Western Digital®

RISC-V SweRV™ EH1
Programmer's Reference Manual

Revision 1.0

January 24, 2019

Document Revision History

Revision	Date	Contents
1.0	Jan 24, 2019	Initial revision

Table of Contents

List of Figures

List of Tables

Reference Documents

Item #	Document	Revision Used	Comment
1	The RISC-V Instruction Set Manual Volume I: User-Level ISA	20181106-Base-Ratication	
2	The RISC-V Instruction Set Manual Volume II: Privileged Architecture	20190125-Public-Review-*draft*	
3	RISC-V External Debug Support	0.13	Spec ratified

Abbreviations

Abbreviation	Description
AHB	Advanced High-performance Bus (by ARM®)
AMBA	Advanced Microcontroller Bus Architecture (by ARM)
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface (by ARM)
CCM	Closely Coupled Memory (= TCM)
CPU	Central Processing Unit
CSR	Control and Status Register
DCCM	Data Closely Coupled Memory (= DTCM)
DEC	DECoder unit (part of core)
DMA	Direct Memory Access
DTCM	Data Tightly Coupled Memory (= DCCM)
ECC	Error Correcting Code
EXU	EXecution Unit (part of core)
ICCM	Instruction Closely Coupled Memory (= ITCM)
IFU	Instruction Fetch Unit
ITCM	Instruction Tightly Coupled Memory (= ICCM)
JTAG	Joint Test Action Group
LSU	Load/Store Unit (part of core)
NMI	Non-Maskable Interrupt
PIC	Programmable Interrupt Controller
PLIC	Platform-Level Interrupt Controller
POR	Power-On Reset
RAM	Random Access Memory
ROM	Read-Only Memory
SECDED	Single-bit Error Correction/Double-bit Error Detection
SEDDED	Single-bit Error Detection/Double-bit Error Detection
SOC	System On Chip
TBD	To Be Determined
TCM	Tightly Coupled Memory (= CCM)

1 SweRV EH1 Core Overview

This chapter provides a high-level overview of the SweRV EH1 core and core complex. SweRV EH1 is a 32-bit CPU core which supports RISC-V's integer (I), compressed instruction (C), multiplication and division (M), and instruction-fetch fence and CSR instructions (Z) extensions, (i.e., RV32IMCZifencei_Zicsr). The core is a 9-stage, dual-issue, superscalar, mostly in-order pipeline with some out-of-order execution capability.

1.1 Features

The SweRV EH1 core complex's feature set includes:

- RV32IMCZifencei_Zicsr-compliant RISC-V core with branch predictor
- Optional instruction and data closely-coupled memories with ECC protection
- Optional 4-way set-associative instruction cache with parity or ECC protection
- Optional programmable interrupt controller supporting up to 255 external interrupts
- Four system bus interfaces for instruction fetch, data accesses, debug accesses, and external DMA accesses to closely-coupled memories (configurable as 64-bit AXI or AHB-Lite)
- Core debug unit compliant with the RISC-V Debug specification [3]
- 1GHz target frequency (for 28nm technology node)

1.2 Core Complex

Figure 1-1 depicts the core complex and its functional blocks which are described further in Section 1.3.

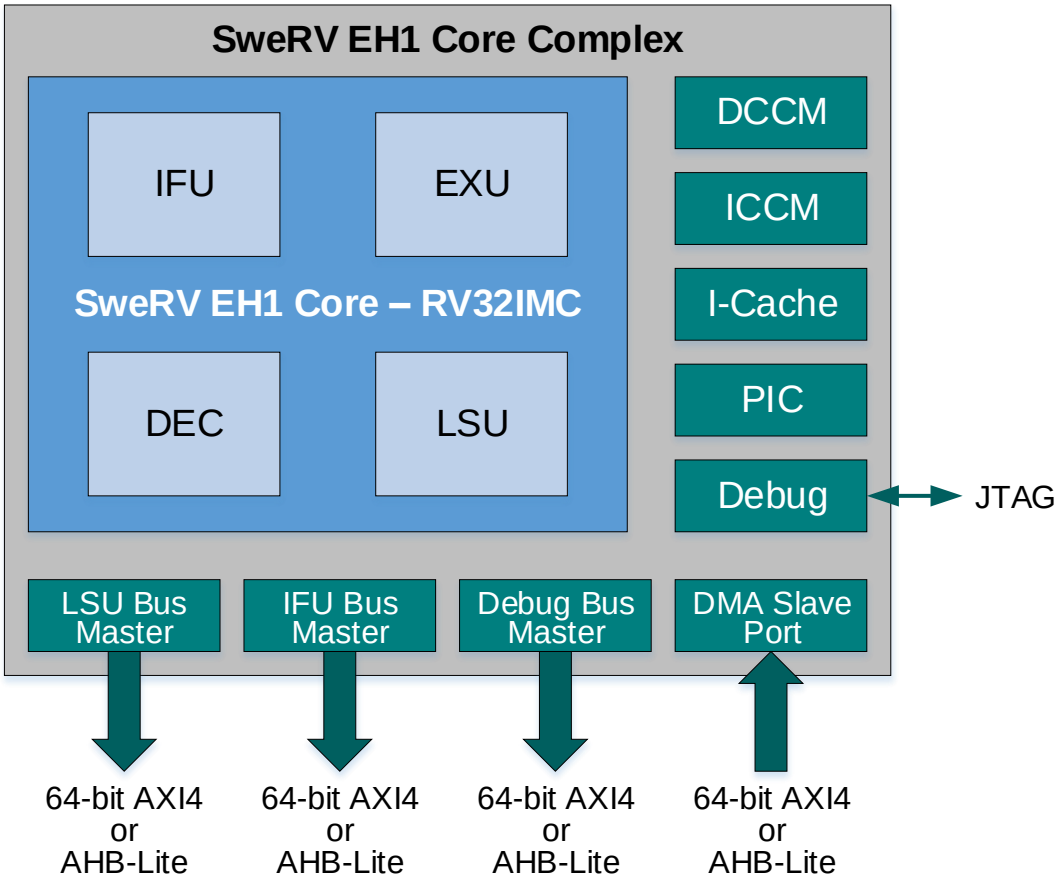


Figure 1-1 SweRV EH1 Core Complex

1.3 Functional Blocks

The SweRV EH1 core complex's functional blocks are described in the following sections in more detail.

1.3.1 Core

Figure 1-2 depicts the 9-stage core pipeline.

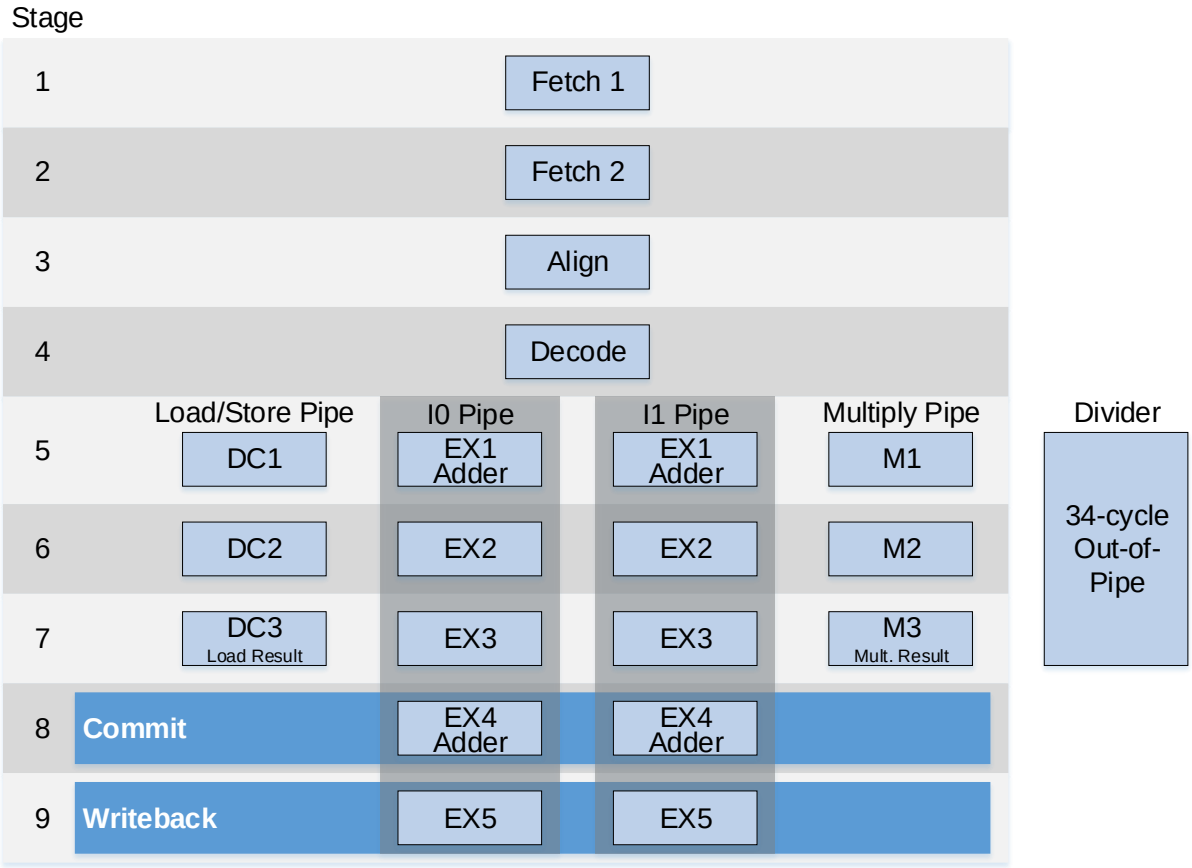


Figure 1-2 SweRV EH1 Core Pipeline

2 Memory Map

This chapter describes the memory map as well as the various memories and their properties of the SweRV EH1 core.

2.1 Address Regions

The 32-bit address space is subdivided into sixteen fixed-sized, contiguous 256MB regions. Each region has a set of access control bits associated with it.

2.2 Access Properties

Each region has two access properties which can be independently controlled. They are:

- **Cacheable:** Indicates if this region is allowed to be cached or not.
- **Side effect:** Indicates if read/write accesses to this region may have side effects (i.e., non-idempotent accesses which may potentially have side effects on any read/write access; typical for I/O, speculative or redundant accesses must be avoided) or have no side effects (i.e., idempotent accesses which have no side effects even if the same access is performed multiple times; typical for memory). Note that stores with potential side effects (i.e., to non-idempotent addresses) cannot be combined with other stores in the core's write buffer.

2.3 Memory Types

There are two different classes of memory types mapped into the core's 32-bit address range, core local and system bus attached.

2.3.1 Core Local

2.3.1.1 ICCM and DCCM

Two dedicated memories, one for instruction and the other for data, are tightly coupled to the core. These memories provide low-latency access and SECDED ECC protection. Their respective sizes (4, 8, 16, 32, 48[1], 64, 128, 256, or 512KB) are set as arguments at build time of the core.

2.3.1.2 Local Memory-mapped Control/Status Registers

To provide control for regular operation, the core requires a number of memory-mapped control/status registers. For example, some external interrupt functions are controlled and serviced with accesses to various registers while the system is running.

2.3.2 Accessed via System Bus

2.3.2.1 System ROMs

The SoC may host ROMs which are mapped to the core's memory address range and accessed via the system bus. Both instruction and data accesses are supported to system ROMs.

2.3.2.2 System SRAMs

The SoC hosts a variety of SRAMs which are mapped to the core's memory address range and accessed via the system bus.

2.3.2.3 System Memory-mapped I/O

The SoC hosts a variety of I/O device interfaces which are mapped to the core's memory address range and accessed via the system bus.

[1] DCCM only

2.3.3 Mapping Restrictions

Core-local memories and system bus-attached memories must be mapped to different regions. Mapping both classes of memory types to the same region is not allowed.

Furthermore, it is recommended that all core-local memories are mapped to the same region.

2.4 Memory Type Access Properties

Table 2-1 specifies the access properties of each memory type. During system boot, firmware must initialize the properties of each region based on the memory type present in that region.

Note that some memory-mapped I/O and control/status registers may have no side effects (i.e., are idempotent), but characterizing all these registers as having potentially side effects (i.e., are non-idempotent) is safe.

Table 2-1 Access Properties for each Memory Type

Memory Type	Cacheable	Side Effect
Core Local		
ICCM	No	No
DCCM	No	No
Memory-mapped control/status registers	No	Yes
Accessed via System Bus		
ROMs	Yes	No
SRAMs	Yes	No
I/Os	No	Yes
Memory-mapped control/status registers	No	Yes

Note: ‘Cacheable = Yes’ and ‘Side Effect = Yes’ is an illegal combination.

2.5 Memory Access Ordering

Loads and stores to system bus-attached memory (i.e., accesses with no side effects, idempotent) and devices (i.e., accesses with potential side effects, non-idempotent) go through a read buffer and a write buffer, respectively. The buffers are implemented as FIFOs.

2.5.1 Load-to-Load and Store-to-Store Ordering

Loads are typically sent to the system bus interface in program order. However, for system buses allowing multiple outstanding transactions, reordering may occur in some cases.

All stores are always sent to the system bus interface in program order.

2.5.2 Load/Store Ordering

2.5.2.1 Accesses with Potential Side Effects (i.e., Non-Idempotent)

When a load with potential side effects (i.e., non-idempotent) enters the read buffer, the entire write buffer is drained, i.e., both stores with no side effects (i.e., idempotent) and with potential side effects (i.e., non-idempotent) are drained out. Loads with potential side effects (i.e., non-idempotent) are sent out to the system bus with their exact size.

Stores with potential side effects (i.e., non-idempotent) are neither coalesced nor forwarded to a load.

2.5.2.2 Accesses with No Side Effects (i.e., Idempotent)

Loads with no side effects (i.e., idempotent) are always issued as double-words and check the contents of the write buffer:

1. **Full address match** (all load bytes present in the write buffer): Data is forwarded from the write buffer. The load does not freeze the pipe, and won't go out to the system bus.
2. **Partial address match** (some of the load bytes are in the write buffer): The entire write buffer is drained, then the load request goes to the system bus.
3. **No match** (none of the bytes are in the write buffer): The load is presented to the system bus interface without waiting for the stores to drain.

2.5.2.3 Ordering of Store – Load with No Side Effects (i.e., Idempotent)

A `fence` instruction is required to order an older store before a younger load with no side effects (i.e., idempotent).

Note: All memory-mapped register writes must be followed by a `fence` instruction to enforce ordering and synchronization.

2.5.3 Fencing

2.5.3.1 Instructions

The `fence.i` instruction operates on the instruction memory and/or I-cache. This instruction causes a flush, a flash invalidation of the I-cache, and a refetch of the next program counter (RFNPC). The refetch is guaranteed to miss the I-cache. Note that since the `fence.i` instruction is used to synchronize the instruction and data streams, it also includes the functionality of the `fence` instruction (see Section 2.5.3.2).

2.5.3.2 Data

The `fence` instruction is implemented conservatively in SweRV EH1 to keep the implementation simple. It always performs the most conservative fencing, independent of the instruction's arguments. The `fence` instruction is pre-synced to make sure that there are no instructions in the LSU pipe. It stalls until the LSU indicates that the read buffer has been cleared as well as the store and write buffers have been fully drained. It is only committed after all LSU buffers are idle.

2.5.4 Imprecise Data Bus Errors

All store errors as well as non-blocking load errors on the system bus are imprecise. The address of the first occurring imprecise data system bus error is logged and a non-maskable interrupt (NMI) is flagged for the first reported error only. For stores, if there are other stores in the write buffer behind the store which had the error, these stores are sent out on the system bus and any error responses are ignored. Similarly, for non-blocking loads, any error responses on subsequent loads sent out on the system bus are ignored. NMIs are fatal, architectural state is lost, and the core needs to be reset. The reset also unlocks the first error address capture register again.

Note: It is possible to unlock the first error address capture register with a write to an unlock register as well (see Section 2.7.4 for more details), but this may result in unexpected behavior.

2.6 Exception Handling

Capturing the faulting effective address causing an exception helps assist firmware in handling the exception and/or provides additional information for firmware debugging. For precise exceptions, the faulting effective address is captured in the standard RISC-V `mtval` register (see Section 3.1.21 in [2]). For imprecise exceptions, the address of the first occurrence of the error is captured in a platform-specific error address capture register (see Section 2.7.3).

2.6.1 Imprecise Bus Error Non-Maskable Interrupt

Store bus errors are fatal and cause a non-maskable interrupt (NMI). The store bus error NMI has an `mcause` value of 0xF000_0000.

Likewise, non-blocking load bus errors are fatal and cause a non-maskable interrupt (NMI). The non-blocking load bus error NMI has an `mcause` value of 0xF000_0001.

Note: The address of the first store or non-blocking load error on the D-bus is captured in the `mdseac` register (see Section 2.7.3). The register is unlocked either by resetting the core after the NMI has been handled or by a write to the `mdeau` register (see Section 2.7.4). While the `mdseac` register is locked, subsequent D-bus errors are gated (i.e., they do not cause another NMI), but NMI requests originating external to the core are still honored.

Note: If store and non-blocking load bus errors are reported in the same clock cycle (i.e., the LSU's write and read buffers simultaneous indicate a bus error), the non-blocking load bus error has higher priority.

2.6.2 Correctable Error Local Interrupt

I-cache parity/ECC errors, ICCM correctable ECC errors, and DCCM correctable ECC errors are counted in separate correctable error counters (see Sections 3.5.1, 3.5.2, and 3.5.3, respectively). Each counter also has its separate programmable error threshold. If any of these counters has reached its threshold, a correctable error local interrupt is signaled. Firmware should determine which of the counters has reached the threshold and reset that counter.

A local-to-the-core interrupt for correctable errors has pending (*mceip*) and enable (*mceie*) bits in bit position 30 of the standard RISC-V `mip` (see Table 9-2) and `mie` (see Table 9-1) registers, respectively. The priority is lower than RISC-V External interrupt, but higher than RISC-V Timer interrupt (see Table 11-1). The correctable error local interrupt has an `mcause` value of 0x8000_001E (see Table 9-3).

2.6.3 Rules for Core-Local Memory Accesses

The rules for instruction fetch and load/store accesses to core-local memories are:

1. An instruction fetch access to a region
    a. containing one or more ICCM sub-region(s) causes an exception if
        i. the access is not completely within the ICCM sub-region, or
        ii. the boundary of an ICCM to a non-ICCM sub-region and vice versa is crossed,

    even if the region contains a DCCM/PIC memory-mapped control register sub-region.
    b. not containing an ICCM sub-region goes out to the system bus, even if the region contains a DCCM/PIC memory-mapped control register sub-region.
2. A load/store access to a region
    a. containing one or more DCCM/PIC memory-mapped control register sub-region(s) causes an exception if
        i. the access is not completely within the DCCM/PIC memory-mapped control register sub-region, or
        ii. the boundary of
            1. a DCCM to a non-DCCM sub-region and vice versa, or
            2. a PIC memory-mapped control register sub-region

        is crossed,

    even if the region contains an ICCM sub-region.
    b. not containing a DCCM/PIC memory-mapped control register sub-region goes out to the system bus, even if the region contains an ICCM sub-region.

2.6.4 Unmapped Addresses

Table 2-2 Handling of Unmapped Addresses

Access	Core/Bus	Side Effect	Action	Comments
Fetch	Core	N/A	Instruction access fault exception[2, 3]	Precise exception (e.g., address out-of-range)
	Bus	N/A	Instruction access fault exception[2]	

[2] If any byte of an instruction is from an unmapped address, an instruction access fault precise exception is flagged.

[3] Exception also flagged for fetches to the DCCM address range if located in the same region, or if located in different regions and no SoC address is a match.

Access	Core/Bus	Side Effect	Action	Comments
Load	Core	No	Load access fault exception[4]	Precise exception (e.g., address out-of-range)
	Bus	No (for non-blocking load)	Non-blocking load bus error NMI (see Section 2.6.1)	• Imprecise, fatal • Capture store address in core bus interface
		No (for blocking load)	Load access fault exception	Precise exception (e.g., address out-of-range)
		Yes		• Precise exception • Hold off all external interrupts
Store	Core	No	Store/AMO[5] access fault exception[6]	Precise exception
	Bus	No	Store bus error NMI (see Section 2.6.1)	• Imprecise, fatal • Capture store address in core bus interface
		Yes		
DMA Read	Bus	N/A	DMA slave bus error	Send error response to master
DMA Write				

Note: It is recommended to provide address gaps between different memories to ensure unmapped address exceptions are flagged if memory boundaries are inadvertently crossed.

2.6.5 Misaligned Accesses

General notes:

- The core performs a misalignment check during the address calculation.
- Accesses across region boundaries always cause a misaligned exception.
- Splitting a load/store from/to an address with no side effects (i.e., idempotent) is not of concern for SweRV EH1.

Table 2-3 Handling of Misaligned Accesses

Access	Core/Bus	Side Effect	Region Cross	Action	Comments
Fetch	Core	N/A	No	N/A	Not possible[7]
	Bus	N/A			

[4] Exception also flagged for loads to the ICCM address range if located in the same region, or if located in different regions and no SoC address is a match.

[5] AMO refers to the RISC-V “A” (atomics) extension, which is not implemented in SweRV EHX1.

[6] Exception also flagged for stores to the ICCM address range if located in the same region, or if located in different regions and no SoC address is a match.

[7] Accesses to the I-cache or ICCM initiated by fetches never cross 16B boundaries. I-cache fills are always aligned to 64B. Misaligned accesses are therefore not possible.

