

CLIC Shared Region				
Offset	**Width**	**Attr.**	**Description**	**Notes**
0x0	4B	RW	`msip` for hart 0	MSIP Registers (1 bit wide)
0x4008			Reserved	
…				
0xbff7				
0x4000	8B	RW	`mtimecmp` for hart 0	MTIMECMP Registers
0x4008			Reserved	
…				
0xbff7				
0xbff8	8B	RW	`mtime`	Timer Register
0xc000			Reserved	

Table 14: CLIC Shared Register Map

CLIC Hart Specific Region			
Offset	**Width**	**Name**	**Notes**
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 15: CLIC Hart Specific Region Map

6.3 Registers

This section will describe the functionality of the CLIC's registers.

6.3.1 CLIC Interrupt Pending (`clicintip`)

CLIC Interrupt Pending				
Address	CLIC Hart Base + 1×Interrupt ID			
Bits	**Field Name**	**Attr.**	**RST.**	**Description**
0	clicintip	RW	0	When `clicintip` is set, the corresponding Interrupt ID is pending
[7:1]	Reserved	RO	0	

Table 16: `clicintip` Register

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.

6.3.2 CLIC Interrupt Enable (`clicintie`)

CLIC Interrupt Enable				
Address	CLIC Hart Base + 0x400 + 1×Interrupt ID			
Bits	**Field Name**	**Attr.**	**RST.**	**Description**
0	clicintie	RW	0	When `clicintie` is set, the corresponding Interrupt ID is enabled
[7:1]	Reserved	RO	0	

Table 17: `clicintip` Register

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.

6.3.3 CLIC Interrupt Configuration (`clicintcfg`)

CLIC Interrupt Configuration				
Address	CLIC Hart Base + 0x800 + 1×Interrupt ID			
Bits	**Field Name**	**Attr.**	**RST.**	**Description**
[1:0]	Reserved	RO	0	
[7: 5]	clicintcfg	RW	0	`clicintcfg` sets the pre-emption level and priority of a given interrupt.

Table 18: `clicintcfg` Register

The E20 Core Complex has a total of 2 bits in `clicintcfg` which specify how to encode a given interrupt's pre-emption level and/or priority. The actual number of bits which determine the pre-emption level is determined by `cliccfg.NLBITS`. If `cliccfg.NLBITS` is < 2, then the remaining least significant implemented bits are used to encode priorities within a given pre-emption level. If `cliccfg.NLBITS` is set to zero, then all interrupts are treated as level 15 and all 2 bits are used to set priorities.

See Section 5.2.2 for a description of CLIC levels and priorities.

6.3.4 CLIC Configuration (`cliccfg`)

CLIC Configuration				
Address	CLIC Hart Base + 0xC00			
Bits	**Field Name**	**Attr.**	**RST.**	**Description**
[0]	nvbits	RW	0	When set, selective hardware vectoring is enabled.
[4:1]	nlbits	RW	0	Determines the number of Level bits available in `clicintcfg`
[6:5]	nmbits	RO	0	Determines the number Mode bits available in `clicintcfg`.
[7]	Reserved	WARL	0	

Table 19: `clicintcfg` Register

The `cliccfg` register is used to configure the operation of the CLIC primarily by determining the function of the bits implemented in `clicintcfg` bits. The E20 Core Complex only supports Machine Mode interrupts, therefore `cliccfg.NMBITS` is set to zero.

`cliccfg.NLBITS` is used to determine the number of `clicintcfg` bits used for levels vs priorities. The CLIC supports a maximum of 256 pre-emption levels which requires 8 bits to encode all 16 levels. For values of `cliccfg.NLBITS`< 8, the lower bits are assumed to be all 1s. The resulting encoding of `cliccfg.NLBITS` to interrupt levels is shown below.

```
#NLBITS   encoding           interrupt levels
      0    1111                                                           255
      1    x111                         127,                              255
      2    xx11             63,         127,            191,              255
      3    xxx1        31,   63,   95,   127,    159,    191,    223,    255
      4    xxxx     15,31,47,63,79,95,111,127,143,159,175,191,207,223,239,255

x bits are available `clicintcfg` bits
```

See Section 6.3.3 for a description of the effects of `cliccfg.NLBITS` on `clicintcfg`.

`cliccfg.NVBITS` allows for certain, selected, interrupts to be vectored while in the CLIC's Direct mode. If in CLIC Direct mode and `cliccfg.NVBITS` = 1, then selective interrupt vectoring is turned on. The least-significant implemented bit of `clicintcfg` (bit 5 in 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.