Access	Core/Bus	Side Effect	Region Cross	Action	Comments
Load	Core	No		Load split into multiple DCCM read accesses	Split performed by core
	Bus	No		Load split into multiple bus transactions	Split performed by core
		Yes		Load address misaligned exception	Precise exception
Store	Core	No		Store split into multiple DCCM write accesses	Split performed by core
	Bus	No		Store split into multiple bus transactions	Split performed by core
		Yes		Store/AMO address misaligned exception	Precise exception
Fetch	N/A	N/A	Yes	N/A	Not possible[7]
Load				Load address misaligned exception	Precise exception
Store				Store/AMO address misaligned exception	Precise exception
DMA Read	Bus	N/A	N/A	DMA slave bus error	Send error response to master
DMA Write[8]					

2.6.6 Uncorrectable ECC Errors

Table 2-4 Handling of Uncorrectable ECC Errors

Access	Core/Bus	Side Effect	Action	Comments
Fetch	Core	N/A	Instruction access fault exception	Precise exception
	Bus	N/A		
Load	Core	No	Load access fault exception	Precise exception
		Yes		
	Bus	No (for non-blocking load)	Non-blocking load bus error NMI (see Section 2.6.1)	• Imprecise, fatal • Capture store address in core bus interface
		No (for blocking load)	Load access fault exception	Precise exception
		Yes		

[8] This case is in violation with the write alignment rules specified in Section 2.10.2.

Access	Core/Bus	Side Effect	Action	Comments
Store	Core	No	Store/AMO access fault exception	Precise exception
		Yes		
	Bus	No	Store bus error NMI (see Section 2.6.1)	• Imprecise, fatal • Capture store address in core bus interface
		Yes		
DMA Read	Bus	N/A	DMA slave bus error	Send error response to master

Note: DMA write accesses to the ICCM or DCCM always overwrite entire 32-bit words and their corresponding ECC bits. Therefore, ECC bits are never checked and errors not detected on DMA writes.

2.6.7 Correctable ECC/Parity Errors

Table 2-5 Handling of Correctable ECC/Parity Errors

Access	Core/Bus	Side Effect	Action	Comments

Fetch	Core	N/A	For I-cache accesses:
			• Increment correctable I-cache error counter in core
			• If I-cache error threshold reached, signal correctable error local interrupt (see Section 3.5.1)
			• Invalidate all cache lines of set
			• Perform RFPC flush
			  • Flush core pipeline
			  • Refetch cache line from SoC memory
				For I-cache with tag/instruction ECC protection, single- and double-bit errors are recoverable

			For ICCM accesses:
			• Increment correctable ICCM error counter in core
			• If ICCM error threshold reached, signal correctable error local interrupt (see Section 3.5.2)
			• Perform RFPC flush
			  • Flush core pipeline
			  • Write corrected data back to ICCM
			  • Refetch instruction(s) from ICCM
				ICCM errors trigger an RFPC (ReFetch PC) flush since in-pipeline correction would require an additional cycle

	Bus	N/A	• Increment correctable error counter in SoC • If error threshold reached, signal external interrupt • Write corrected data back to SoC memory	Errors in SoC memories are corrected at memory boundary and autonomously written back to memory array

Access	Core/Bus	Side Effect	Action	Comments
Load	Core	No	• Increment correctable DCCM error counter in core • If DCCM error threshold reached, signal correctable error local interrupt (see Section 3.5.3) • Write corrected data back to DCCM	DCCM errors are in-pipeline corrected and written back to DCCM
		Yes		
	Bus	No	• Increment correctable error counter in SoC • If error threshold reached, signal external interrupt • Write corrected data back to SoC memory	Errors in SoC memories are corrected at memory boundary and autonomously written back to memory array
		Yes		
Store	Core	No	• Increment correctable DCCM error counter in core • If DCCM error threshold reached, signal correctable error local interrupt (see Section 3.5.3) • Write corrected data back to DCCM	DCCM errors are in-pipeline corrected and written back to DCCM
		Yes		
	Bus	No	• Increment correctable error counter in SoC • If error threshold reached, signal external interrupt • Write corrected data back to SoC memory	Errors in SoC memories are corrected at memory boundary and autonomously written back to memory array
		Yes		
DMA Read	Bus	N/A	For ICCM accesses: • Increment correctable ICCM error counter in core • If ICCM error threshold reached, signal correctable error local interrupt (see Section 3.5.2) • Write corrected data back to ICCM	DMA read access errors to ICCM are in-pipeline corrected and written back to ICCM
			For DCCM accesses: • Increment correctable DCCM error counter in core • If DCCM error threshold reached, signal correctable error local interrupt (see Section 3.5.3) • Write corrected data back to DCCM	DMA read access errors to DCCM are in-pipeline corrected and written back to DCCM

Note: Counted errors could be from different, unknown memory locations.

Note: DMA write accesses to the ICCM or DCCM always overwrite entire 32-bit words and their corresponding ECC bits. Therefore, ECC bits are never checked and errors not detected on DMA writes.

2.7 Control/Status Registers

A summary of platform-specific control/status registers in CSR space:

- Region Access Control Register (mrac) (see Section 2.7.1)
- Memory Synchronization Trigger Register (dmst) (see Section 2.7.2)
- D-Bus First Error Address Capture Register (mdseac) (see Section 2.7.3)
- D-Bus Error Address Unlock Register (mdeau) (see Section 2.7.4)

All reserved and unused bits in these control/status registers must be hardwired to '0'. Unless otherwise noted, all read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

2.7.1 Region Access Control Register (mrac)

A single region access control register is sufficient to provide independent control for 16 address regions.

Note: To guarantee that updates to the `mrac` register are in effect, if a region being updated is in the load/store space, a `fence` instruction is required. Likewise, if a region being updated is in the instruction space, a `fence.i` instruction (which flushes the I-cache) is required.

Note: The access control bits for any region hosting a core-local memory (i.e., ICCM or DCCM) are ignored by the core.

Note: The combination '11' (i.e., side effect and cacheable) is illegal. Writing '11' is mapped by hardware to the legal value '10' (i.e., side effect and non-cacheable).

This register is mapped to the non-standard read/write CSR address space.

Table 2-6 Region Access Control Register (mrac, at CSR 0x7C0)

Field	Bits	Description	Access	Reset
Y = 0..15 (= Region)				
sideeffect	*Y**2+1	Side effect indication for region *Y*: 0: No side effects (idempotent) 1: Side effects possible (non-idempotent)	R/W	0
cacheable	*Y**2	Caching control for region *Y*: 0: Caching not allowed 1: Caching allowed	R/W	0

2.7.2 Memory Synchronization Trigger Register (dmst)

The `dmst` register provides triggers to force the synchronization of memory accesses. Specifically, it allows a debugger to initiate operations that are equivalent to the `fence.i` (see Section 2.5.3.1) and `fence` (see Section 2.5.3.2) instructions.

Note: This register is accessible in **debug mode only**. Attempting to access this register in machine mode raises an illegal instruction exception.

The *fence.i* and *fence* fields of the `dmst` register have W1R0 (Write 1, Read 0) behavior, as also indicated in the 'Access' column.

This register is mapped to the non-standard read/write CSR address space.

Table 2-7 Memory Synchronization Trigger Register (dmst, at CSR 0x7C4)

Field	Bits	Description	Access	Reset
Reserved	31:2	Reserved	R	0
fence	1	Trigger operation equivalent to `fence` instruction	R0/W1	0
fence.i	0	Trigger operation equivalent to `fence.i` instruction	R0/W1	0

2.7.3 D-Bus First Error Address Capture Register (mdseac)

The address of the first occurrence of a store or non-blocking load error on the D-bus is captured in the `mdseac` register. Latching the address also locks the register. While the `mdseac` register is locked, subsequent D-bus errors are gated (i.e., they do not cause another NMI), but NMI requests originating external to the core are still honored. The `mdseac` register is unlocked by either a core reset (which is the safer option) or by writing to the `mdeau` register (see Section 2.7.4).

Note: The NMI handler may use the value stored in the `mcause` register to differentiate between a D-bus store error, a D-bus non-blocking load error, and a core-external event triggering an NMI.

This register is mapped to the non-standard read-only CSR address space.

Table 2-8 D-Bus First Error Address Capture Register (mdseac, at CSR 0xFC0)

Field	Bits	Description	Access	Reset
erraddr	31:0	Address of first occurrence of D-bus store or non-blocking load error	R	0

2.7.4 D-Bus Error Address Unlock Register (mdeau)

Writing to the `mdeau` register unlocks the `mdseac` register (see Section 2.7.3) after a D-bus error address has been captured. This write access also reenables the signaling of an NMI for a subsequent D-bus error.

Note: Nested NMIs might destroy core state and, therefore, receiving an NMI should still be considered fatal. Issuing a core reset is a safer option to deal with a D-bus error.

The `mdeau` register has WAR0 (Write Any value, Read 0) behavior. Writing '0' is recommended.

This register is mapped to the non-standard read/write CSR address space.

Table 2-9 D-Bus Error Address Unlock Register (mdeau, at CSR 0xBC0)

Field	Bits	Description	Access	Reset
Reserved	31:0	Reserved	R0/WA	0

2.8 Memory Address Map

Table 2-10 summarizes an example of the SweRV EH1 memory address map, including regions as well as start and end addresses for the various memory types.

Table 2-10 SweRV EH1 Memory Address Map (Example)

Region	Start Address	End Address	Memory Type
0x0	0x0000_0000	0x0003_FFFF	Reserved
	0x0004_0000	0x0005_FFFF	ICCM (region: 0, offset: 0x4000, size: 128KB)
	0x0006_0000	0x0007_FFFF	Reserved
	0x0008_0000	0x0009_FFFF	DCCM (region: 0, offset: 0x8000, size: 128KB)
	0x000A_0000	0x0FFF_FFFF	Reserved
0x1	0x1000_0000	0x1FFF_FFFF	System memory-mapped CSRs
0x2	0x2000_0000	0x2FFF_FFFF	
0x3	0x3000_0000	0x3FFF_FFFF	

Region	Start Address	End Address	Memory Type
0x4	0x4000_0000	0x4FFF_FFFF	System SRAMs, system ROMs, and system memory-mapped I/O device interfaces
0x5	0x5000_0000	0x5FFF_FFFF	
0x6	0x6000_0000	0x6FFF_FFFF	
0x7	0x7000_0000	0x7FFF_FFFF	
0x8	0x8000_0000	0x8FFF_FFFF	
0x9	0x9000_0000	0x9FFF_FFFF	
0xA	0xA000_0000	0xAFFF_FFFF	
0xB	0xB000_0000	0xBFFF_FFFF	
0xC	0xC000_0000	0xCFFF_FFFF	
0xD	0xD000_0000	0xDFFF_FFFF	
0xE	0xE000_0000	0xEFFF_FFFF	
0xF	0xF000_0000	0xFFFF_FFFF	

2.9 Partial Writes

Rules for partial writes handling are:

- **Core-local addresses:** The core performs a read-modify-write operation and updates ECC to core-local memories (i.e., I- and DCCMs).
- **SoC addresses:** The core indicates the valid bytes for each bus write transaction. The addressed SoC memory or device performs a read-modify-write operation and updates its ECC.

2.10 DMA Slave Port

The Direct Memory Access (DMA) slave port is used for read/write accesses to core-local memories initiated by external masters. For example, external masters could be DMA controllers or other CPU cores located in the SoC.

2.10.1 Access

The DMA slave port allows read/write access to the core's ICCM and DCCM. However, the PIC memory-mapped control registers are not accessible via the DMA port.

2.10.2 Write Alignment Rules

For writes to the ICCM and DCCM through the DMA slave port, **a**ccesses must be 32- or 64-bit aligned, and 32 bits (word) or 64 bits (double-word), respectively, wide to avoid read-modify-write operations for ECC generation.

2.10.3 Quality of Service

Accesses to the ICCM and DCCM by the core have higher priority. However, to avoid starvation, the DMA slave port's DMA controller may periodically request a stall to get access to the pipe if a DMA request is continuously blocked.

2.11 Reset Signal and Vector

The core provides a 31-bit wide input bus at its periphery for a reset vector. The SoC must provide the reset vector on the `rst_vec[31:1]` bus, which could be hardwired or from a register. The `rst_l` input signal is active-low,

asynchronously asserted, and synchronously de-asserted (see also Section 12.3). When the core is reset, it fetches the first instruction to be executed from the address provided on the reset vector bus.

Note: The core's 31 general-purpose registers (`x1` - `x31`) are cleared on reset.

2.12 Non-Maskable Interrupt (NMI) Signal and Vector

The core also provides a separate 31-bit wide input bus at its periphery for a non-maskable interrupt (NMI) vector. If the SoC makes use of this feature, it must provide the NMI vector on the `nmi_vec[31:1]` bus, which could be hardwired or from a register.

The `nmi_int` input signal is low-to-high edge-triggered and asynchronous. It must be asserted for at least two full core clock cycles to guarantee it is detected by the core since shorter pulses might be dropped by the synchronizer circuit. Furthermore, the `nmi_int` signal must be de-asserted for a minimum of two full core clock cycles and then reasserted to signal the next NMI request to the core.

When the core receives an NMI request, it fetches the next instruction to be executed from the address provided on the NMI vector bus. If the SoC does not use the NMI feature, it must hardwire the `nmi_int` and `nmi_vec[31:1]` input signals to 0.

NMIs triggered by asserting the core's `nmi_int` input signal have an `mcause` value of 0x0000_0000.

3 Memory Error Protection

3.1 General Description

3.1.1 Parity

Parity is a simple and relatively cheap protection scheme generally used when the corrupted data can be restored from some other location in the system. A single parity check bit typically covers several data bits. Two parity schemes are used: even and odd parity. The total number of '1' bits are counted in the protected data word, including the parity bit. For even parity, the data is deemed to be correct if the total count is an even number. Similarly, for odd parity if the total count is an odd number. Note that double-bit errors cannot be detected.

3.1.2 Error Correcting Code (ECC)

A robust memory hierarchy design often includes ECC functions to detect and, if possible, correct corrupted data. The ECC functions described are made possible by Hamming code, a relatively simple yet powerful ECC code. It involves storing and transmitting data with multiple check bits (parity) and decoding the associated check bits when retrieving or receiving data to detect and correct errors.

The ECC feature can be implemented with Hamming based SECDED (Single-bit Error Correction and Double-bit Error Detection) algorithm. The design can use the (39, 32) code – 32 data bits and 7 parity bits depicted in Figure 5-1 below. In other words, the Hamming code word width is 39 bits, comprised of 32 data bits and 7 check bits. The minimum number of check bits needed for correcting a single-bit error in a 32-bit word is six. The extra check bit expands the function to detect double-bit errors as well.

ECC codes may also be used for error detection only if other means exist to correct the data. For example, the I-cache stores exact copies of cache lines which are also residing in SoC memory. Instead of correcting corrupted data fetched from the I-cache, erroneous cache lines may also be invalidated in the I-cache and refetched from SoC memory. A SEDDED (Single-bit Error Detection and Double-bit Error Detection) code is sufficient in that case and provides even better protection than a SECDED code since double-bit errors are corrected as well, but requires fewer bits to protect each codeword. Note that flushing and refetching is the industry standard mechanism for recovering from I-cache errors, though commonly still referred to as 'SECDED'.

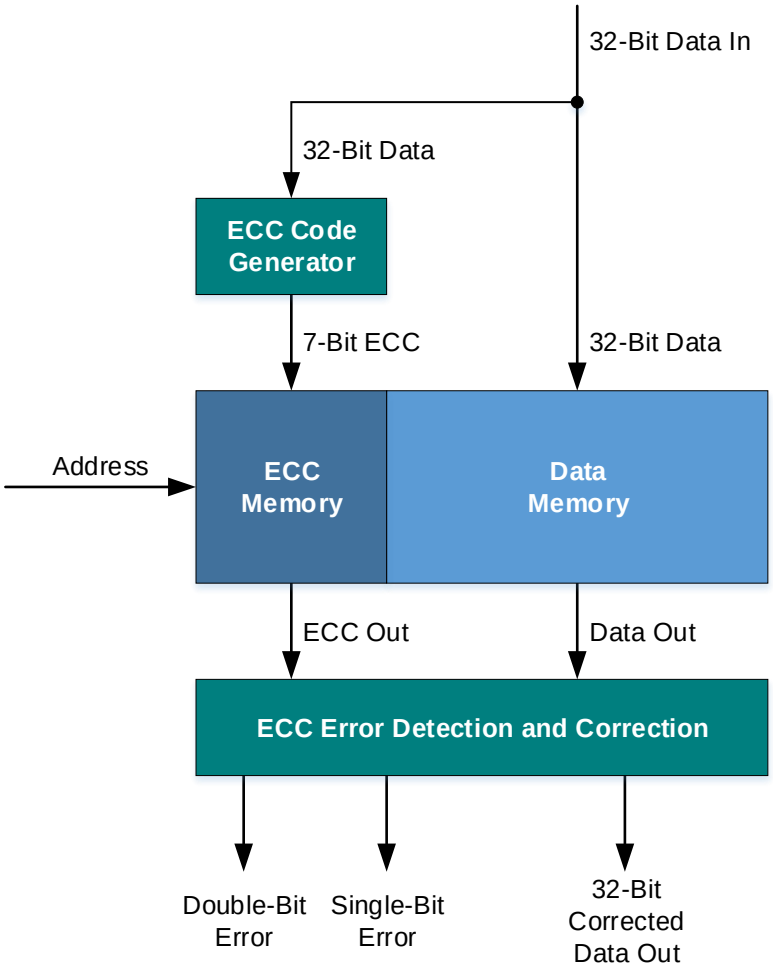


Figure 3-1 Conceptual Block Diagram – ECC in a Memory System

3.2 Selecting the Proper Error Protection Level

Choosing a protection level that is too weak might lead to loss of data or silent data corrupted, choosing a level that is too strong incurs additional chip die area (i.e., cost) and power dissipation. Supporting multiple protection schemes for the same design increases the design and verification effort.

Sources of errors can be divided into two major categories:

- Hard errors (e.g., stuck-at bits), and
- Soft errors (e.g., weak bits, cosmic-induced soft errors)

Selecting an adequate error protection level – e.g., none, parity, or ECC -- depends on the probability of an error to occur, which depends on several factors:

- Technology node
- SRAM structure size
- SRAM cell design
- Type of stored information
  - E.g., instructions in I-cache can be refetched, but
  - data might be lost if not adequately protected
- Stored information being used again after corruption

Typically, a FIT (Failure In Time) rate analysis is done to determine the proper protection level of each memory in a system. This analysis is based on FIT rate information for a given process and SRAM cell design which are typically available from chip manufacturer.

Also important is the SRAM array design. The SRAM layout can have an impact on if an error is correctable or not. For example, a single cosmic-induced soft error event may destroy the content of multiple bit cells in an array. If the destroyed bits are covered by the same codeword, the data cannot be corrected or possibly even detected. Therefore, the bits of each codeword should be physically spread in the array as far apart as feasibly possible. In a properly laid out SRAM array, multiple corrupted bits may result in several single-bit errors of different codewords which are correctable.

3.3 Memory Hierarchy

Table 3-1 summarizes the components of the SweRV EH1 memory hierarchy and their respective protection scheme.

Table 3-1 Memory Hierarchy Components and Protection

Memory Type	Abbreviation	Protection	Reason/Justification
Instruction Cache	I-cache	Parity or SEDDED ECC[9] (data and tag)	• Instructions can be refetched if error is detected
Instruction Closely-Coupled Memory	ICCM	SECDED ECC	• Large SRAM arrays • Data could be modified and is only valid copy
Data Closely-Coupled Memory	DCCM		
Core-complex-external Memories	SoC memories		

3.4 Error Detection and Handling

Table 3-2 summarizes the detection of errors, the recovery steps taken, and the logging of error events for each of the SweRV EH1 memories.

Note: Memories with parity or ECC protection must be initialized with correct parity or ECC. Otherwise, a read access to an uninitialized memory may report an error. The method of initialization depends on the organization and capabilities of the memory. Initialization might be performed by a memory self-test or depend on firmware to overwrite the entire memory range (e.g., via DMA accesses).

[9] Some highly reliable/available applications (e.g., automotive) might want to use an ECC-protected I-cache, instead of parity protection. Therefore, SEDDED ECC protection is optionally provided in SweRV EHX1 as well, selectable as a core build argument. Note that the I-cache area increases significantly if ECC protection is used.

Table 3-2 Error Detection, Recovery, and Logging

<table>
<tr><th rowspan="2">Memory Type</th><th rowspan="2">Detection</th><th colspan="2">Recovery</th><th colspan="2">Logging</th></tr>
<tr><th>Single-bit Error</th><th>Double-bit Error</th><th>Single-bit Error</th><th>Double-bit Error</th></tr>
<tr><td rowspan="4">I-cache</td><td rowspan="4">• Each 16-bit chunk of instructions protected with 1 parity bit or 5 ECC bits
• Each cache line tag protected with 1 parity bit or 5 ECC bits
• Parity/ECC bits checked in pipeline</td><td colspan="4">For parity:</td></tr>
<tr><td>• For instruction and tag parity errors, invalidate all cache lines of set
• Refetch cache line from SoC memory</td><td>Undetected</td><td>• Increment I-cache correctable error counter[10]
• If error counter has reached threshold, signal correctable error local interrupt (see Section 3.5.1)</td><td>No action</td></tr>
<tr><td colspan="4">For ECC:</td></tr>
<tr><td colspan="2">• For instruction and tag single- and double ECC errors, invalidate all cache lines of set
• Refetch cache line from SoC memory[11]</td><td colspan="2">• Increment I-cache correctable error counter[10]
• If error counter has reached threshold, signal correctable error local interrupt (see Section 3.5.1)</td></tr>
<tr><td rowspan="2">ICCM</td><td rowspan="2">• Each 32-bit chunk protected with 7 ECC bits
• ECC checked in pipeline</td><td>For fetches:
• Write corrected data/ECC back to ICCM
• Refetch instruction from ICCM[11]</td><td rowspan="2">Fatal error (uncorrectable)</td><td rowspan="2">• Increment ICCM single-bit error counter
• If error counter has reached threshold, signal correctable error local interrupt (see Section 3.5.2)</td><td>For fetches:
Instruction access fault exception</td></tr>
<tr><td>For DMA reads:
• Correct error in-line
• Write corrected data/ECC back to ICCM</td><td>For DMA reads:
Send error response on DMA slave bus to master</td></tr>
<tr><td rowspan="2">DCCM</td><td rowspan="2">• Each 32-bit chunk protected with 7 ECC bits
• ECC checked in pipeline</td><td rowspan="2">• Correct error in-line
• Write corrected data/ECC back to DCCM</td><td rowspan="2">Fatal error (uncorrectable)</td><td rowspan="2">• Increment DCCM single-bit error counter
• If error counter has reached threshold, signal</td><td>For loads:
Load access fault exception</td></tr>
<tr><td>For stores:
Store/AMO access fault exception</td></tr>
</table>

[10] It is unlikely, but possible that multiple I-cache parity/ECC errors are detected on a cache line in a single cycle, however, the I-cache single-bit error counter is incremented only by one.

[11] A RFPC (ReFetch PC) flush is performed since in-line correction would create timing issues and require an additional clock cycle as well as a different architecture.

Memory Type	Detection	Recovery		Logging	
		Single-bit Error	Double-bit Error	Single-bit Error	Double-bit Error
				correctable error local interrupt (see Section 3.5.3)	For DMA reads: Send error response on DMA slave bus to master
SoC memories	ECC checked at memory boundary	• Correct error • Send corrected data on bus • Write corrected data/ECC back to SRAM array	• Fatal error (uncorrectable) • Data sent on bus with error indication • Core must ignore sent data	• Increment SoC single-bit error counter local to memory • If error counter has reached threshold, signal external interrupt	For fetches: Instruction access fault exception
					For loads: Load access fault exception
					For stores: Store bus error NMI (see Section 2.6.1)

General comments:

- No address information of each individual correctable error is captured.
- Stuck-at bits would reach threshold relatively quickly. Use MBIST to determine exact location of the bad bit.

3.5 Core Error Counter/Threshold Registers

A summary of platform-specific core error counter/threshold control/status registers in CSR space:

- I-Cache Error Counter/Threshold Register (micect) (see Section 3.5.1)
- ICCM Correctable Error Counter/Threshold Register (miccmect) (see Section 3.5.2)
- DCCM Correctable Error Counter/Threshold Register (mdccmect) (see Section 3.5.3)

All read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

3.5.1 I-Cache Error Counter/Threshold Register (micect)

The `micect` register holds the I-cache error counter and its threshold. The *count* field of the `micect` register is incremented, if a parity/ECC error is detected on any of the cache line tags of the set or the instructions fetched from the I-cache. The *thresh* field of the `micect` register holds a pointer to a bit position of the *count* field. If the selected bit of the *count* field transitions from '0' to '1', a correctable error local interrupt (see Section 2.6.2) is signaled.

Hardware increments the *count* field on a detected error. Firmware can non-destructively read the current *count* and *thresh* values or write to both these fields (e.g., to change the threshold and reset the counter).

Note: The counter may overflow if not serviced and reset by firmware.

Note: The correctable error local interrupt is not latched (i.e., "sticky"), but it stays pending for a long period of time (i.e., 2^{32}-2^{thresh} errors). When firmware resets the counter, the correctable error local interrupt condition is cleared.

This register is mapped to the non-standard read/write CSR address space.

Table 3-3 I-Cache Error Counter/Threshold Register (micect, at CSR 0x7F0)

Field	Bits	Description	Access	Reset
thresh	31:27	I-cache parity/ECC error threshold: 0..26: Value *i* selects *count[i]* bit 27..31: Invalid (when written, mapped by hardware to 26)	R/W	0
count	26:0	Counter incremented if I-cache parity/ECC error(s) detected. If *count[thresh]* transitions from'0' to '1', signal correctable error local interrupt (see Section 2.6.2).	R/W	0

3.5.2 ICCM Correctable Error Counter/Threshold Register (miccmect)

The `miccmect` register holds the ICCM correctable error counter and its threshold. The *count* field of the `miccmect` register is incremented, if a correctable ECC error is detected on instructions fetched from the ICCM. The *thresh* field of the `miccmect` register holds a pointer to a bit position of the *count* field. If the selected bit of the *count* field transitions from '0' to '1', a correctable error local interrupt (see Section 2.6.2) is signaled.

Hardware increments the *count* field on a detected error. Firmware can non-destructively read the current *count* and *thresh* values or write to both these fields (e.g., to change the threshold and reset the counter).

Note: The counter may overflow if not serviced and reset by firmware.

Note: The correctable error local interrupt is not latched (i.e., "sticky"), but it stays pending for a long period of time (i.e., 2^{32}-2^{thresh} errors). When firmware resets the counter, the correctable error local interrupt condition is cleared.

Note: DMA accesses while in power management Sleep (pmu/fw-halt) or debug halt (db-halt) state may encounter ICCM single-bit errors. Correctable errors are counted in the `miccmect` error counter irrespective of the core's power state.

This register is mapped to the non-standard read/write CSR address space.

Table 3-4 ICCM Correctable Error Counter/Threshold Register (miccmect, at CSR 0x7F1)

Field	Bits	Description	Access	Reset
thresh	31:27	ICCM correctable ECC error threshold: 0..26: Value *i* selects *count[i]* bit 27..31: Invalid (when written, mapped by hardware to 26)	R/W	0
count	26:0	Counter incremented for each detected ICCM correctable ECC error. If *count[thresh]* transitions from'0' to '1', signal correctable error local interrupt (see Section 2.6.2).	R/W	0

3.5.3 DCCM Correctable Error Counter/Threshold Register (mdccmect)

The `mdccmect` register holds the DCCM correctable error counter and its threshold. The *count* field of the `mdccmect` register is incremented, if a correctable ECC error is detected on a read or write operation to the DCCM. The *thresh* field of the `mdccmect` register holds a pointer to a bit position of the *count* field. If the selected bit of the *count* field transitions from '0' to '1', a correctable error local interrupt (see Section 2.6.2) is signaled.

Hardware increments the *count* field on a detected error. Firmware can non-destructively read the current *count* and *thresh* values or write to both these fields (e.g., to change the threshold and reset the counter).

Note: The counter may overflow if not serviced and reset by firmware.

Note: The correctable error local interrupt is not latched (i.e., "sticky"), but it stays pending for a long period of time (i.e., 2^{32}-2^{thresh} errors). When firmware resets the counter, the correctable error local interrupt condition is cleared.

Note: DMA accesses while in power management Sleep (pmu/fw-halt) or debug halt (db-halt) state may encounter DCCM single-bit errors. Correctable errors are counted in the `mdccmect` error counter irrespective of the core's power state.

This register is mapped to the non-standard read/write CSR address space.

Table 3-5 DCCM Correctable Error Counter/Threshold Register (mdccmect, at CSR 0x7F2)

Field	Bits	Description	Access	Reset
thresh	31:27	DCCM correctable ECC error threshold: 0..26: Value *i* selects *count[i]* bit 27..31: Invalid (when written, mapped by hardware to 26)	R/W	0
count	26:0	Counter incremented for each detected DCCM correctable ECC error. If *count[thresh]* transitions from'0' to '1', signal correctable error local interrupt (see Section 2.6.2).	R/W	0

4 Power Management

This chapter specifies the power management functionality provided or supported by the SweRV EH1 core.

4.1 Features

SweRV EH1 power management supports and provides the following features:

- Support for three system-level power states: Active (C0), Sleep (C3), Power Off (C6)
- Firmware-initiated halt to enter sleep state
- Fine-grain clock gating in active state
- Enhanced clock gating in sleep state
- Halt/run control interface (to system-level Power Management Unit (PMU))
- Signal indicating that core is halted
- Signal indicating that core is in debug mode
- PAUSE feature to help avoid firmware spinning

4.2 Power States

From a system's perspective, the core may be placed in one of three power states: Active (C0), Sleep (C3), and Power Off (C6). Active and Sleep states require hardware support from the core, but in the Power Off state the core is power-gated so no special hardware support is needed.

Figure 4-1 depicts and Table 4-1 describes the core activity states as well as the events to transition between them.

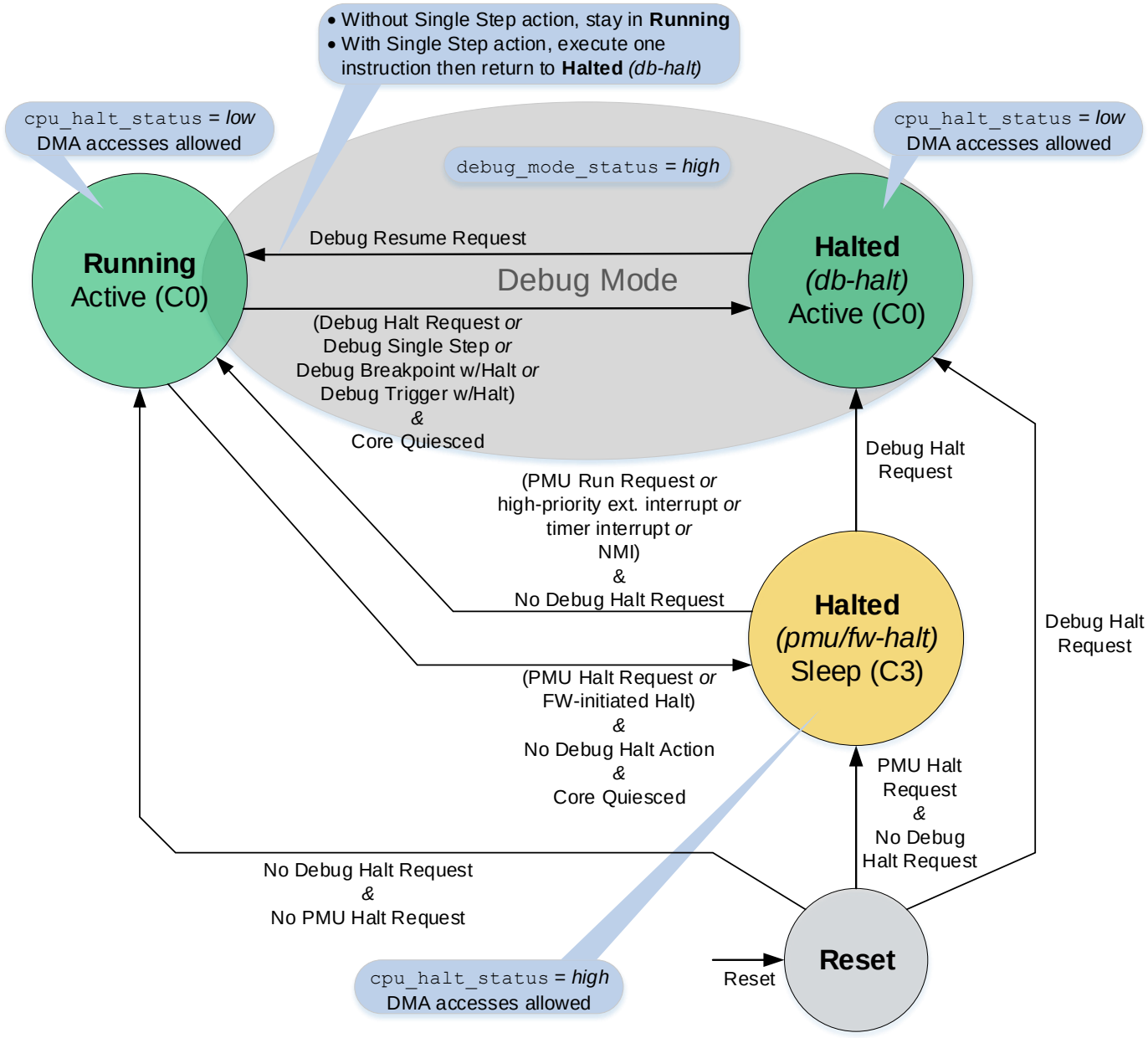


Figure 4-1 SweRV EH1 Core Activity States

Note: 'Core Quiesced' implies that no new instructions are executed and all outstanding bus transactions are completed (i.e., the store queue (SQ) and the write buffer (WB) are fully drained, the DMA FIFO is empty, and all outstanding I-cache misses are finished).

Table 4-1 Core Activity States

	Active (C0)		Sleep (C3)
	Running	Halted	
		db-halt	*pmu/fw-halt*
State Description	Core operating normally	Core halted in debug mode	Core halted by PMU halt request or by core firmware-initiated halt
Power Savings	Fine-grain clock gating integrated in core minimizes power consumption during regular operation	Fine-grain clock gating	Enhanced clock gating in addition to fine-grain clock gating
DMA Access	DMA accesses allowed		
State Indication	• `cpu_halt_status` is *low* • `debug_mode_status` is *low* (except for Debug Resume Request with Single Step action)	• `cpu_halt_status` is *low* • `debug_mode_status` is *high*	• `cpu_halt_status` is *high* • `debug_mode_status` is *low*
Machine Cycle Performance-Monitoring Counter	`mcycle` incremented every core clock cycle	Depends on *stopcount* bit in `dcsr` (Debug Control and Status Register) register: 0: `mcycle` incremented every core clock cycle 1: `mcycle` not incremented	`mcycle` not incremented

4.3 Power Control

The priority order of simultaneous halt requests is as follows:

1. Any debug halt action
    a. Debug halt request
    b. Debug single step
    c. Debug breakpoint
    d. Debug trigger
2. PMU halt request or core firmware-initiated halt

If the PMU sends a halt request while the core is in debug mode, the core disregards the halt request. If the PMU's halt request is still pending when the core exits debug mode, the request is honored at that time. Similarly, core firmware can't initiate a halt while in debug mode. However, it is not possible for a core firmware-initiated halt request to be pending when the core exits debug mode.

4.3.1 Debug Mode

Debug mode must be able to seize control of the core. Therefore, debug has higher priority than power control.

Debug mode is entered under any of the following conditions:

- Debug halt request
- Debug single step
- Debug breakpoint with halt action
- Debug trigger with halt action

Debug mode is exited with:

- Debug resume request with no single step action

The state 'db-halt' is the only halt state allowed while in debug mode.

4.3.2 Core Power Control Signals

Figure 4-2 depicts the power control and status signals which connect the SweRV EH1 core to the PMU. Signals from the PMU to the core are asynchronous and must be synchronized to the core clock domain. Similarly, signals from the core are asynchronous to the PMU clock domain and must be synchronized to the PMU's clock.

Note: The synchronizer of the `cpu_run_req` signal may not be clock-gated. Otherwise, the core may not be woken up again via the PMU interface.

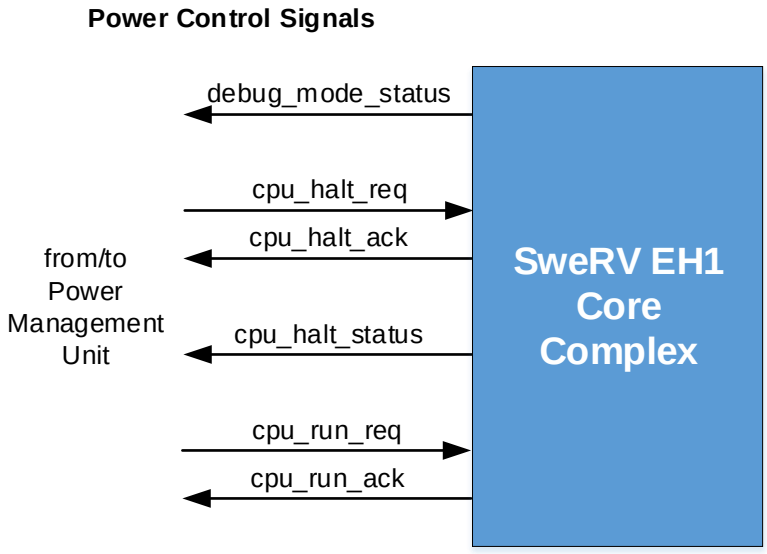


Figure 4-2 SweRV EH1 Power Control Signals

There are four types of signals between the Power Management Unit and the SweRV EH1 core, as described in Table 4-2. All signals are active-high.

Table 4-2 SweRV EH1 Power Control Signals

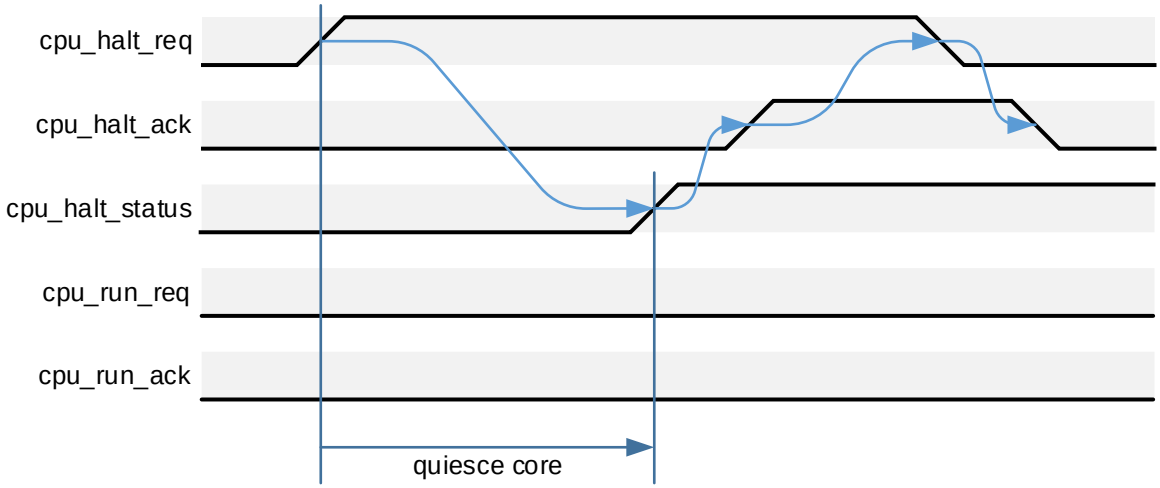
Signal(s)	Description
`cpu_halt_req` and `cpu_halt_ack`	Full handshake to request the core to halt. The PMU requests the core to halt by asserting the `cpu_halt_req` signal. The core is quiesced before halting. The core then asserts the `cpu_halt_ack` signal. When the PMU detects the asserted `cpu_halt_ack` signal, it deasserts the `cpu_halt_req` signal. Finally, when the core detects the deasserted `cpu_halt_req` signal, it deasserts the `cpu_halt_ack` signal.
`cpu_run_req` and `cpu_run_ack`	Full handshake to request the core to run. The PMU requests the core to run by asserting the `cpu_run_req` signal. The core exits the halt state and starts execution again. The core then asserts the `cpu_run_ack` signal. When the PMU detects the asserted `cpu_run_ack` signal, it deasserts the `cpu_run_req` signal. Finally, when the core detects the deasserted `cpu_run_req` signal, it deasserts the `cpu_run_ack` signal.
`cpu_halt_status`	Indication from the core to the PMU that the core has been gracefully halted.
`debug_mode_status`	Indication from the core to the PMU that the core is currently in debug mode. When the core is in debug mode, the PMU should refrain from sending a halt or run request.

Note: Power control protocol violations (e.g., simultaneously sending a run and a halt request) may lead to unexpected behavior.

Note: If the core is already in the activity state being requested (i.e., the core is already either in the pmu/fw-halt state and `cpu_halt_req` is asserted, or in the Running state and `cpu_run_req` is asserted), no acknowledgement is signaled (i.e., the `cpu_halt_ack` or `cpu_run_ack` signal, respectively, is not asserted).

Figure 4-3 depicts conceptual timing diagrams of a halt and a run request. Note that entering debug mode is an asynchronous event relative to power control commands sent by the PMU. Debug mode has higher priority and can interrupt and override PMU requests.

PMU Halt Request:



PMU Run Request:

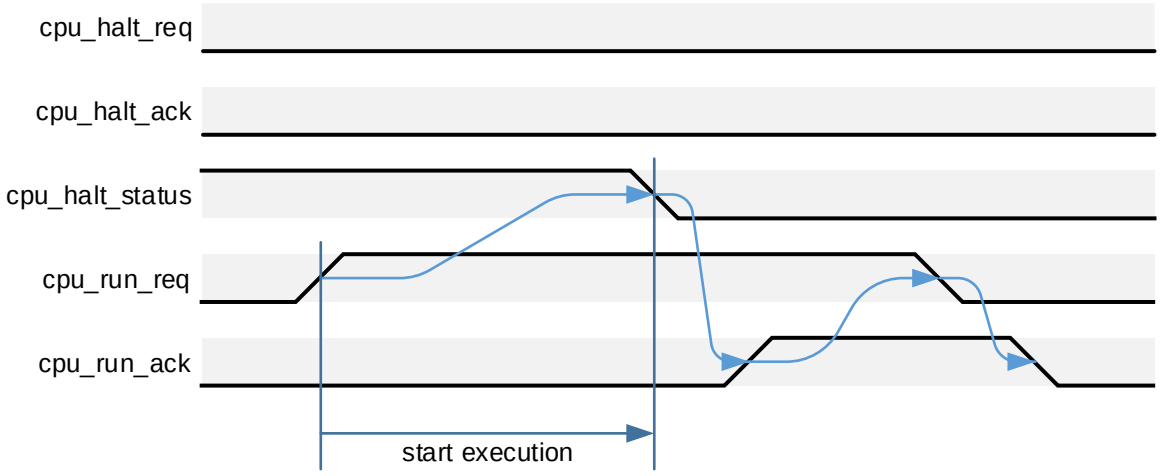


Figure 4-3 SweRV EH1 Power Control Interface Timing Diagrams

4.3.2.1 Core Wake-Up Events

When not in debug mode (i.e., the core is in pmu/fw-halt state), the core is woken up on several events:

- PMU run request
- High-priority external interrupt (`mhwakeup` signal from PIC) and core interrupts are enabled

- Timer interrupt
- Non-maskable interrupt (NMI) (`nmi_int` signal)

The PIC is part of the core logic and the `mhwakeup` signal is connected directly inside the core. The standard RISC-V timer interrupt and NMI signals are external to the core and originate in the SoC. If desired, these signals can be routed through the PMU and further qualified there.

4.3.3 Core Firmware-Initiated Halt

The firmware running on the core may also initiate a halt by writing a '1' to the *halt* field of the `mpmc` register (see Section 4.4.1). The core is quiesced before indicating that it has gracefully halted.

4.3.4 DMA Operations While Halted

When the core is halted in the 'pmu/fw-halt' or the 'db-halt' state, DMA operations are supported.