Chapter 7

Debug

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

7.1 Debug CSRs

This section describes the per-hart trace and debug registers (TDRs), which are mapped into the CSR space as follows:

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

Table 20: Debug Control and Status Registers

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

7.1.1 Trace and Debug Register Select (`tselect`)

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

The `tselect` register has the format shown below:

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

Table 21: `tselect` CSR

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

7.1.2 Trace and Debug Data Registers (`tdata1-3`)

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

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

Table 22: `tdata1` CSR

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

Table 23: `tdata2/3` CSRs

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

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

Table 24: `tdata` Types

The `dmode` bit selects between debug mode (`dmode`=1) and machine mode (`dmode`=1) views of the registers, where only debug mode code can access the debug mode view of the TDRs. Any attempt to read/write the `tdata1-3` registers in machine mode when `dmode=1` raises an illegal instruction exception.

7.1.3 Debug Control and Status Register (`dcsr`)

This register gives information about debug capabilities and status. Its detailed functionality is described in *The RISC-V Debug Specification 0.13*.

7.1.4 Debug PC `dpc`

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

7.1.5 Debug Scratch `dscratch`

This register is generally reserved for use by Debug ROM in order to save registers needed by the code in Debug ROM. The debugger may use it as described in *The RISC-V Debug Specification 0.13*.

7.2 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 25: TDR CSRs when used as Breakpoints

7.2.1 Breakpoint Match Control Register `mcontrol`

Each breakpoint control register is a read/write register laid out in Table 26.

Breakpoint Control Register (`mcontrol`)				
Register Offset		CSR		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	R	WARL	X	Address match on LOAD
1	W	WARL	X	Address match on STORE
2	X	WARL	X	Address match on Instruction FETCH
3	U	WARL	X	Address match on User Mode
4	S	WARL	X	Address match on Supervisor Mode
5	Reserved	WPRI	X	Reserved
6	M	WARL	X	Address match on Machine Mode
[10:7]	match	WARL	X	Breakpoint Address Match type
11	chain	WARL	0	Chain adjacent conditions.
[17:12]	action	WARL	0	Breakpoint action to take. 0 or 1.
18	timing	WARL	0	Timing of the breakpoint. Always 0.
19	select	WARL	0	Perform match on address or data. Always 0.
20	Reserved	WPRI	X	Reserved
[26:21]	maskmax	RO	4	Largest supported NAPOT range
27	dmode	RW	0	Debug-Only access mode
[31:28]	type	RO	2	Address/Data match type, always 2

Table 26: Test and Debug Data Register 3

The `type` field is a 4-bit read-only field holding the value 2 to indicate this is a breakpoint containing address match logic.

The `bpaction` field is an 8-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/instruction fetches, respectively, and 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/user modes, respectively, and all combinations of implemented bits must be supported.

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

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

maddress	**Match type and size**
a…aaaaaa	Exact 1 byte
a…aaaaa0	2-byte NAPOT range
a…aaaa01	4-byte NAPOT range
a…aaa011	8-byte NAPOT range
a…aa0111	16-byte NAPOT range
a…a01111	32-byte NAPOT range
…	…
a01…1111	2^{31}-byte NAPOT range

Table 27: 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 `maddress` with the 30 least-significant bits set to 1, bit 30 set to 0, and bit 31 holding the only address bit considered in the address comparison.

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.

7.2.2 Breakpoint Match Address Register (`maddress`)

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

7.2.3 Breakpoint Execution

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

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

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

7.2.4 Sharing Breakpoints Between Debug and Machine Mode

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

7.3 Debug Memory Map

This section describes the debug module's memory map when accessed via the regular system interconnect. The debug module is only accessible to debug code running in debug mode on a hart (or via a debug transport module).

7.3.1 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 `DMHARTINFO` register as described in the RISC-V Debug Specification. This RAM space is used to pass data for the Access Register abstract command described in the RISC-V Debug Specification. The E20 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.

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

7.3.2 Debug ROM (`0x800–0xFFF`)

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

7.3.3 Debug Flags (`0x100–0x110`, `0x400–0x7FF`)

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

7.3.4 Safe Zero Address

In the E20 Core Complex, the debug module contains the address `0x0` in the memory map. Reads to this address always return 0, and writes to this address have no impact. This property allows a "safe" location for unprogrammed parts, as the default `mtvec` location is `0x0`.

Chapter 8

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, May 2017. [Online]. Available: https://riscv.org/specifications/

[2] ——, The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10, May 2017. [Online]. Available: https://riscv.org/specifications/

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