4.3.5 External Interrupts While Halted

All non-highest-priority external interrupts are temporarily ignored while halted. Only external interrupts which activate the `mhwakeup` signal (see Chapter 5) are honored, if the core is enabled to service external interrupts (i.e., the *mie* bit of the `mstatus` and the *meie* bit of the `mie` standard RISC-V registers are both set, otherwise the core remains in the 'pmu/fw-halt' state). External interrupts which are still pending and have a high enough priority to be signaled to the core are serviced once the core is back in the Running state.

4.4 Control/Status Registers

A summary of platform-specific control/status registers in CSR space:

- Power Management Control Register (mpmc) (see Section 4.4.1)
- Core Pause Control Register (mcpc) (see Section 4.4.2)

All reserved and unused bits in these control/status registers must be hardwired to '0'. Unless otherwise noted, all read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

4.4.1 Power Management Control Register (mpmc)

The `mpmc` register provides core power management control functionality. It allows the firmware running on the core to initiate a transition to the Halted (pmu/fw-halt) state.

The *halt* field of the `mpmc` register has W1R0 (Write 1, Read 0) behavior, as also indicated in the 'Access' column.

This register is mapped to the non-standard read/write CSR address space.

Table 4-3 Power Management Control Register (mpmc, at CSR 0x7C6)

Field	Bits	Description	Access	Reset
Reserved	31:1	Reserved	R	0
halt	0	Initiate core halt (i.e., transition to Halted (pmu/fw-halt) state) **Note:** Write ignored if in debug mode	R0/W1	0

4.4.2 Core Pause Control Register (mcpc)

The `mcpc` register supports functions to temporarily stop the core from executing instructions. This helps to save core power since busy-waiting loops can be avoided in the firmware.

PAUSE stops the core from executing instructions for a specified number[12] of clock ticks or until an interrupt is received.

Note: `PAUSE` is a long-latency, interruptible instruction and does not change the core's activity state (i.e., the core remains in the Running state). Therefore, even though this function may reduce core power, it is not part of core power management.

Note: `PAUSE` has a skid of several cycles. Therefore, instruction execution might not be stopped for precisely the number of cycles specified in the *pause* field of the `mcpc` register. However, this is acceptable for the intended use case of this function.

Note: If the PMU sends a halt request while PAUSE is still executing, the core enters the Halted (pmu/fw-halt) state and the *pause* clock counter stops until the core is back in the Running state.

Note: WFI is another candidate for a function that stops the core temporarily. Currently, the WFI instruction is implemented as NOP, which is a fully RISC-V-compliant option.

The *pause* field of the `mcpc` register has WAR0 (Write Any value, Read 0) behavior, as also indicated in the 'Access' column.

This register is mapped to the non-standard read/write CSR address space.

Table 4-4 Core Pause Control Register (mcpc, at CSR 0x7C2)

Field	Bits	Description	Access	Reset
pause	31:0	Pause execution for number of core clock cycles specified **Note:** *pause* is decremented by 1 for each core clock cycle. Execution continues either when *pause* is 0 or any interrupt is received.	R0/W	0

[12] The field width provided by the `mcpc` register allows to pause execution for about 4 seconds at a 1 GHz core clock.

5 External Interrupts

See *Chapter 7, Platform-Level Interrupt Controller (PLIC)* in [2] for general information.

Note: Even though this specification is modeled to a large extent after the RISC-V PLIC (Platform-Level Interrupt Controller) specification, this interrupt controller is associated with the core, not the platform. Therefore, the more general term PIC (Programmable Interrupt Controller) is used.

5.1 Features

The PIC provides these core-level external interrupt features:

- Up to 255 global (core-external) interrupt sources (from 1 (highest) to 255 (lowest)) with separate enable control for each source
- 15 priority levels (numbered 1 (lowest) to 15 (highest)), separately programmable for each interrupt source
- Programmable reverse priority order (14 (lowest) to 0 (highest))
- Programmable priority threshold to disable lower-priority interrupts
- Wake-up priority threshold (hardwired to highest priority level) to wake up core from power-saving (Sleep) mode if interrupts are enabled
- One interrupt target (RISC-V hart M-mode context)
- Support for vectored external interrupts
- Support for interrupt chaining and nested interrupts

5.2 Naming Convention

5.2.1 Unit, Signal, and Register Naming

***S* suffix:** Unit, signal, and register names which have an *S* suffix indicate an entity specific to an interrupt source.

***X* suffix:** Register names which have an *X* suffix indicate a consolidated register for multiple interrupt sources.

5.2.2 Address Map Naming

Control/status register: A control/status register mapped to either the memory or the CSR address space.

Memory-mapped register: Register which is mapped to RISC-V's 32-bit memory address space.

Register in CSR address space: Register which is mapped to RISC-V's 12-bit CSR address space.

5.3 Overview of Major Functional Units

5.3.1 External Interrupt Source

All functional units on the chip which generate interrupts to be handled by the RISC-V core are referred to as external interrupt sources. External interrupt sources indicate an interrupt request by sending an asynchronous signal to the PIC.

5.3.2 Gateway

Each external interrupt source connects to a dedicated gateway. The gateway is responsible for synchronizing the interrupt request to the core's clock domain, and for converting the request signal to a common interrupt request format (i.e., active-high and level-triggered) for the PIC. The PIC core can only handle one single interrupt request per interrupt source at a time.

All current SoC IP interrupts are asynchronous and level-triggered. Therefore, the gateway's only function for SoC IP interrupts is to synchronize the request to the core clock domain. There is no state kept in the gateway.

A gateway suitable for ASIC-external interrupts must provide programmability for interrupt type (i.e., edge- vs. level-triggered) as well as interrupt signal polarity (i.e., low-to-high vs. high-to-low transition for edge-triggered interrupts, active-high vs. -low for level-triggered interrupts). For edge-triggered interrupts, the gateway must latch the interrupt request in an interrupt pending (IP) flop to convert the edge- to a level-triggered interrupt signal. Firmware must clear the IP flop while handling the interrupt.

Note: For asynchronous interrupt sources, the pulse duration of an interrupt request must be at least two full clock cycles of the receiving (i.e., PIC core) clock domain to guarantee it will be recognized as an interrupt request. Shorter pulses might be dropped by the synchronizer circuit.

5.3.3 PIC Core

The PIC core's responsibility is to evaluate all pending and enabled interrupt requests and to pick the highest-priority request with the lowest interrupt source ID. It then compares this priority with a programmable priority threshold and, to support nested interrupts, the priority of the interrupt handler if one is currently running. If the picked request's priority is higher than both thresholds, it sends an interrupt notification to the core. In addition, it compares the picked request's priority with the wake-up threshold (highest priority level) and sends a wake-up signal to the core, if the priorities match. The PIC core also provides the interrupt source ID of the picked request in a status register.

Implementation Note: Different levels in the evaluation tree may be staged wherever necessary to meet timing, provided that all signals of a request (ID, priority, etc.) are equally staged.

5.3.4 Interrupt Target

The interrupt target is a specific RISC-V hart context. For the SweRV EH1 core, the interrupt target is the M privilege mode of the hart.

5.4 PIC Block Diagram

Figure 5-1 depicts a high-level view of the PIC. A simple gateway for asynchronous, level-triggered interrupt sources is shown in Figure 5-2, whereas Figure 5-3 depicts conceptually the internal functional blocks of a configurable gateway. Figure 5-4 shows a single comparator which is the building block to form the evaluation tree logic in the PIC core.

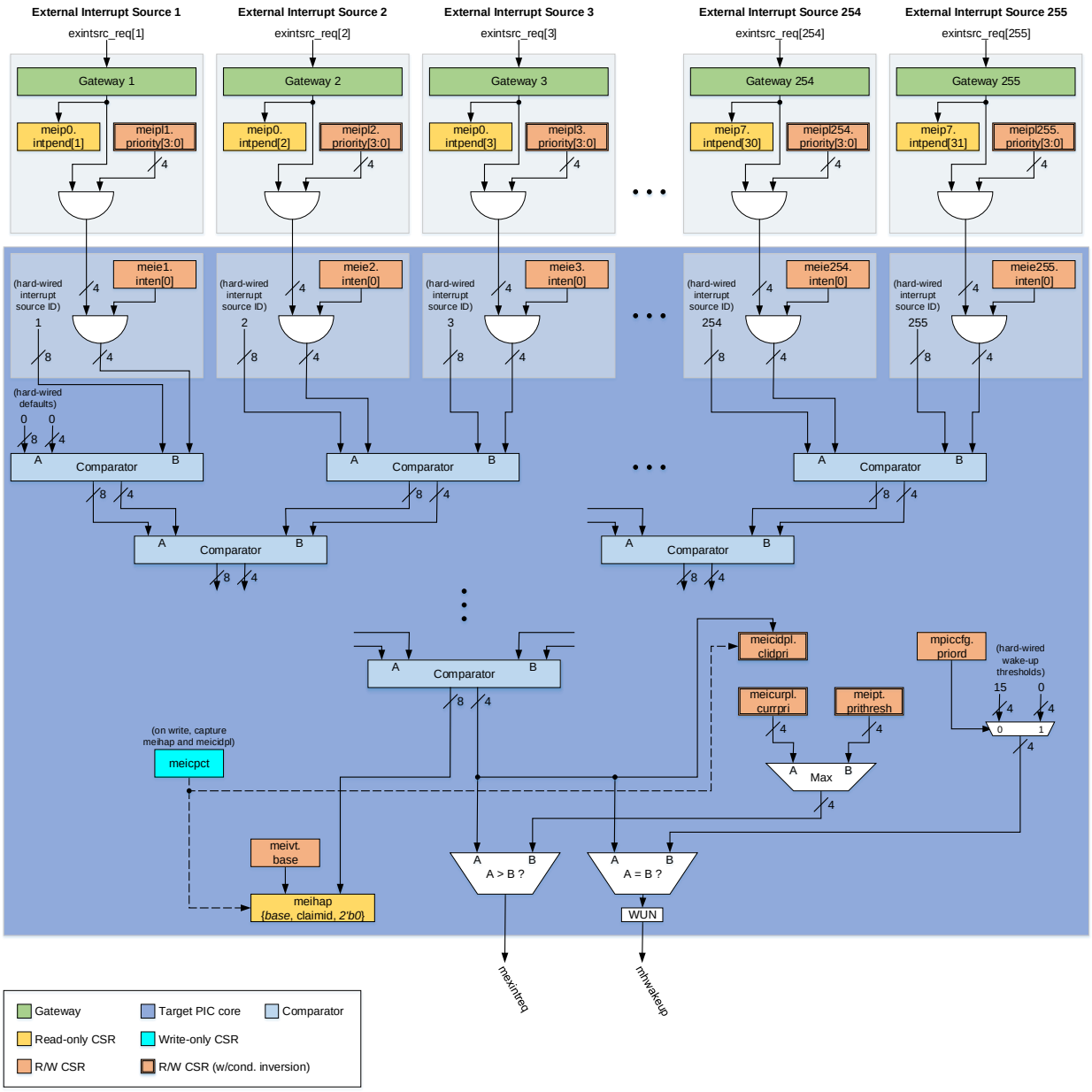


Figure 5-1 PIC Block Diagram

Implementation Note: For R/W control/status registers with double-borders in Figure 5-1, the outputs of the registers are conditionally bit-wise inverted, depending on the priority order set in the *priord* bit of the `mpiccfg` register. This is necessary to support the reverse priority order feature.

Note: The PIC logic always operates in regular priority order. When in reverse priority order mode, firmware reads and writes the control/status registers with reverse priority order values. The values written to and read from the control/status registers are inverted. Therefore, from the firmware's perspective, the PIC operates in reverse priority order.

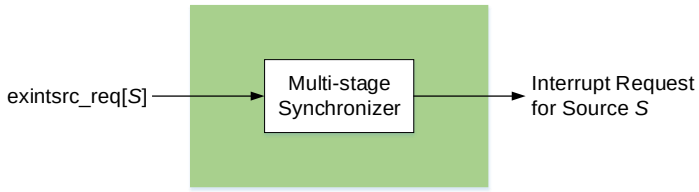


Figure 5-2 Gateway for Asynchronous, Level-triggered Interrupt Sources

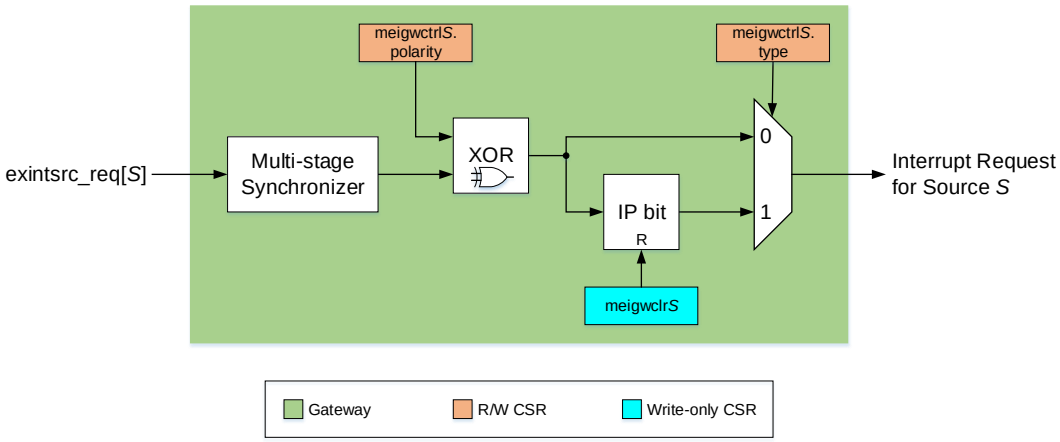


Figure 5-3 Conceptual Block Diagram of a Configurable Gateway

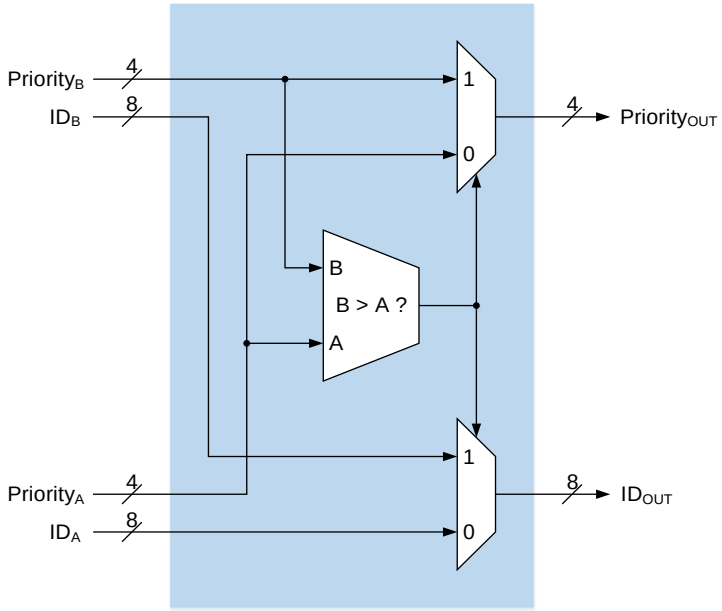


Figure 5-4 Comparator

5.5 Theory of Operation

5.5.1 Initialization

The control registers must be initialized in the following sequence:

1. Configure the priority order by writing the *priord* bit of the `mpiccfg` register.
2. For each configurable gateway *S*, set the polarity (*polarity* field) and type (*type* field) in the `meigwctrl`*S* register and clear the IP bit by writing to the gateway's `meigwclr`*S* register.
3. Set the base address of the external vectored interrupt address table by writing the *base* field of the `meivt` register.
4. Set the priority level for each external interrupt source *S* by writing the corresponding *priority* field of the `meipl`*S* registers.
5. Set the priority threshold by writing *prithresh* field of the `meipt` register.
6. Initialize the nesting priority thresholds by writing '0' (or '15' for reversed priority order) to the *clidpri* field of the `meicidpl` and the *currpri* field of the `meicurpl` registers.
7. Enable interrupts for the appropriate external interrupt sources by setting the *inten* bit of the `meie`*S* registers for each interrupt source *S*.

5.5.2 Regular Operation

A step-by-step description of interrupt control and delivery:

1. The external interrupt source *S* signals an interrupt request to its gateway by activating the corresponding `exintsrc_req[`*S*`]` signal.
2. The gateway synchronizes the interrupt request from the asynchronous interrupt source's clock domain to the PIC core clock domain (`pic_clk`).
3. For edge-triggered interrupts, the gateway also converts the request to a level-triggered interrupt signal by setting its internal interrupt pending (IP) bit.
4. The gateway then signals the level-triggered request to the PIC core by asserting its interrupt request signal.
5. The pending interrupt is visible to firmware by reading the corresponding *intpend* bit of the `meip`*X* register.
6. With the pending interrupt, the source's interrupt priority (indicated by the *priority* field of the `meipl`*S* register) is forwarded to the evaluation logic.
7. If the corresponding interrupt enable (i.e., *inten* bit of the `meie`*S* register is set), the pending interrupt's priority is sent to the input of the first-level 2-input comparator.
8. The priorities of a pair of interrupt sources are compared:
    a. If the two priorities are different, the higher priority and its associated hardwired interrupt source ID are forwarded to the second-level comparator.
    b. If the two priorities are the same, the priority and the lower hardwired interrupt source ID are forwarded to the second-level comparator.
9. Each subsequent level of comparators compares the priorities from two comparator outputs of the previous level:
    a. If the two priorities are different, the higher priority and its associated interrupt source ID are forwarded to the next-level comparator.
    b. If the two priorities are the same, the priority and the lower interrupt source ID are forwarded to the next-level comparator.
10. The output of the last-level comparator indicates the highest priority (maximum priority) and lowest interrupt source ID (interrupt ID) of all currently pending and enabled interrupts.
11. Maximum priority is compared to the higher of the two priority thresholds (i.e., *prithresh* field of the `meipt` and *currpri* field of the `meicurpl` registers):
    a. If maximum priority is higher than the two priority thresholds, the `mexintirq` signal is asserted.
    b. If maximum priority is the same as or lower than the two priority thresholds, the `mexintirq` signal is de-asserted.
12. The `mexintirq` signal's state is then reflected in the *meip* bit of the RISC-V hart's `mip` register.
13. In addition, maximum priority is compared to the wake-up priority level:
    a. If maximum priority is 15 (or 0 for reversed priority order), the wake-up notification (WUN) bit is set.
    b. If maximum priority is lower than 15 (or 0 for reversed priority order), the wake-up notification (WUN) bit is not set.
14. The WUN state is indicated to the target hart with the `mhwakeup` signal[13].
15. When the target hart takes the external interrupt, it disables all interrupts (i.e., clears the *mei* bit of the RISC-V hart's `mstatus` register) and jumps to the external interrupt handler.

[13] Note that the core is only woken up from the power management Sleep (pmu/fw-halt) state if the *mie* bit of the `mstatus` and the *meie* bit of the `mie` standard RISC-V registers are both set.

16. The external interrupt handler writes to the `meicpct` register to trigger the capture of the interrupt source ID of the currently highest-priority pending external interrupt (in the `meihap` register) and its corresponding priority (in the `meicidpl` register). Note that the captured content of the *claimid* field of the `meihap` register and its corresponding priority in the `meicidpl` register is neither affected by the priority thresholds (*prithresh* field of the `meipt` and *currpri* field of the `meicurpl` registers) nor by the core's external interrupt enable bit (*meie* bit of the RISC-V hart's `mie` register).
17. The handler then reads the `meihap` register to obtain the interrupt source ID provided in the *claimid* field. Based on the content of the `meihap` register, the external interrupt handler jumps to the handler specific to this external interrupt source.
18. The source-specific interrupt handler services the external interrupt, and then:
    a. For level-triggered interrupt sources, the interrupt handler clears the state in the SoC IP which initiated the interrupt request.
    b. For edge-triggered interrupt sources, the interrupt handler clears the IP bit in the source's gateway by writing to the `meigwclr`*S* register.
19. The clearing de-asserts the source's interrupt request to the PIC core and stops this external interrupt source from participating in the highest priority evaluation.
20. In the background, the PIC core continuously evaluates the next pending interrupt with highest priority and lowest interrupt source ID:
    a. If there are other interrupts pending, enabled, and with a priority level higher than *prithresh* field of the `meipt` and *currpri* field of the `meicurpl` registers, `mexintirq` stays asserted.
    b. If there are no further interrupts pending, enabled, and with a priority level higher than *prithresh* field of the `meipt` and *currpri* field of the `meicurpl` registers, `mexintirq` is de-asserted.
21. Firmware may update the content of the `meihap` and `meicidpl` registers by writing to the `meicpct` register to trigger a new capture.

5.6 Support for Vectored External Interrupts

Note: The RISC-V standard defines support for vectored interrupts down to an interrupt class level (i.e., timer, software, and external interrupts for each privilege level), but not to the granularity of individual external interrupt sources (as described in this section). The two mechanisms are intendent of each other and should be used together for lowest interrupt latency. For more information on the standard RISC-V vectored interrupt support, see Section 3.1.12 in [2].

The SweRV EH1 PIC implementation provides support for vectored external interrupts. The content of the `meihap` register is a full 32-bit pointer to the specific vector to the handler of the external interrupt source which needs service. This pointer consists of a 22-bit base address (*base*) of the external interrupt vector table, the 8-bit claim ID (*claimid*), and a 2-bit '0' field. The *claimid* field is adjusted with 2 bits of zeros to construct the offset into the vector table containing 32-bit vectors. The external interrupt vector table resides either in the DCCM, SoC memory, or a dedicated flop array in the core.

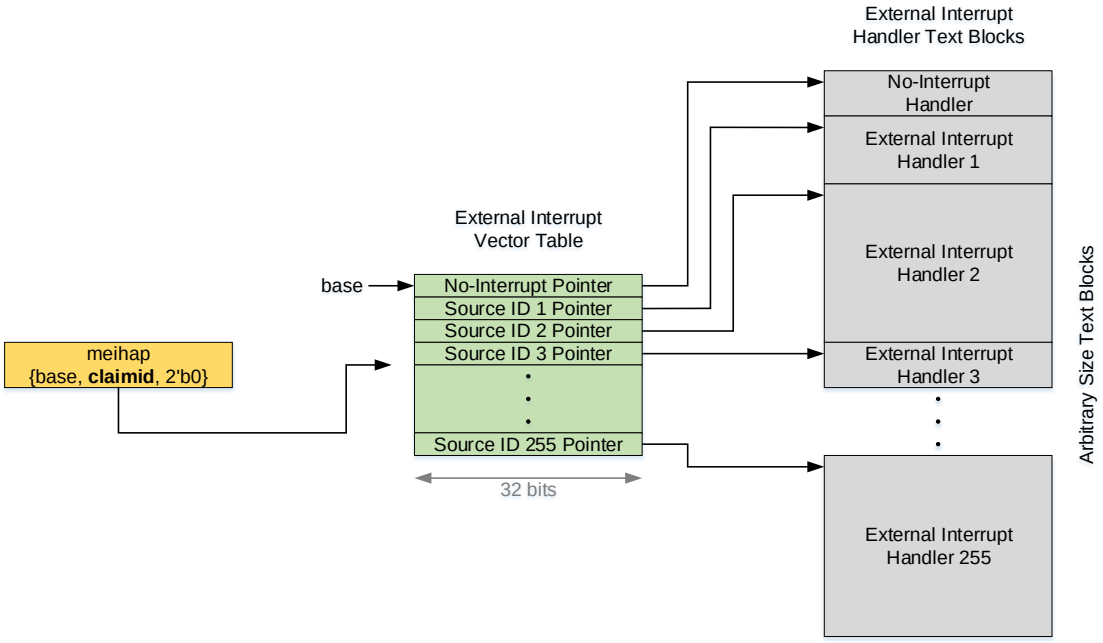


Figure 5-5 Vectored External Interrupts

Figure 5-5 depicts the steps from taking the external interrupt to starting to execute the interrupt source specific handler. When the core takes an external interrupt, the initiated external interrupt handler executes the following operations:

1. Save register(s) used in this handler to the `mscratch` register[14] or on the stack
2. Store to the `meicpct` control/status register to capture a consistent claim ID / priority level pair
3. Load the `meihap` control/status register into regX
4. Load memory location at address in regX into regY
5. Jump to address in regY (i.e., start executing the interrupt source-specific handler)

Note: Two registers (regX and regY) are shown above for clarification only. The same register can actually be used.

It is possible in some corner cases that the captured claim ID read from the `meihap` register is 0 (i.e., no interrupt request is pending). To keep the interrupt latency at a minimum, the external interrupt handler above should not check for this condition. Instead, the pointer stored at the base address of the external interrupt vector table (i.e., pointer 0) must point to a 'no-interrupt' handler, as shown in Figure 5-5 above. That handler can be as simple as executing a return from interrupt (i.e., `mret`) instruction.

Note that it is possible for multiple interrupt sources to share the same interrupt handler by populating their respective interrupt vector table entries with the same pointer to that handler.

5.6.1 Full Hardware Implementation of Vectored External Interrupts

If the mechanism described in the previous section still incurs too much latency or has too much impact on performance, implementing vectored external interrupts fully in hardware would be possible as well.

The *claimid* can be used to select the interrupt vector associated with the selected highest-priority source ID. When the core takes an external interrupt, it would start fetching directly from the address indicated by the interrupt vector selected by *claimid*.

Implementation Note: The external interrupt vector table can either be implemented in flops or mapped to SRAM memory. If implemented as flops, the address is MUX-ed out of the array based on the *claimid*. If mapped to SRAM, the core issues a dummy load instruction to a *claimid*-dependent memory addresses to retrieve the interrupt vector.

[14] If saved to `mscratch` register, the interrupt source-specific handler must save it from there to the stack before re-enabling interrupts.

5.7 Interrupt Chaining

Figure 5-6 depicts the concept of chaining interrupts. The goal of chaining is to reduce the overhead of pushing and popping state to and from the stack while handling a series of Interrupt Service Routines (ISR) of the same priority level. The first ISR of the chain saves the state common to all interrupt handlers of this priority level to the stack and then services its interrupt. If this handler needs to save additional state, it does so immediately after saving the common state and then restores only the additional state when done. At the end of the handler routine, the ISR writes to the `meicpct` register to capture the latest interrupt evaluation result, then reads the `meihap` register to determine if any other interrupts of the same priority level are pending. If no, it restores the state from the stack and exits. If yes, it immediately jumps into the next interrupt handler skipping the restoring of state in the finished handler as well as the saving of the same state in the next handler. The chaining continues until no other ISRs of the same priority level are pending, at which time the last ISR of the chain restores the original state from the stack again.

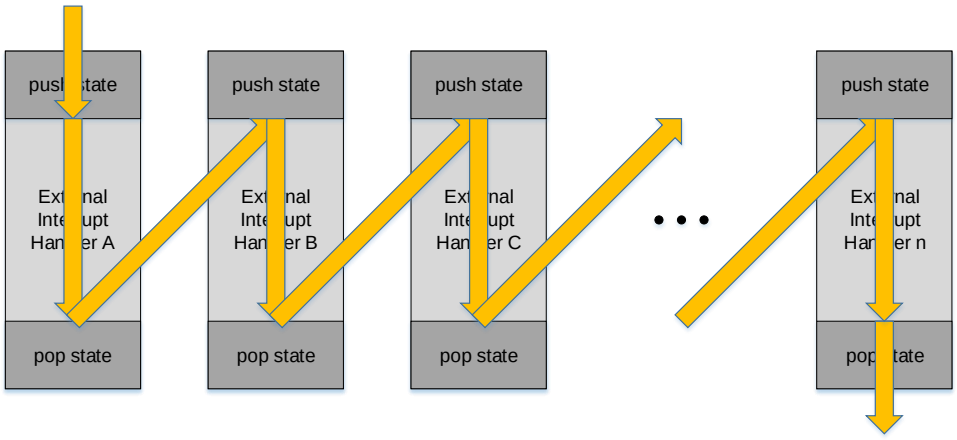


Figure 5-6 Concept of Interrupt Chaining

5.8 Interrupt Nesting

Support for multiple levels of nested interrupts helps to provide a more deterministic interrupt latency at higher priority levels. To achieve this, a running interrupt handler with lower priority must be preemptable by a higher-priority interrupt. The state of the preempted handler is saved before the higher priority interrupt is executed, so that it can continue its execution at the point it was interrupted.

SweRV EH1 and its PIC provide supported for up to 15 nested interrupts, one interrupt handler at each priority level. The conceptual steps of nesting are:

1. The external interrupt is taken as described in step 15. of Section 5.5.2 *Regular Operation*. When the core takes the external interrupt, it automatically disables all interrupts.
2. The external interrupt handler executes the following steps to get into the source-specific interrupt handler, as described in Section 5.6:

```
st meicpct    // atomically captures winning claim ID and priority level
ld meihap     // get pointer to interrupt handler starting address
ld isr_addr   // load interrupt handler starting address
jmp isr_addr  // jump to source-specific interrupt handler
```

3. The source-specific interrupt handler then saves the state of the code it interrupted (including the priority level in case it was an interrupt handler) to the stack, sets the priority threshold to its own priority, and then reenables interrupts:

```
push mepc, mstatus, mie, …
push meicurpl              // save interrupted code's priority level
ld meicidpl                // read interrupt handler's priority level
st meicurpl                // change threshold to handler's priority
mstatus.mei=1              // reenable interrupts
```

4. Any external interrupt with a higher priority can now safely preempt the currently executing interrupt handler.

5. Once the interrupt handler finished its task, it disables any interrupts and restores the state of the code it interrupted:

```
mstatus.mei=0              // disable all interrupts
pop meicurpl               // get interrupted code's priority level
st meicurpl                // set threshold to previous priority
pop mepc, mstatus, mie, …
mret                       // return from interrupt, reenable interrupts
```

6. The interrupted code continues to execute.

5.9 Performance Targets

The target latency through the PIC, including the clock domain crossing latency incurred by the gateway, is 4 core clock cycles.

5.10 Configurability

Typical implementations require fewer than 255 external interrupt sources. Code should only be generated for functionality needed by the implementation.

5.10.1 Rules

- The IDs of external interrupt sources must start at 1, and be contiguous.
- All unused register bits must be hardwired to '0'.

5.10.2 Build Arguments

The PIC build arguments are:

- **PIC base address for memory-mapped control/status registers (PIC_base_addr)**
  - See Section 14.1.2
- **Number of external interrupt sources**
  - Total interrupt sources (RV_PIC_TOTAL_INT): 2..255

5.10.3 Impact on Generated Code

5.10.3.1 External Interrupt Sources

The number of required external interrupt sources has an impact on the following:

- General impact:
  - Signal pins:
    - `exintsrc_req[`*`S`*`]`
  - Registers:
    - `meipl`*`S`*
    - `meip`*`X`*
  - Logic:
    - Gateway *S*
- Target PIC core impact:
  - Registers:
    - `meie`*`S`*
  - Logic:
    - Gating of priority level with interrupt enable
    - Number of first-level comparators
    - Unnecessary levels of the comparator tree

5.10.3.2 Further Optimizations

Register fields, bus widths, and comparator MUXs are sized to cover the maximum external interrupt source IDs of 255. For approximately every halving of the number of interrupt sources, it would be possible to reduce the number of register fields holding source IDs, bus widths carrying source IDs, and source ID MUXs in the comparators by one. However, the overall reduction in logic is quite small, so it might not be worth the effort.

5.11 PIC Control/Status Registers

A summary of the PIC control/status registers in CSR address space:

- External Interrupt Priority Threshold Register (meipt) (see Section 5.11.5)
- External Interrupt Vector Table Register (meivt) (see Section 5.11.6)
- External Interrupt Handler Address Pointer Register (meihap) (see Section 5.11.7)
- External Interrupt Claim ID / Priority Level Capture Trigger Register (meicpct) (see Section 5.11.8)
- External Interrupt Claim ID's Priority Level Register (meicidpl) (see Section 5.11.9)
- External Interrupt Current Priority Level Register (meicurpl) (see Section 5.11.10)

A summary of the PIC memory-mapped control/status registers:

- PIC Configuration Register (mpiccfg) (see Section 5.11.1)
- External Interrupt Priority Level Registers (meipl*S*) (see Section 5.11.2)
- External Interrupt Pending Registers (meip*X*) (see Section 5.11.3)
- External Interrupt Enable Registers (meie*S*) (see Section 5.11.4)
- External Interrupt Gateway Configuration Registers (meigwctrl*S*) (see Section 5.11.11)
- External Interrupt Gateway Clear Registers (meigwclr*S*) (see Section 5.11.12)

All reserved and unused bits in these control/status registers must be hardwired to '0'. Unless otherwise noted, all read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

Note: All memory-mapped register writes must be followed by a `fence` instruction to enforce ordering and synchronization.

Note: All memory-mapped control/status register accesses must be word-sized and word-aligned. Non-word sized/aligned loads cause a load access fault exception, and non-word sized/aligned stores cause a store/AMO access fault exception.

Note: Accessing unused addresses within the 32KB PIC address range do not trigger an unmapped address exception. Reads to unmapped addresses return 0, writes to unmapped addresses are silently dropped.

5.11.1 PIC Configuration Register (mpiccfg)

The PIC configuration register is used to select the operational parameters of the PIC.

This 32-bit register is an idempotent memory-mapped control register.

Table 5-1 PIC Configuration Register (mpiccfg, at PIC_base_addr+0x3000)

Field	Bits	Description	Access	Reset
Reserved	31:1	Reserved	R	0
priord	0	Priority order: 0: RISC-V standard compliant priority order (0=lowest to 15=highest) 1: Reverse priority order (15=lowest to 0=highest)	R/W	0

5.11.2 External Interrupt Priority Level Registers (meipl*S*)

There are 255 priority level registers, one for each external interrupt source. Implementing individual priority level registers allows a debugger to autonomously discover how many priority level bits are supported for this interrupt source. Firmware must initialize the priority level for each used interrupt source. Firmware may also read the priority level.

Implementation Note: The read and write paths between the core and the `meipl`*`S`* registers must support direct and inverted accesses, depending on the priority order set in the *priord* bit of the `mpiccfg` register. This is necessary to support the reverse priority order feature.

These 32-bit registers are idempotent memory-mapped control registers.

Table 5-2 External Interrupt Priority Level Register *S=1..255* (meipl*S*, at PIC_base_addr+*S4)**

Field	Bits	Description	Access	Reset
Reserved	31:4	Reserved	R	0
priority	3:0	External interrupt priority level for interrupt source ID *S*: RISC-V standard compliant priority order: 0: Never interrupt 1..15: Interrupt priority level (1 is lowest, 15 is highest) Reverse priority order: 15: Never interrupt 14..0: Interrupt priority level (14 is lowest, 0 is highest)	R/W	0

5.11.3 External Interrupt Pending Registers (meip*X*)

Eight external interrupt pending registers are needed to report the current status of up to 255 independent external interrupt sources. Each bit of these registers corresponds to an interrupt pending indication of a single external interrupt source. These registers only provide the status of pending interrupts and cannot be written.

These 32-bit registers are idempotent memory-mapped status registers.

Table 5-3 External Interrupt Pending Register *X=0..7* (meip*X*, at PIC_base_addr+0x1000+*X4)**

Field	Bits	Description	Access	Reset
X = 0, *Y* = 1..31 and *X* = 1..7, *Y* = 0..31				
intpend	*Y*	External interrupt pending for interrupt source ID *X**32+*Y*: 0: Interrupt not pending 1: Interrupt pending	R	0
X = 0, *Y* = 0				
Reserved	0	Reserved	R	0

5.11.4 External Interrupt Enable Registers (meie*S*)

Each of the up to 255 independently controlled external interrupt sources has a dedicated interrupt enable register. Separate registers per interrupt source were chosen for ease-of-use and compatibility with existing controllers.

(**Note:** Not packing together interrupt enable bits as bit vectors results in context switching being a more expensive operation.)

These 32-bit registers are idempotent memory-mapped control registers.

Table 5-4 External Interrupt Enable Register *S=1..255* (meie*S*, at PIC_base_addr+0x2000+*S4)**

Field	Bits	Description	Access	Reset
Reserved	31:1	Reserved	R	0
inten	0	External interrupt enables for interrupt source ID *S:* 0: Interrupt disabled 1: Interrupt enabled	R/W	0

5.11.5 External Interrupt Priority Threshold Register (meipt)

The `meipt` register is used to set the interrupt target's priority threshold. Interrupt notifications are sent to a target only for external interrupt sources with a priority level strictly higher than this target's threshold. Hosting the threshold in a separate register allows a debugger to autonomously discover how many priority threshold level bits are supported.

Implementation Note: The read and write paths between the core and the `meipt` register must support direct and inverted accesses, depending on the priority order set in the *priord* bit of the `mpiccfg` register. This is necessary to support the reverse priority order feature.

This 32-bit register is mapped to the non-standard read/write CSR address space.

Table 5-5 External Interrupt Priority Threshold Register (meipt, at CSR 0xBC9)

Field	Bits	Description	Access	Reset
Reserved	31:4	Reserved	R	0
prithresh	3:0	External interrupt priority threshold: RISC-V standard compliant priority order: 0: No interrupts masked 1..14: Mask interrupts with priority strictly lower than or equal to this threshold 15: Mask all interrupts Reverse priority order: 15: No interrupts masked 14..1: Mask interrupts with priority strictly lower than or equal to this threshold 0: Mask all interrupts	R/W	0

5.11.6 External Interrupt Vector Table Register (meivt)

The `meivt` register is used to set the base address of the external vectored interrupt address table. The value written to the *base* field of the `meivt` register appears in the *base* field of the `meihap` register.

This 32-bit register is mapped to the non-standard read-write CSR address space.

Table 5-6 External Interrupt Vector Table Register (meivt, at CSR 0xBC8)

Field	Bits	Description	Access	Reset
base	31:10	Base address of external interrupt vector table	R/W	0
Reserved	9:0	Reserved	R	0

5.11.7 External Interrupt Handler Address Pointer Register (meihap)

The `meihap` register provides a pointer into the vectored external interrupt table for the highest-priority pending external interrupt. The winning claim ID is captured in the *claimid* field of the `meihap` register when firmware writes to the `meicpct` register to claim an external interrupt. The priority level of the external interrupt source corresponding to the *claimid* field of this register is simultaneously captured in the *clidpri* field of the `meicidpl` register. Since the PIC core is constantly evaluating the currently highest-priority pending interrupt, this mechanism provides a consistent snapshot of the highest-priority source requesting an interrupt and its associated priority level. This is important to support nested interrupts.

The meihap register contains the full 32-bit address of the pointer to the starting address of the specific interrupt handler for this external interrupt source. The external interrupt handler then loads the interrupt handler's starting address and jumps to that address.

Alternatively, the external interrupt source ID indicated by the *claimid* field of the meihap register may be used by the external interrupt handler to calculate the address of the interrupt handler specific to this external interrupt source.

Implementation Note: The *base* field in the meihap register reflects the current value of the *base* field in the meivt register. I.e., *base* is not stored in the meihap register.

This 32-bit register is mapped to the non-standard read-only CSR address space.

Table 5-7 External Interrupt Handler Address Pointer Register (meihap, at CSR 0xFC8)

Field	Bits	Description	Access	Reset
base	31:10	Base address of external interrupt vector table (i.e., *base* field of meivt register)	R	0
claimid	9:2	External interrupt source ID of highest-priority pending interrupt (i.e., lowest source ID with highest priority)	R	0
00	1:0	Must read as '00'	R	0

5.11.8 External Interrupt Claim ID / Priority Level Capture Trigger Register (meicpct)

The meicpct register is used to trigger the simultaneous capture of the currently highest-priority interrupt source ID (in the *claimid* field of the meihap register) and its corresponding priority level (in the *clidpri* field of the meicidpl register) by writing to this register. Since the PIC core is constantly evaluating the currently highest-priority pending interrupt, this mechanism provides a consistent snapshot of the highest-priority source requesting an interrupt and its associated priority level. This is important to support nested interrupts.

The meicpct register has WAR0 (Write Any value, Read 0) behavior. Writing '0' is recommended.

Implementation Note: The meicpct register does not have any physical storage elements associated with it. It is write-only and solely serves as the trigger to simultaneously capture the winning claim ID and corresponding priority level.

This 32-bit register is mapped to the non-standard read/write CSR address space.

Table 5-8 External Interrupt Claim ID / Priority Level Capture Trigger Register (meicpct, at CSR 0xBCA)

Field	Bits	Description	Access	Reset
Reserved	31:0	Reserved	R0/WA	0

5.11.9 External Interrupt Claim ID's Priority Level Register (meicidpl)

The meicidpl register captures the priority level corresponding to the interrupt source indicated in the *claimid* field of the meihap register when firmware writes to the meicpct register. Since the PIC core is constantly evaluating the currently highest-priority pending interrupt, this mechanism provides a consistent snapshot of the highest-priority source requesting an interrupt and its associated priority level. This is important to support nested interrupts.

Implementation Note: The read and write paths between the core and the meicidpl register must support direct and inverted accesses, depending on the priority order set in the *priord* bit of the mpiccfg register. This is necessary to support the reverse priority order feature.

This 32-bit register is mapped to the non-standard read/write CSR address space.

Table 5-9 External Interrupt Claim ID's Priority Level Register (meicidpl, at CSR 0xBCB)

Field	Bits	Description	Access	Reset
Reserved	31:4	Reserved	R	0
clidpri	3:0	Priority level of preempting external interrupt source (corresponding to source ID read from *claimid* field of `meihap` register)	R/W	0

5.11.10 External Interrupt Current Priority Level Register (meicurpl)

The `meicurpl` register is used to set the interrupt target's priority threshold for nested interrupts. Interrupt notifications are signaled to the core only for external interrupt sources with a priority level strictly higher than the thresholds indicated in this register and the `meipt` register.

The `meicurpl` register is written by firmware, and not updated by hardware. The interrupt handler should read its own priority level from the *clidpri* field of the `meicidpl` register and write it to the *currpri* field of the `meicurpl` register. This avoids potentially being interrupted by another interrupt request with lower or equal priority once interrupts are reenabled.

Note: Providing the `meicurpl` register in addition to the `meipt` threshold register enables an interrupt service routine to temporarily set the priority level threshold to its own priority level. Therefore, only new interrupt requests with a strictly higher priority level are allowed to preempt the current handler, without modifying the longer-term threshold set by firmware in the `meipt` register.

Implementation Note: The read and write paths between the core and the `meicurpl` register must support direct and inverted accesses, depending on the priority order set in the *priord* bit of the `mpiccfg` register. This is necessary to support the reverse priority order feature.

This 32-bit register is mapped to the non-standard read/write CSR address space.

Table 5-10 External Interrupt Current Priority Level Register (meicurpl, at CSR 0xBCC)

Field	Bits	Description	Access	Reset
Reserved	31:4	Reserved	R	0
currpri	3:0	Priority level of current interrupt service routine (managed by firmware)	R/W	0

5.11.11 External Interrupt Gateway Configuration Registers (meigwctrl*S*)

Each configurable gateway has a dedicated configuration register to control the interrupt type (i.e., edge- vs. level-triggered) as well as the interrupt signal polarity (i.e., low-to-high vs. high-to-low transition for edge-triggered interrupts, active-high vs. -low for level-triggered interrupts).

Note: A register is only present for interrupt source *S* if a configurable gateway is instantiated.

These 32-bit registers are idempotent memory-mapped control registers.

Table 5-11 External Interrupt Gateway Configuration Register *S=1..255* (meigwctrl*S*, at PIC_base_addr+0x4000+*S4)**

Field	Bits	Description	Access	Reset
Reserved	31:2	Reserved	R	0
type	1	External interrupt type for interrupt source ID *S:* 0: Level-triggered interrupt 1: Edge-triggered interrupt	R/W	0

Field	Bits	Description	Access	Reset
polarity	0	External interrupt polarity for interrupt source ID *S:* 0: Active-high interrupt 1: Active-low interrupt	R/W	0

5.11.12 External Interrupt Gateway Clear Registers (meigwclr*S*)

Each configurable gateway has a dedicated clear register to reset its interrupt pending (IP) bit. For edge-triggered interrupts, firmware must clear the gateway's IP bit while servicing the external interrupt of source ID *S* by writing to the `meigwclr`*S* register.

Note: A register is only present for interrupt source *S* if a configurable gateway is instantiated.

The `meigwclr`*S* register has WAR0 (Write Any value, Read 0) behavior. Writing '0' is recommended.

Implementation Note: The `meigwclr`*S* register does not have any physical storage elements associated with it. It is write-only and solely serves as the trigger to clear the interrupt pending (IP) bit of the configurable gateway *S*.

These 32-bit registers are idempotent memory-mapped control registers.

Table 5-12 External Interrupt Gateway Clear Register *S=1..255* (meigwclr*S*, at PIC_base_addr+0x5000+*S4)**

Field	Bits	Description	Access	Reset
Reserved	31:0	Reserved	R0/WA	0

5.12 PIC CSR Address Map

Table 5-13 summarizes the PIC non-standard RISC-V CSR address map.

Table 5-13 PIC Non-standard RISC-V CSR Address Map

Number	Privilege	Name	Description
0xBC8	MRW	meivt	External interrupt vector table register
0xBC9	MRW	meipt	External interrupt priority threshold register
0xBCA	MRW	meicpct	External interrupt claim ID / priority level capture trigger register
0xBCB	MRW	meicidpl	External interrupt claim ID's priority level register
0xBCC	MRW	meicurpl	External interrupt current priority level register
0xFC8	MRO	meihap	External interrupt handler address pointer register

5.13 PIC Memory-mapped Register Address Map

Table 5-14 summarizes the PIC memory-mapped register address map.

Table 5-14 PIC Memory-mapped Register Address Map

Address Offset from PIC_base_addr		Name	Description
Start	End		
+ 0x0000	+ 0x0003	Reserved	Reserved

Address Offset from PIC_base_addr		Name	Description
Start	**End**		
+ 0x0004	+ 0x0004 + S_{max}*4-1	meipl*S*	External interrupt priority level register
+ 0x0004 + S_{max}*4	+ 0x0FFF	Reserved	Reserved
+ 0x1000	+ 0x1000 + (X_{max}+1)*4-1	meip*X*	External interrupt pending register
+ 0x1000 + (X_{max}+1)*4	+ 0x1FFF	Reserved	Reserved
+ 0x2000	+ 0x2003	Reserved	Reserved
+ 0x2004	+ 0x2004 + S_{max}*4-1	meie*S*	External interrupt enable register
+ 0x2004 + S_{max}*4	+ 0x2FFF	Reserved	Reserved
+ 0x3000	+ 0x3003	mpiccfg	External interrupt PIC configuration register
+ 0x3004	+ 0x3FFF	Reserved	Reserved
+ 0x4000	+ 0x4003	Reserved	Reserved
+ 0x4004	+ 0x4004 + S_{max}*4-1	meigwctrl*S*	External interrupt gateway configuration register (for configurable gateways only)
+ 0x4004 + S_{max}*4	+ 0x4FFF	Reserved	Reserved
+ 0x5000	+ 0x5003	Reserved	Reserved
+ 0x5004	+ 0x5004 + S_{max}*4-1	meigwclr*S*	External interrupt gateway clear register (for configurable gateways only)
+ 0x5004 + S_{max}*4	+ 0x7FFF	Reserved	Reserved

Note: X_{max} = (S_{max} + 31) // 32, whereas // is an integer division ignoring the remainder

5.14 Interrupt Enable/Disable Code Samples

5.14.1 Example Interrupt Flows

- Macro flow to enable interrupt source id 5 with priority set to 7, threshold set to 1, and gateway configured for edge-triggered/active-low interrupt source:

```
disable_ext_int        // Disable interrupts (MIE[meip]=0)
set_threshold 1        // Program global threshold to 1
init_gateway 5, 1, 1   // Configure gateway id=5 to edge-triggered/low
clear_gateway 5        // Clear gateway id=5
set_priority 5, 7      // Set id=5 threshold at 7
enable_interrupt 5     // Enable id=5
enable_ext_int         // Enable interrupts (MIE[meip]=1)
```

- Macro flow to initialize priority order:
  - To RISC-V standard order:

```
init_priorityorder 0   // Set priority to standard RISC-V order
init_nstthresholds 0   // Initialize nesting thresholds to 0
```

  - To reverse priority order:

```
init_priorityorder 1   // Set priority to reverse order
init_nstthresholds 15  // Initialize nesting thresholds to 15
```


- Code to jump to the interrupt handler from the RISC-V trap vector:

```
trap_vector:          // Interrupt trap starts here when MTVEC[mode]=1
    csrwi meicpct, 1 // Capture winning claim id and priority
    csrr t0, meihap  // Load pointer index
    lw t1, 0(t0)     // Load vector address
    jr t1            // Go there
```

- Code to handle the interrupt:

```
eint_handler:
   :                 // Do some useful interrupt handling
   mret              // Return from ISR
```

5.14.2 Example Interrupt Macros

- Disable external interrupt:

```
.macro disable_ext_int
    // Clear MIE[miep]
disable_ext_int_\@:
    li a0, (1<<11)
    csrrc zero, mie, a0
.endm
```

- Enable external interrupt:

```
.macro enable_ext_int
enable_ext_int_\@:
    // Set MIE[miep]
    li a0, (1<<11)
    csrrs zero, mie, a0
.endm
```

- Initialize external interrupt priority order:

```
.macro init_priorityorder priord
init_priorityorder_\@:
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MPICCFG_OFFSET)
    li t0, \priord
    sw t0, 0(tp)
.endm
```

- Initialize external interrupt nesting priority thresholds:

```
.macro init_nstthresholds threshold
init_nstthresholds_\@:
    li t0, \threshold
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MEICIDPL_OFFSET)
    sw t0, 0(tp)
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MEICURPL_OFFSET)
    sw t0, 0(tp)
.endm
```

- Set external interrupt priority threshold:

```
.macro set_threshold threshold
set_threshold_\@:
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MEIPT_OFFSET)
    li t0, \threshold
    sw t0, 0(tp)
.endm
```

- Enable interrupt for source *id*:

```
.macro enable_interrupt id
enable_interrupt_\@:
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MEIE_OFFSET + (\id <<2))
    li t0, 1
    sw t0, 0(tp)
.endm
```

- Set priority of source *id*:

```
.macro set_priority id, priority
set_priority_\@:
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MEIPL_OFFSET + (\id <<2))
    li t0, \priority
    sw t0, 0(tp)
.endm
```

- Initialize gateway of source *id*:

```
.macro init_gateway id, polarity, type
init_gateway_\@:
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MEIGWCTRL_OFFSET + (\id <<2))
    li t0, ((\polarity<<1) | \type)
    sw t0, 0(tp)
.endm
```

- Clear gateway of source *id*:

```
.macro clear_gateway id
clear_gateway_\@:
    li tp, (RV_PIC_BASE_ADDR + RV_PIC_MEIGWCLR_OFFSET + (\id <<2))
    sw zero, 0(tp)
.endm
```

6 Performance Monitoring

This chapter describes the performance monitoring features of the SweRV EH1 core.

6.1 Features

SweRV EH1 provides these performance monitoring features:

- Four standard 64-bit wide event counters
- Standard separate event selection for each counter
- Standard selective count enable/disable controllability
- Synchronized counter enable/disable controllability
- Standard cycle counter
- Standard retired instructions counter
- Support for standard SoC-based machine timer registers

6.2 Control/Status Registers

6.2.1 Standard RISC-V Registers

A list of performance monitoring-related standard RISC-V CSRs with references to their definitions:

- Machine Hardware Performance Monitor (`mcycle{|h}`, `minstret{|h}`, `mhpmcounter3{|h}`-`mhpmcounter31{|h}`, and `mhpmevent3`-`mhpmevent31`) (see Section 3.1.16 in [2])
- Machine Timer Registers (`mtime` and `mtimecmp`) (see Section 3.1.15 in [2])

6.2.2 Platform-specific Control/Status Registers

A summary of platform-specific control/status registers in CSR space:

- Group Performance Monitor Control Register (mgpmc) (see Section 6.2.2.1)

All reserved and unused bits in these control/status registers must be hardwired to '0'. Unless otherwise noted, all read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

6.2.2.1 Group Performance Monitor Control Register (mgpmc)

The `mgpmc` register allows to synchronously enable or disable the four machine hardware performance monitor counters `mhpmcounter3..6`. This register only controls if incrementing the counters on selected events is enabled or disabled, but does not affect the counter values of the hardware performance monitor counters (i.e., the counters are not reset or changed in any way).

This register is mapped to the non-standard read/write CSR address space.

Table 6-1 Group Performance Monitor Control Register (mgpmc, at CSR 0x7D0)

Field	Bits	Description	Access	Reset
Reserved	31:1	Reserved	R	0
enable	0	Group performance monitor counter control (`mhpmcounter3..6`): 0: Disable incrementing of all performance monitor counters 1: Enable incrementing of all performance monitor counters	R/W	1

6.3 Counters

Only event counters 3 to 6 (`mhpmcounter3{|h}`-`mhpmcounter6{|h}`) and their corresponding event selectors (`mhpmevent3`-`mhpmevent6`) are functional on SweRV EH1. Event counters 7 to 31 (`mhpmcounter7{|h}`-`mhpmcounter31{|h}`) and their corresponding event selectors (`mhpmevent7`-`mhpmevent31`) are hardwired to '0'.

6.4 Count-Impacting Conditions

A few comments to consider on conditions that have an impact on the performance monitor counting:

- While in the pmu/fw-halt power management state, performance counters (including the `mcycle` counter) are disabled.
- While in debug halt (db-halt) state, the *stopcount* bit in the `dcsr` (Debug Control and Status Register) register determines if performance counters are enabled.
- While in the pmu/fw-halt power management state or the debug halt (db-halt) state with the *stopcount* bit set, DMA accesses are allowed, but not counted by the performance counters. It would be up to the bus master to count accesses while the core is in a halt state.
- While executing PAUSE, performance counters are enabled.

Also, it is recommended that the performance counters are disabled (using the `mgpmc` register) before the counters and event selectors are modified, and then reenabled again. This minimizes the impact of reading and writing the counter and event selector CSRs on the event count values, specifically for the CSR read/write events (i.e., events #16 and #17). In general, performance counters are incremented after a read access to the counter CSRs, but before a write access to the counter CSRs.

6.5 Events

Table 6-2 provides a list of the countable events.

Note: The event selector registers `mhpmevent3-mhpmevent6` have WARL behavior. When writing a value larger than the highest supported event number, the event selector is set to the highest event number.

Table 6-2 List of Countable Events

Legend: IP = In-Pipe; OOP = Out-Of-Pipe

Event No	Event Name	Description
0		Reserved (no event counted)
1	cycles clocks active	Number of cycles clock active (OOP)
2	I-cache hits	Number of I-cache hits (OOP, speculative, valid fetch & hit)
3	I-cache misses	Number of I-cache misses (OOP, valid fetch & miss)
4	instr committed - all	Number of all (16b+62b) instructions committed (IP, non-speculative, 0/1/2)
5	instr committed - 16b	Number of 16b instructions committed (IP, non-speculative, 0/1/2)
6	instr committed - 32b	Number of 32b instructions committed (IP, non-speculative, 0/1/2)
7	instr aligned - all	Number of all (16b+32b) instructions aligned (OOP, speculative, 0/1/2)
8	instr decoded - all	Number of all (16b+32b) instructions decoded (OOP, speculative, 0/1/2)
9	muls committed	Number of multiplications committed (IP, 0/1)
10	divs committed	Number of divisions committed (IP, 0/1)
11	loads committed	Number of loads committed (IP, 0/1)
12	stores committed	Number of stores committed (IP, 0/1)
13	misaligned loads	Number of misaligned loads (IP, 0/1)
14	misaligned stores	Number of misaligned stores (IP, 0/1)

Event No	Event Name	Description
15	alus committed	Number of ALU[15] operations committed (IP, 0/1/2)
16	CSR read	Number of CSR read instructions committed (IP, 0/1)
17	CSR read/write	Number of CSR read/write instructions committed (IP, 0/1)
18	CSR write rd==0	Number of CSR write rd==0 instructions committed (IP, 0/1)
19	ebreak	Number of ebreak instructions committed (IP, 0/1)
20	ecall	Number of ecall instructions committed (IP, 0/1)
21	fence	Number of fence instructions committed (IP, 0/1)
22	fence.i	Number of fence.i instructions committed (IP, 0/1)
23	mret	Number of mret instructions committed (IP, 0/1)
24	branches committed	Number of branches committed (IP)
25	branches mispredicted	Number of branches mispredicted (IP)
26	branches taken	Number of branches taken (IP)
27	unpredictable branches	Number of unpredictable branches (IP)
28	cycles fetch stalled	Number of cycles fetch ready but stalled (OOP)
29	cycles aligner stalled	Number of cycles one or more instructions valid in aligner but IB full (OOP)
30	cycles decode stalled	Number of cycles one or more instructions valid in IB but decode stalled (OOP)
31	cycles postsync stalled	Number of cycles postsync stalled at decode (OOP)
32	cycles presync stalled	Number of cycles presync stalled at decode (OOP)
33	cycles frozen (lsu_freeze_dc3)	Number of cycles pipe is frozen by LSU (OOP)
34	cycles SB/WB stalled (lsu_store_stall_any)	Number of cycles decode stalled due to SB or WB full (OOP)
35	cycles DMA DCCM transaction stalled (dma_dccm_stall_any)	Number of cycles DMA stalled due to decode for load/store (OOP)
36	cycles DMA ICCM transaction stalled (dma_iccm_stall_any)	Number of cycles DMA stalled due to fetch (OOP)
37	exceptions taken	Number of exceptions taken (IP)
38	timer interrupts taken	Number of timer interrupts taken (IP)
39	external interrupts taken	Number of external interrupts taken (IP)
40	TLU flushes (flush lower)	Number of TLU flushes (flush lower) (IP)
41	branch error flushes	Number of branch error flushes (IP)
42	I-bus transactions - instr	Number of instr transactions on I-bus interface (OOP)

[15] NOP is an ALU operation. WFI is implemented as a NOP in SweRV EHX1 and, hence, counted as an ALU operation was well.

Event No	Event Name	Description
43	D-bus transactions - ld/st	Number of ld/st transactions on D-bus interface (OOP)
44	D-bus transactions - misaligned	Number of misaligned transactions on D-bus interface (OOP)
45	I-bus errors	Number of transaction errors on I-bus interface (OOP)
46	D-bus errors	Number of transaction errors on D-bus interface (OOP)
47	cycles stalled due to I-bus busy	Number of cycles stalled due to AXI or AHB-Lite I-bus busy (OOP)
48	cycles stalled due to D-bus busy	Number of cycles stalled due to AXI or AHB-Lite D-bus busy (OOP)
49	cycles interrupts disabled	Number of cycles interrupts disabled (MSTATUS.MIE==0) (OOP)
50	cycles interrupts stalled while disabled	Number of cycles interrupts stalled while disabled (MSTATUS.MIE==0) (OOP)

7 Cache Control

This chapter describes the features to control the SweRV EH1 core's instruction cache (I-cache).

7.1 Features

The SweRV EH1's I-cache control features are:

- Flushing the I-cache
- Capability to enable/disable I-cache
- Diagnostic access to data, tag, and status information of the I-cache

Note: The I-cache is an optional core feature. Instantiation of the I-cache is controlled by the RV_ICACHE_ENABLE build argument.

7.2 Feature Descriptions

7.2.1 Cache Flushing

As described in Section 2.7.2, a debugger may initiate an operation that is equivalent to a `fence.i` instruction by writing a '1' to the *fence.i* field of the `dmst` register. As part of executing this operation, the I-cache is flushed (i.e., all entries in the I-cache are invalidated).

7.2.2 Enabling/Disabling I-Cache

As described in Section 2.7.1, each of the 16 memory regions has two control bits which are hosted in the `mrac` register. One of these control bits, *cacheable*, controls if accesses to that region may be cached. If the *cacheable* bits of all 16 regions are set to '0', the I-cache is effectively turned off.

7.2.3 Diagnostic Access

For firmware as well as hardware debug, direct access to the raw content of the data array, tag array, and status bits of the I-cache may be important. Instructions stored in the cache, the tag of a cache line as well as status information including a line's valid bit and a set's LRU bits can be manipulated. It is also possible to inject a parity/ECC error in the data or tag array to check error recovery. Four control registers are used to provide read/write diagnostic access to the two arrays and status bits. The `dicawics` register controls the selection of the array, way, and index of a cache line. The `dicad0`/`1` and `dicago` registers are used to perform a read or write access to the selected array location. See Sections 7.5.1 - 7.5.4 for more detailed information.

Note: The instructions and the tags are stored in parity/ECC-protected SRAM arrays. The status bits are stored in flops.

7.3 Use Cases

The I-cache control features can be broadly divided into two categories:

1. **Debug Support**

   A few examples how diagnostic accesses (Section 7.2.3) may be useful for debug:

   - Generating an I-cache dump (e.g., to investigate performance issues).
   - Injecting parity/ECC errors in the data or tag array of the I-cache.
   - Diagnosing stuck-at bits in the data or tag array of the I-cache.
   - Preloading the I-cache if a hardware bug prevents instruction fetching from memory.

2. **Performance Evaluation**

   To evaluate the performance advantage of the I-cache, it is useful to run code with and without the cache enabled. Enabling and disabling the I-cache (Section 7.2.2) is an essential feature for this.

7.4 Theory of Operation

7.4.1 Read a Chunk of an I-cache Cache Line

The following steps must be performed to read a 32-bit chunk of instruction data and its associated 2 parity / 10 ECC bits in an I-cache cache line:

1. Write array/way/address information which location to access in the I-cache to the `dicawics` register:
    - *array* field: 0 (i.e., I-cache data array)
    - *way* field: way to be accessed (i.e., 0..3)
    - *index* field: index of cache line to be accessed
2. Read the `dicago` register which causes a read access from the I-cache data array at the location selected by the `dicawics` register
3. Read the `dicad0` register to get the selected 32-bit cache line chunk (*instr* field), and read the `dicad1` register to get the associated parity/ECC bits (*parity0* and *parity1* / *ecc0* and *ecc1* fields)

7.4.2 Write a Chunk of an I-cache Cache Line

The following steps must be performed to write a 32-bit chunk of instruction data and its associated 2 parity / 10 ECC bits in an I-cache cache line:

1. Write array/way/address information which location to access in the I-cache to the `dicawics` register:
    - *array* field: 0 (i.e., I-cache data array)
    - *way* field: way to be accessed (i.e., 0..3)
    - *index* field: index of cache line to be accessed
2. Write the new instruction information to the *instr* field of the `dicad0` register, and write the calculated correct instruction parity/ECC bits (unless error injection should be performed) to the *parity0* and *parity1* / *ecc0* and *ecc1* fields of the `dicad1` register
3. Write a '1' to the *go* field of the `dicago` register which causes a write access to the I-cache data array copying the information stored in the `dicad0`/`1` registers to the location selected by the `dicawics` register

7.4.3 Read or Write a Full I-cache Cache Line

The following steps must be performed to read or write instruction data and associated parity/ECC bits of a full I-cache cache line:

1. Start with an index naturally aligned to the 64-byte cache line size (i.e., *index[5:2]* = '0000').
2. Perform steps in Section 7.4.1 to read or Section 7.4.2 to write.
3. Increment the index.
4. Go back to step 2.) for a total of 16 iterations.

7.4.4 Read a Tag and Status Information of an I-cache Cache Line

The following steps must be performed to read the tag, tag's parity/ECC bit(s), and status information of an I-cache cache line:

1. Write array/way/address information which location to access in the I-cache to the `dicawics` register:
    - *array* field: 1 (i.e., I-cache tag array and status)
    - *way* field: way to be accessed (i.e., 0..3)
    - *index* field: index of cache line to be accessed
2. Read the `dicago` register which causes a read access from the I-cache tag array and status bits at the location selected by the `dicawics` register
3. Read the `dicad0` register to get the selected cache line's tag (*tag* field) and valid bit (*valid* field) as well as the set's LRU bits (*lru* field), and read the `dicad1` register to get the tag's parity/ECC bit(s) (*parity0* / *ecc0* field)

7.4.5 Write a Tag and Status Information of an I-cache Cache Line

The following steps must be performed to write the tag, tag's parity/ECC bit, and status information of an I-cache cache line:

1. Write array/way/address information which location to access in the I-cache to the `dicawics` register:

- *array* field: 1 (i.e., I-cache tag array and status)
- *way* field: way to be accessed (i.e., 0..3)
- *index* field: index of cache line to be accessed

2. Write the new tag, valid, and LRU information to the *tag*, *valid*, and *lru* fields of the `dicad0` register, and write the calculated correct tag parity/ECC bit (unless error injection should be performed) to the *parity0* / *ecc0* field of the `dicad1` register
3. Write a '1' to the *go* field of the `dicago` register which causes a write access to the I-cache tag array and status bits copying the information stored in the `dicad0/1` registers to the location selected by the `dicawics` register

7.5 I-Cache Control/Status Registers

A summary of the I-cache control/status registers in CSR address space:

- I-Cache Array/Way/Index Selection Register (dicawics) (see Section 7.5.1)
- I-Cache Array Data 0 Register (dicad0) (see Section 7.5.2)
- I-Cache Array Data 1 Register (dicad1) (see Section 7.5.3)
- I-Cache Array Go Register (dicago) (see Section 7.5.4)

All reserved and unused bits in these control/status registers must be hardwired to '0'. Unless otherwise noted, all read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

7.5.1 I-Cache Array/Way/Index Selection Register (dicawics)

The `dicawics` register is used to select a specific location in either the data array or the tag array / status of the I-cache. In addition to selecting the array, the location in the array must be specified by providing the way, and index. Once selected, the `dicad0/1` registers (see Sections 7.5.2 and 7.5.3) hold the information read from or written to the specified location, and the `dicago` register (see Section 7.5.4) is used to control the read/write access to the specified I-cache array.

The cache line size of the I-cache is 64 bytes. The *dicawics* register addresses two chunks consisting each of 16 consecutive bits of instruction data and separately protected by parity/ECC bits. There are 16 such chunk pairs in a cache line.

Note: This register is accessible in **debug mode only**. Attempting to access this register in machine mode raises an illegal instruction exception.

This register is mapped to the non-standard read-write CSR address space.

Table 7-1 I-Cache Array/Way/Index Selection Register (dicawics, at CSR 0x7C8)

Field	Bits	Description	Access	Reset
Reserved	31:25	Reserved	R	0
array	24	Array select: 0: I-cache data array (incl. parity/ECC bits) 1: I-cache tag array (incl. parity/ECC bits) and status (incl. valid and LRU bits)	R/W	0
Reserved	23:22	Reserved	R	0
way	21:20	Way select	R/W	0
Reserved	19:16	Reserved	R	0

Field	Bits	Description	Access	Reset
index[16]	15:2	Index address bits select **Notes:** • Index bits are right-justified; for I-cache sizes smaller than 256 KB, unused upper bits are 0 • For tag array and status, bits 5..2 are ignored by hardware	R/W	0
Reserved	1:0	Reserved	R	0

7.5.2 I-Cache Array Data 0 Register (dicad0)

The `dicad0` register, in combination with the `dicad1` register (see Section 7.5.3), is used to store information read from and written to the I-cache array location specified with the `dicawics` register (see Section 7.5.1). Triggering a read or write access of the I-cache array is controlled by the `dicago` register (see Section 7.5.4). The layout of the `dicad` register is different for the data array and the tag array / status, as described in Table 7-2 below.

Note: During normal operation, the parity/ECC bits over the 32-bit instruction data as well as the tag are generated and checked by hardware. However, to enable error injection, the parity/ECC bits must be computed by software for I-cache data and tag array diagnostic writes.

Note: This register is accessible in **debug mode only**. Attempting to access this register in machine mode raises an illegal instruction exception.

This register is mapped to the non-standard read-write CSR address space.

Table 7-2 I-Cache Array Data 0 Register (dicad0, at CSR 0x7C9)

Field	Bits	Description	Access	Reset
I-cache data array				
instr	31:0	Instruction data 31:16: instruction data bytes 3/2 (protected by *parity1* / *ecc1*) 15:0: instruction data bytes 1/0 (protected by *parity0* / *ecc0*)	R/W	0
I-cache tag array and status bits				
tag	31:12	Tag **Note:** • Tag bits are left-justified; for I-cache sizes larger than 16 KB, unused lower bits are 0	R/W	0
Unused	11:7	Unused	R/W	0

[16] SweRV EHX1's I-cache supports four way-set associativity, each way is subdivided into 4 banks, and each bank hosts 16 bytes of a 64-byte cache line. A bank is selected by *index[5:4]*. The 16 bytes within a bank are selected by *index[3:2]* in increasing 32-bit chunk pairs (i.e., '00': bytes 3..0, '01': bytes 7..4, '10': bytes 11..8, and '11': bytes 15..12).

Field	Bits	Description	Access	Reset
lru	6:4	Pseudo LRU bits (same bits are accessed independent of selected way): Bit 4: way0/1 / way2/3 selection 0: way0/1 1: way2/3 Bit 5: way0 / way1 selection 0: way0 1: way1 Bit 6: way2 / way3 selection 0: way2 1: way3	R/W	0
Unused	3:1	Unused	R/W	0
valid	0	Cache line valid/invalid: 0: cache line invalid 1: cache line valid	R/W	0

7.5.3 I-Cache Array Data 1 Register (dicad1)

The `dicad1` register, in combination with the `dicad0` register (see Section 7.5.37.5.2), is used to store information read from and written to the I-cache array location specified with the `dicawics` register (see Section 7.5.1). Triggering a read or write access of the I-cache array is controlled by the `dicago` register (see Section 7.5.4). The layout of the `dicad1` register is described in Table 7-3 below.

Note: During normal operation, the parity/ECC bits over the 32-bit instruction data as well as the tag are generated and checked by hardware. However, to enable error injection, the parity/ECC bits must be computed by software for I-cache data and tag array diagnostic writes.

Note: This register is accessible in **debug mode only**. Attempting to access this register in machine mode raises an illegal instruction exception.

This register is mapped to the non-standard read-write CSR address space.

Table 7-3 I-Cache Array Data 1 Register (dicad1, at CSR 0x7CA)

Field	Bits	Description	Access	Reset
Parity				
Reserved	31:2	Reserved	R	0
parity1	1	Even parity for I-cache data bytes 3/2 (*instr[31:16]*)	R/W	0
parity0	0	Even parity for I-cache data bytes 1/0 (*instr[15:0]*), or Even parity for I-cache tag (*tag*)	R/W	0
ECC				
Reserved	31:10	Reserved	R	0
ecc1	9:5	ECC for I-cache data bytes 3/2 (*instr[31:16]*)	R/W	0
ecc0	4:0	ECC for I-cache data bytes 1/0 (*instr[15:0]*), or ECC for I-cache tag (*tag*)	R/W	0

7.5.4 I-Cache Array Go Register (dicago)

The `dicago` register is used to trigger a read from or write to the I-cache array location specified with the `dicawics` register (see Section 7.5.1). Reading the `dicago` register populates the `dicad0`/`dicad1` registers (see Sections 7.5.2 and 7.5.3) with the information read from the I-cache array. Writing a '1' to the *go* field of the `dicago` register copies the information stored in the `dicad0`/`dicad1` registers to the I-cache array. The layout of the `dicago` register is described in Table 7-4 below.

Note: This register is accessible in **debug mode only**. Attempting to access this register in machine mode raises an illegal instruction exception.

The *go* field of the `dicago` register has W1R0 (Write 1, Read 0) behavior, as also indicated in the 'Access' column.

This register is mapped to the non-standard read-write CSR address space.

Table 7-4 I-Cache Array Go Register (dicago, at CSR 0x7CB)

Field	Bits	Description	Access	Reset
Reserved	31:1	Reserved	R	0
go	0	Read triggers an I-cache read, write-1 triggers an I-cache write	R0/W1	0

8 Low-Level Core Control

This chapter describes some low-level core control registers.

8.1 Control/Status Registers

A summary of platform-specific control/status registers in CSR space:

- Feature Disable Register (mfdc) (see Section 8.1.1)
- Clock Gating Control Register (mcgc) (see Section 8.1.2)

All reserved and unused bits in these control/status registers must be hardwired to '0'. Unless otherwise noted, all read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

8.1.1 Feature Disable Register (mfdc)

The `mfdc` register hosts low-level core control bits to disable specific features. This may be useful in case a feature intended to increase core performance should prove to have problems.

Note: `fence.i` instructions are required before and after writes to the `mfdc` register.

Note: The default state of the controllable features is 'enabled'. Firmware may turn off a feature if needed.

This register is mapped to the non-standard read/write CSR address space.

Table 8-1 Feature Disable Register (mfdc, at CSR 0x7F9)

Field	Bits	Description	Access	Reset
Reserved	31:11	Reserved	R	0
	10	Dual issue disable	R/W	0
	9	PIC multiple interrupts disable	R/W	0
	8	Core ECC check disable	R/W	0
	7	Secondary ALU disable	R/W	0
Reserved	6	Reserved	R	0
	5	LSU/DIV non-blocking disable	R/W	0
	4	Fast divide disable	R/W	0
	3	Branch prediction disable	R/W	0
	2	Write Buffer (WB) coalescing disable	R/W	0
	1	Load miss bypass Write Buffer (WB) disable	R/W	0
	0	Pipelining disable	R/W	0

8.1.2 Clock Gating Control Register (mcgc)

The `mcgc` register hosts low-level core control bits to override clock gating for specific units. This may be useful in case a unit intended to be clock gated should prove to have problems when in lower power mode.

Note: The default state of the clock gating overrides is 'disabled'. Firmware may turn off clock gating (i.e., set the clock gating override bit) for a specific unit if needed.

This register is mapped to the non-standard read/write CSR address space.

Table 8-2 Clock Gating Control Register (mcgc, at CSR 0x7F8)

Field	Bits	Description	Access	Reset
Reserved	31:9	Reserved	R	0
	8	Miscellaneous	R/W	0
	7	DEC	R/W	0
	6	EXU	R/W	0
	5	IFU	R/W	0
	4	LSU	R/W	0
	3	Bus	R/W	0
	2	PIC	R/W	0
	1	DCCM	R/W	0
	0	ICCM	R/W	0

9 Standard RISC-V CSRs with Core-Specific Adaptations

A summary of standard RISC-V control/status registers in CSR space with platform-specific adaptations:

- Machine Interrupt Enable (mie) and Machine Interrupt Pending (mip) Registers (see Section 9.1.1)
- Machine Cause Enable Register (mcause) (see Section 9.1.2)

All reserved and unused bits in these control/status registers must be hardwired to '0'. Unless otherwise noted, all read/write control/status registers must have WARL (Write Any value, Read Legal value) behavior.

9.1.1 Machine Interrupt Enable (mie) and Machine Interrupt Pending (mip) Registers

The standard RISC-V `mie` and `mip` registers hold the machine interrupt enable and interrupt pending bits, respectively. Since SweRV EH1 only supports machine mode, all supervisor- and user-specific bits are not implemented. In addition, the `mie`/`mip` registers also host the platform-specific local interrupt enable/pending bits (shown with a gray background in Table 9-1 and Table 9-2 below).

Table 9-1 Machine Interrupt Enable Register (mie, at CSR 0x304)

Field	Bits	Description	Access	Reset
Reserved	31	Reserved	R	0
mceie	30	Correctable error local interrupt enable	R/W	0
Reserved	29:12	Reserved	R	0
meie	11	Machine external interrupt enable	R/W	0
Reserved	10:8	Reserved	R	0
mtie	7	Machine timer interrupt enable	R/W	0
Reserved	6:4	Reserved	R	0
msie	3	Machine software interrupt enable[17]	R/W	0
Reserved	2:0	Reserved	R	0

Table 9-2 Machine Interrupt Pending Register (mip, at CSR 0x344)

Field	Bits	Description	Access	Reset
Reserved	31	Reserved	R	0
mceip	30	Correctable error local interrupt pending	R	0
Reserved	29:12	Reserved	R	0
meip	11	Machine external interrupt pending	R	0
Reserved	10:8	Reserved	R	0
mtip	7	Machine timer interrupt pending	R	0
Reserved	6:0	Reserved	R	0

[17] The *msie* bit is physically implemented, but has no functional effect since the 'software interrupt' request signal is hardwired to '0'.

9.1.2 Machine Cause Enable Register (mcause)

The standard RISC-V `mcause` register indicates the cause for a trap as shown in Table 9-3, including standard exceptions/interrupts, platform-specific local interrupts (with light gray background), and NMI causes (with dark gray background).

Note: The `mcause` register has WLRL (Write Legal value, Read Legal value) behavior.

Table 9-3 Machine Cause Register (mcause, at CSR 0x342)

Value			Description	Section(s)
Interrupt mcause[31]	**Exception Code mcause[30:0]**	**Hex**		
		0x0000_0000	NMI pin assertion	2.12
0	1	0x0000_0001	Instruction access fault	2.6.4, 2.6.6, and 3.4
	2	0x0000_0002	Illegal instruction	
	3	0x0000_0003	Breakpoint	
	4	0x0000_0004	Load address misaligned	2.6.5
	5	0x0000_0005	Load access fault	2.6.4, 2.6.6, and 3.4
	6	0x0000_0006	Store/AMO address misaligned	2.6.5
	7	0x0000_0007	Store/AMO access fault	2.6.4, 2.6.6, and 3.4
	11	0x0000_000B	Environment call from M-mode	
1	7	0x8000_0007	Machine timer interrupt	
	11	0x8000_000B	Machine external interrupt	
	30	0x8000_001E	Machine correctable error local interrupt	2.6.2
		0xF000_0000	Machine D-bus store error NMI	2.6.1
		0xF000_0001	Machine D-bus non-blocking load error NMI	2.6.1

Note: All other values are reserved.

10 CSR Address Map

10.1 Core-Specific Standard RISC-V CSRs

Table 10-1 lists the SweRV EH1 core-specific standard RISC-V Machine Information CSRs.

Table 10-1 SweRV EH1 Core-Specific Standard RISC-V Machine Information CSRs

Number	Privilege	Name	Description	Value
0x301	MRW	misa	ISA and extensions (**Note:** writes ignored)	0x4000_1104
0xF11	MRO	mvendorid	Vendor ID	0x0000_0045
0xF12	MRO	marchid	Architecture ID	0x0000_000B
0xF13	MRO	mimpid	Implementation ID	0x0000_0001
0xF14	MRO	mhartid	Hardware thread ID	0x0000_0000

10.2 Non-Standard RISC-V CSRs

Table 10-2 summarizes the SweRV EH1 non-standard RISC-V CSR address map.

Table 10-2 SweRV EH1 Non-Standard RISC-V CSR Address Map

Number	Privilege	Name	Description	Section
0x7C0	MRW	mrac	Region access control	2.7.1
0x7C2	MRW	mcpc	Core pause control	4.4.2
0x7C4	DRW	dmst	Memory synchronization trigger (debug mode only)	2.7.2
0x7C6	MRW	mpmc	Power management control	4.4.1
0x7C8	DRW	dicawics	I-cache array/way/index selection (debug mode only)	7.5.1
0x7C9	DRW	dicad0	I-cache array data 0 (debug mode only)	7.5.2
0x7CA	DRW	dicad1	I-cache array data 1 (debug mode only)	7.5.3
0x7CB	DRW	dicago	I-cache array go (debug mode only)	7.5.4
0x7D0	MRW	mgpmc	Group performance monitor control	6.2.2.1
0x7F0	MRW	micect	I-cache error counter/threshold	3.5.1
0x7F1	MRW	miccmect	ICCM correctable error counter/threshold	3.5.2
0x7F2	MRW	mdccmect	DCCM correctable error counter/threshold	3.5.3
0x7F8	MRW	mcgc	Clock gating control	8.1.2
0x7F9	MRW	mfdc	Feature disable control	8.1.1
0xBC0	MRW	mdeau	D-Bus error address unlock	2.7.4
0xBC8	MRW	meivt	External interrupt vector table	5.11.6
0xBC9	MRW	meipt	External interrupt priority threshold	5.11.5
0xBCA	MRW	meicpct	External interrupt claim ID / priority level capture trigger	5.11.8
0xBCB	MRW	meicidpl	External interrupt claim ID's priority level	5.11.9

Number	Privilege	Name	Description	Section
0xBCC	MRW	meicurpl	External interrupt current priority level	5.11.10
0xFC0	MRO	mdseac	D-bus first error address capture	2.7.3
0xFC8	MRO	meihap	External interrupt handler address pointer	5.11.7

11 Interrupt Priorities

Table 11-1 summarizes the SweRV EH1 platform-specific (Local) and standard RISC-V (External and Timer) relative interrupt priorities.

Table 11-1 SweRV EH1 Platform-specific and Standard RISC-V Interrupt Priorities

	Interrupt	Section
Highest Interrupt Priority	*Non-Maskable Interrupt (standard RISC-V)*	
	External interrupt (standard RISC-V)	
	Correctable error (local interrupt)	2.6.2
Lowest Interrupt Priority	*Timer interrupt (standard RISC-V)*	

12 Clock and Reset

This chapter describes clocking and reset signals used by the SweRV EH1 core complex.

12.1 Features

The SweRV EH1 core complex's clock and reset features are:

- Support for independent clock ratios for four separate system bus interfaces
  - System bus clock ratios controlled by SoC
- Single core clock input
  - System bus clock ratios controlled by enable signals
- Single core reset signal
  - Ability to reset to Debug Mode

12.2 Clocking

12.2.1 Regular Operation

The SweRV EH1 core complex is driven by a single clock (`clk`). All input and output signals, except those listed in Table 12-1, are synchronous to `clk`.

The core complex provides three master system bus interfaces (for instruction fetch, load/store data, and debug) as well as one slave (DMA) system bus interface. The SoC controls the clock ratio for each system bus interface via the clock enable signal (`*_bus_clk_en`). The clock ratios selected by the SoC may be the same or different for each system bus.

Figure 12-1 depicts the conceptual relationship of the clock (`clk`), system bus enable (`*_bus_clk_en`) used to select the clock ratio for each system bus, and the data (`*data`) of the respective system bus.

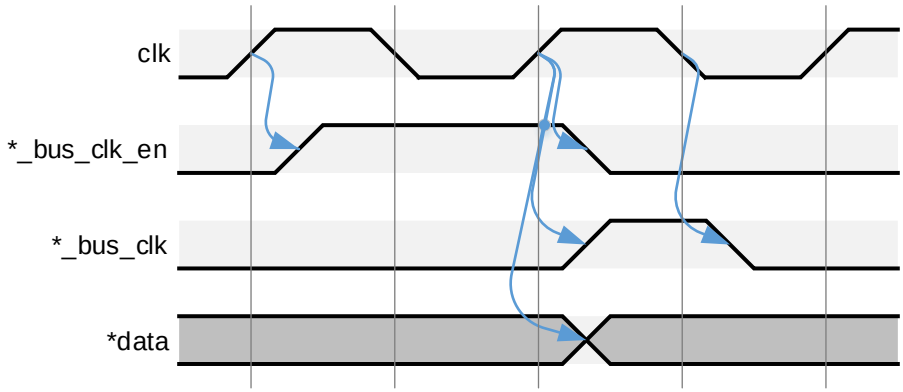


Figure 12-1 Conceptual Clock, Clock-Enable, and Data Timing Relationship

Note that the clock net is not explicitly buffered, as the clock tree is expected to be synthesized during place-and-route. The achievable clock frequency depends on the configuration, the sizes and configuration of I-cache and I/DCCMs, and the silicon implementation technology.

12.2.2 System Bus-to-Core Clock Ratios

Figure 12-2 to Figure 12-9 depict the timing relationships of clock, clock-enable, and data for the supported system bus clock ratios from 1:1 (i.e., the system bus and core run at the same rate) to 1:8 (i.e., the system bus runs eight times slower than the core).

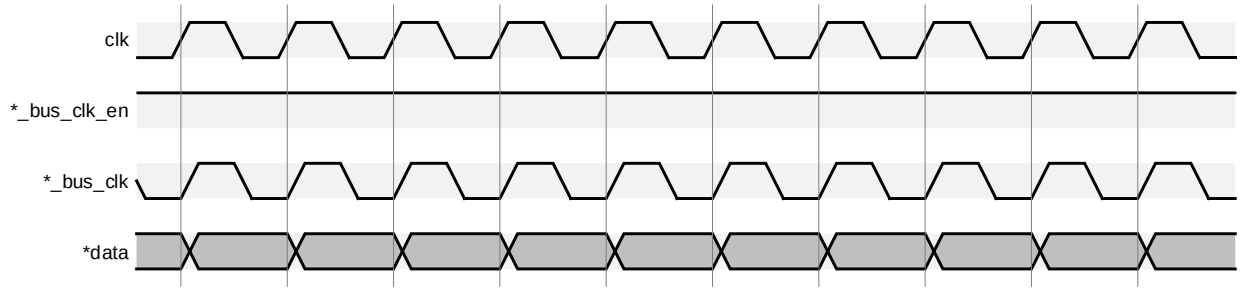


Figure 12-2 1:1 System Bus-to-Core Clock Ratio

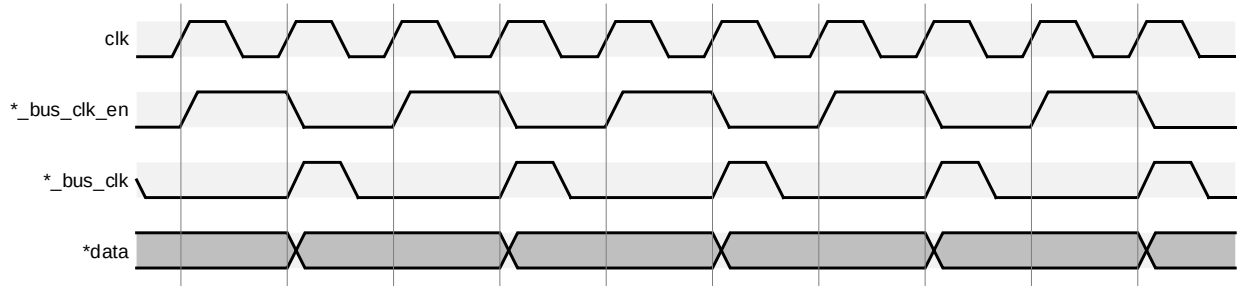


Figure 12-3 1:2 System Bus-to-Core Clock Ratio

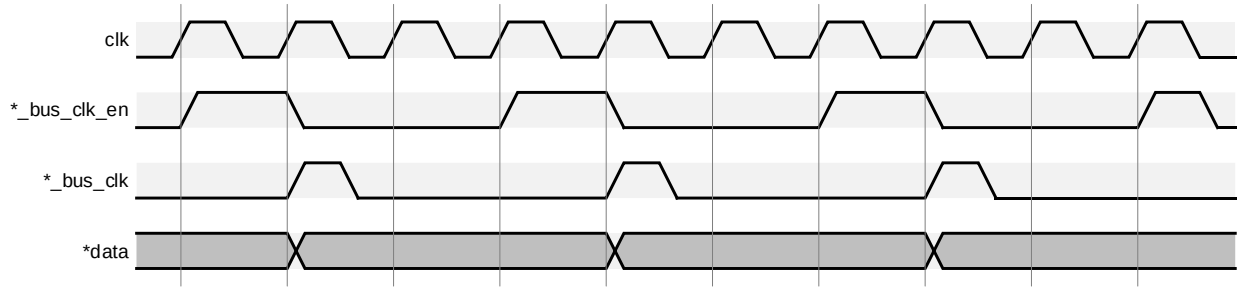


Figure 12-4 1:3 System Bus-to-Core Clock Ratio

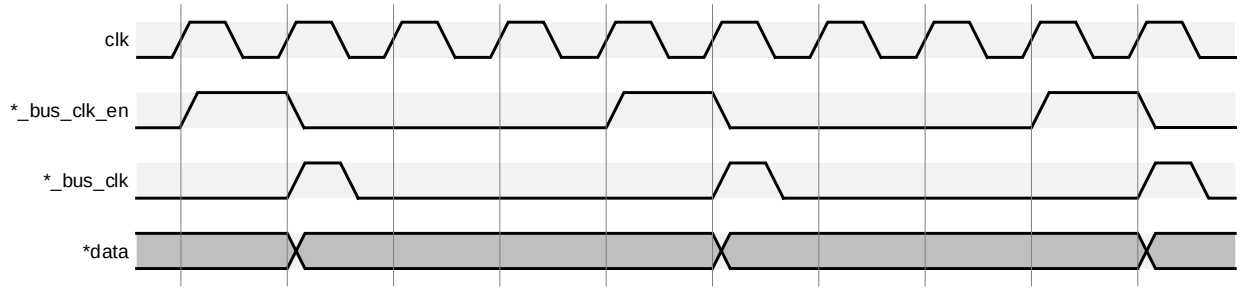


Figure 12-5 1:4 System Bus-to-Core Clock Ratio

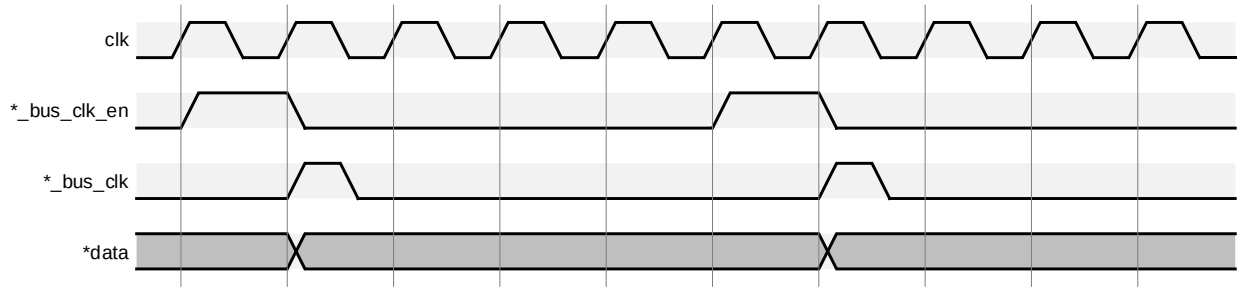


Figure 12-6 1:5 System Bus-to-Core Clock Ratio

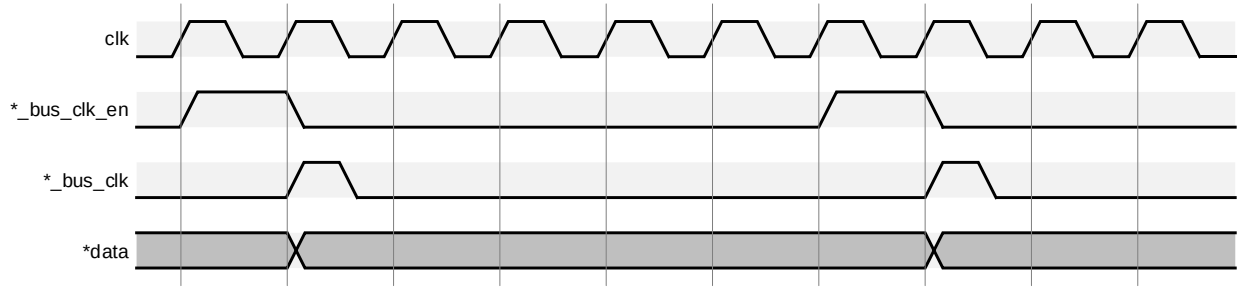


Figure 12-7 1:6 System Bus-to-Core Clock Ratio

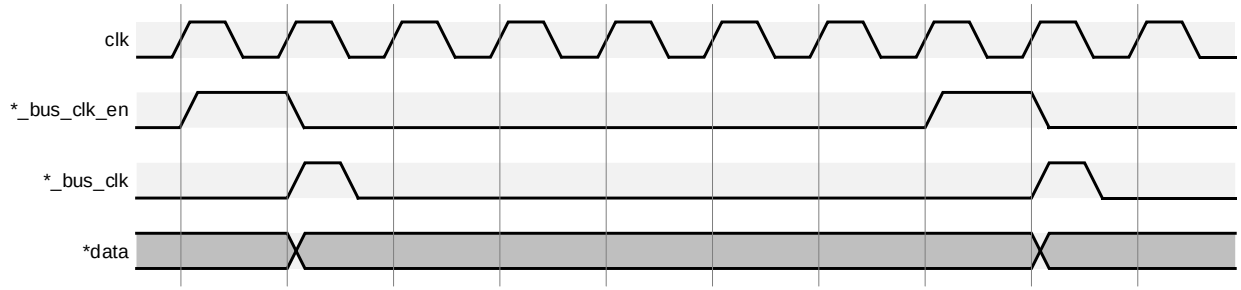


Figure 12-8 1:7 System Bus-to-Core Clock Ratio

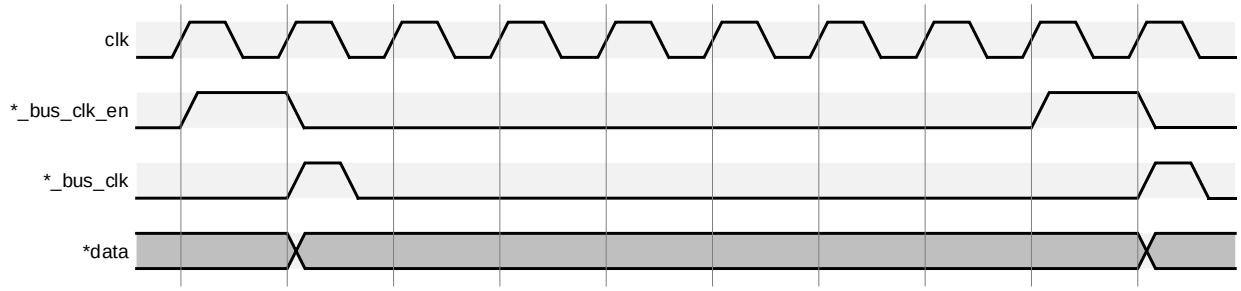


Figure 12-9 1:8 System Bus-to-Core Clock Ratio

12.2.3 Asynchronous Signals

Table 12-1 provides a list of signals which are asynchronous to the core clock (`clk`). Signals which are inputs to the core complex are synchronized to `clk` in the core complex logic. Signals which are outputs of the core complex must

be synchronized outside of the core complex logic if the respective receiving clock domain is driven by a different clock than `clk`.

Note that each asynchronous input passes through a two-stage synchronizer. The signal must be asserted for at least two full `clk` cycles to guarantee it is detected by the core complex logic. Shorter pulses might be dropped by the synchronizer circuit.

Table 12-1 Core Complex Asynchronous Signals

Signal	Dir	Description
Interrupts		
extintsrc_req[RV_PIC_TOTAL_INT:1]	in	External interrupts
timer_int	in	Standard RISC-V timer interrupt
nmi_int	in	Non-Maskable Interrupt
Power Management Interface		
i_cpu_halt_req	in	Halt request to core
o_cpu_halt_ack	out	Core acknowledgement for halt request
o_cpu_halt_status	out	Core halted indication
o_debug_mode_status	out	Core in debug mode indication
i_cpu_run_req	in	Run request to core
o_cpu_run_ack	out	Core acknowledgement for run request
JTAG		
jtag_tck	in	JTAG Test Clock
jtag_tms	in	JTAG Test Mode Select
jtag_tdi	in	JTAG Test Data In
jtag_trst_n	in	JTAG Test Reset
jtag_tdo	out	JTAG Test Data Out

12.3 Reset

As shown in Figure 12-10, the core complex reset signal (`rst_l`) is active-low, may be asynchronously asserted, but must be synchronously de-asserted to avoid any glitches. The `rst_l` input signal is not synchronized to the core clock (`clk`) inside the core complex logic. All core complex flops are reset asynchronously.

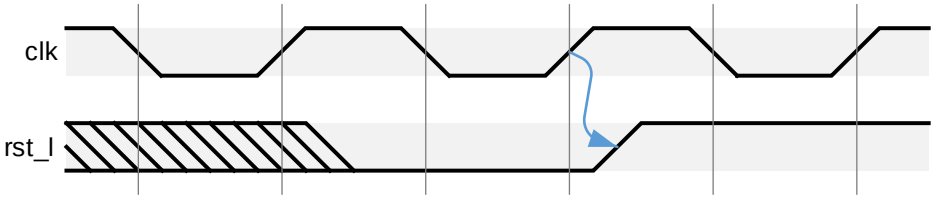


Figure 12-10 Conceptual Clock and Reset Timing Relationship

Note that the core complex clock (`clk`) must be stable before the core complex reset (`rst_l`) is deasserted. Also, the `rst_l` signal is not explicitly buffered, as synthesis tools are expected to automatically buffer the `rst_l` net.

12.3.1 Debugger Initiating Reset via JTAG Interface

A debugger may also initiate a reset of the core complex logic via the JTAG interface. Note that such a reset assertion is not visible to the SoC. Resetting the core complex while the core is accessing any SoC memory locations may result in unpredictable behavior. Recovery may require an assertion of the SoC master reset.

12.3.2 Core Complex Reset to Debug Mode

The RISC-V Debug specification [3] states a requirement that the debugger must be able to be in control from the first executed instruction of a program after a reset.

The Debug Module controls the core-complex-internal `ndmreset` (non-debug module reset) signal. This signal resets the core complex (except for the Debug Module and Debug Transport Module).

The following sequence is used to reset the core and execute the first instruction in debug mode (i.e., db-halt state):

1. Take Debug Module out of reset
    - Set *dmactive* bit in `dmcontrol` register (`dmcontrol` = 0x0000_0001)
2. Halt the core
    - Set *haltreq* bit in `dmcontrol` register (`dmcontrol` = 0x8000_0001)
3. Wait for core halt and remove halt request
    - Clear *haltreq* bit in `dmcontrol` register (`dmcontrol` = 0x0000_0001)
4. Reset core complex
    - Set *ndmreset* bit in `dmcontrol` register (`dmcontrol` = 0x0000_0003)
5. While in reset, assert halt request again with ndmreset still asserted
    - Set *haltreq* bit in `dmcontrol` register (`dmcontrol` = 0x8000_0003)
6. Take core complex out of reset with halt request still asserted
    - Clear *ndmreset* bit in `dmcontrol` register (`dmcontrol` = 0x8000_0001)

13 SweRV EH1 Core Complex Port List

Table 13-1 lists the core complex signals. Not all signals are present in a given instantiation. For example, a core complex can only have one bus interface type (AXI4 or AHB-Lite). Signals which are asynchronous to the core complex clock (`clk`) are marked with "(async)" in the 'Description' column.

Table 13-1 Core Complex Signals

Signal	Dir	Description
Clock and Clock Enables		
clk	in	Core complex clock
ifu_bus_clk_en	in	IFU master system bus clock enable
lsu_bus_clk_en	in	LSU master system bus clock enable
dbg_bus_clk_en	in	Debug master system bus clock enable
dma_bus_clk_en	in	DMA slave system bus clock enable
Reset		
rst_l	in	Core complex reset
rst_vec[31:1]	in	Core complex reset vector
Interrupts		
nmi_int	in	Non-Maskable Interrupt (async)
nmi_vec[31:1]	in	Non-Maskable Interrupt vector
timer_int	in	Standard RISC-V timer interrupt (async)
extintsrc_req[RV_PIC_TOTAL_INT:1]	in	External interrupts (async)
System Bus Interfaces		
AXI4		
Instruction Fetch Unit Master AXI[18]		
Write address channel signals		
ifu_axi_awvalid	out	Write address valid *(hardwired to 0)*
ifu_axi_awready	in	Write address ready
ifu_axi_awid[RV_IFU_BUS_TAG-1:0]	out	Write address ID
ifu_axi_awaddr[31:0]	out	Write address
ifu_axi_awlen[7:0]	out	Burst length
ifu_axi_awsize[2:0]	out	Burst size
ifu_axi_awburst[1:0]	out	Burst type
ifu_axi_awlock	out	Lock type
ifu_axi_awcache[3:0]	out	Memory type
ifu_axi_awprot[2:0]	out	Protection type

[18] The IFU issues only read, but no write transactions. However, the IFU write address, data, and response channels are present, but the valid/ready signals are tied off to disable those channels.

Signal	Dir	Description
ifu_axi_awqos[3:0]	out	Quality of Service (QoS)
ifu_axi_awregion[3:0]	out	Region identifier
Write data channel signals		
ifu_axi_wvalid	out	Write valid *(hardwired to 0)*
ifu_axi_wready	in	Write ready
ifu_axi_wdata[63:0]	out	Write data
ifu_axi_wstrb[7:0]	out	Write strobes
ifu_axi_wlast	out	Write last
Write response channel signals		
ifu_axi_bvalid	in	Write response valid
ifu_axi_bready	out	Write response ready *(hardwired to 0)*
ifu_axi_bid[RV_IFU_BUS_TAG-1:0]	in	Response ID tag
ifu_axi_bresp[1:0]	in	Write response
Read address channel signals		
ifu_axi_arvalid	out	Read address valid
ifu_axi_arready	in	Read address ready
ifu_axi_arid[RV_IFU_BUS_TAG-1:0]	out	Read address ID
ifu_axi_araddr[31:0]	out	Read address
ifu_axi_arlen[7:0]	out	Burst length *(hardwired to 0b0000_0000)*
ifu_axi_arsize[2:0]	out	Burst size *(hardwired to 0b011)*
ifu_axi_arburst[1:0]	out	Burst type *(hardwired to 0b01)*
ifu_axi_arlock	out	Lock type *(hardwired to 0)*
ifu_axi_arcache[3:0]	out	Memory type *(hardwired to 0b1111)*
ifu_axi_arprot[2:0]	out	Protection type *(hardwired to 0b100)*
ifu_axi_arqos[3:0]	out	Quality of Service (QoS) *(hardwired to 0b0000)*
ifu_axi_arregion[3:0]	out	Region identifier
Read data channel signals		
ifu_axi_rvalid	in	Read valid
ifu_axi_rready	out	Read ready
ifu_axi_rid[RV_IFU_BUS_TAG-1:0]	in	Read ID tag
ifu_axi_rdata[63:0]	in	Read data
ifu_axi_rresp[1:0]	in	Read response
ifu_axi_rlast	in	Read last
Load/Store Unit Master AXI		
Write address channel signals		
lsu_axi_awvalid	out	Write address valid

Signal	Dir	Description
lsu_axi_awready	in	Write address ready
lsu_axi_awid[RV_LSU_BUS_TAG-1:0]	out	Write address ID
lsu_axi_awaddr[31:0]	out	Write address
lsu_axi_awlen[7:0]	out	Burst length *(hardwired to 0b0000_0000)*
lsu_axi_awsize[2:0]	out	Burst size
lsu_axi_awburst[1:0]	out	Burst type *(hardwired to 0b01)*
lsu_axi_awlock	out	Lock type *(hardwired to 0)*
lsu_axi_awcache[3:0]	out	Memory type
lsu_axi_awprot[2:0]	out	Protection type *(hardwired to 0b000)*
lsu_axi_awqos[3:0]	out	Quality of Service (QoS) *(hardwired to 0b0000)*
lsu_axi_awregion[3:0]	out	Region identifier
Write data channel signals		
lsu_axi_wvalid	out	Write valid
lsu_axi_wready	in	Write ready
lsu_axi_wdata[63:0]	out	Write data
lsu_axi_wstrb[7:0]	out	Write strobes
lsu_axi_wlast	out	Write last
Write response channel signals		
lsu_axi_bvalid	in	Write response valid
lsu_axi_bready	out	Write response ready
lsu_axi_bid[RV_LSU_BUS_TAG-1:0]	in	Response ID tag
lsu_axi_bresp[1:0]	in	Write response
Read address channel signals		
lsu_axi_arvalid	out	Read address valid
lsu_axi_arready	in	Read address ready
lsu_axi_arid[RV_LSU_BUS_TAG-1:0]	out	Read address ID
lsu_axi_araddr[31:0]	out	Read address
lsu_axi_arlen[7:0]	out	Burst length *(hardwired to 0b0000_0000)*
lsu_axi_arsize[2:0]	out	Burst size
lsu_axi_arburst[1:0]	out	Burst type *(hardwired to 0b01)*
lsu_axi_arlock	out	Lock type *(hardwired to 0)*
lsu_axi_arcache[3:0]	out	Memory type
lsu_axi_arprot[2:0]	out	Protection type *(hardwired to 0b000)*
lsu_axi_arqos[3:0]	out	Quality of Service (QoS) *(hardwired to 0b0000)*
lsu_axi_arregion[3:0]	out	Region identifier
Read data channel signals		

Signal	Dir	Description
lsu_axi_rvalid	in	Read valid
lsu_axi_rready	out	Read ready
lsu_axi_rid[RV_LSU_BUS_TAG-1:0]	in	Read ID tag
lsu_axi_rdata[63:0]	in	Read data
lsu_axi_rresp[1:0]	in	Read response
lsu_axi_rlast	in	Read last
System Bus (Debug) Master AXI		
Write address channel signals		
sb_axi_awvalid	out	Write address valid
sb_axi_awready	in	Write address ready
sb_axi_awid[RV_SB_BUS_TAG-1:0]	out	Write address ID *(hardwired to 0)*
sb_axi_awaddr[31:0]	out	Write address
sb_axi_awlen[7:0]	out	Burst length *(hardwired to 0b0000_0000)*
sb_axi_awsize[2:0]	out	Burst size
sb_axi_awburst[1:0]	out	Burst type *(hardwired to 0b01)*
sb_axi_awlock	out	Lock type *(hardwired to 0)*
sb_axi_awcache[3:0]	out	Memory type *(hardwired to 0b1111)*
sb_axi_awprot[2:0]	out	Protection type *(hardwired to 0b000)*
sb_axi_awqos[3:0]	out	Quality of Service (QoS) *(hardwired to 0b0000)*
sb_axi_awregion[3:0]	out	Region identifier
Write data channel signals		
sb_axi_wvalid	out	Write valid
sb_axi_wready	in	Write ready
sb_axi_wdata[63:0]	out	Write data
sb_axi_wstrb[7:0]	out	Write strobes
sb_axi_wlast	out	Write last
Write response channel signals		
sb_axi_bvalid	in	Write response valid
sb_axi_bready	out	Write response ready
sb_axi_bid[RV_SB_BUS_TAG-1:0]	in	Response ID tag
sb_axi_bresp[1:0]	in	Write response
Read address channel signals		
sb_axi_arvalid	out	Read address valid
sb_axi_arready	in	Read address ready
sb_axi_arid[RV_SB_BUS_TAG-1:0]	out	Read address ID *(hardwired to 0)*
sb_axi_araddr[31:0]	out	Read address

Signal	Dir	Description
sb_axi_arlen[7:0]	out	Burst length *(hardwired to 0b0000_0000)*
sb_axi_arsize[2:0]	out	Burst size *(hardwired to 0b011)*
sb_axi_arburst[1:0]	out	Burst type *(hardwired to 0b01)*
sb_axi_arlock	out	Lock type *(hardwired to 0)*
sb_axi_arcache[3:0]	out	Memory type *(hardwired to 0b0000)*
sb_axi_arprot[2:0]	out	Protection type *(hardwired to 0b000)*
sb_axi_arqos[3:0]	out	Quality of Service (QoS) *(hardwired to 0b0000)*
sb_axi_arregion[3:0]	out	Region identifier
Read data channel signals		
sb_axi_rvalid	in	Read valid
sb_axi_rready	out	Read ready
sb_axi_rid[RV_SB_BUS_TAG-1:0]	in	Read ID tag
sb_axi_rdata[63:0]	in	Read data
sb_axi_rresp[1:0]	in	Read response
sb_axi_rlast	in	Read last
DMA Slave AXI		
Write address channel signals		
dma_axi_awvalid	in	Write address valid
dma_axi_awready	out	Write address ready
dma_axi_awid[RV_DMA_BUS_TAG-1:0]	in	Write address ID
dma_axi_awaddr[31:0]	in	Write address
dma_axi_awlen[7:0]	in	Burst length
dma_axi_awsize[2:0]	in	Burst size
dma_axi_awburst[1:0]	in	Burst type
dma_axi_awprot[2:0]	in	Protection type
Write data channel signals		
dma_axi_wvalid	in	Write valid
dma_axi_wready	out	Write ready
dma_axi_wdata[63:0]	in	Write data
dma_axi_wstrb[7:0]	in	Write strobes
dma_axi_wlast	in	Write last
Write response channel signals		
dma_axi_bvalid	out	Write response valid
dma_axi_bready	in	Write response ready
dma_axi_bid[RV_DMA_BUS_TAG-1:0]	out	Response ID tag
dma_axi_bresp[1:0]	out	Write response

Signal	Dir	Description
Read address channel signals		
dma_axi_arvalid	in	Read address valid
dma_axi_arready	out	Read address ready
dma_axi_arid[RV_DMA_BUS_TAG-1:0]	in	Read address ID
dma_axi_araddr[31:0]	in	Read address
dma_axi_arlen[7:0]	in	Burst length
dma_axi_arsize[2:0]	in	Burst size
dma_axi_arburst[1:0]	in	Burst type
dma_axi_arprot[2:0]	in	Protection type
Read data channel signals		
dma_axi_rvalid	out	Read valid
dma_axi_rready	in	Read ready
dma_axi_rid[RV_DMA_BUS_TAG-1:0]	out	Read ID tag
dma_axi_rdata[63:0]	out	Read data
dma_axi_rresp[1:0]	out	Read response
dma_axi_rlast	out	Read last
AHB-Lite		
Instruction Fetch Unit Master AHB-Lite		
Master signals		
haddr[31:0]	out	System address
hburst[2:0]	out	Burst type *(hardwired to 0b000)*
hmastlock	out	Locked transfer *(hardwired to 0)*
hprot[3:0]	out	Protection control
hsize[2:0]	out	Transfer size
htrans[1:0]	out	Transfer type
hwrite	out	Write transfer
Slave signals		
hrdata[63:0]	in	Read data
hready	in	Transfer finished
hresp	in	Slave transfer response
Load/Store Unit Master AHB-Lite		
Master signals		
lsu_haddr[31:0]	out	System address
lsu_hburst[2:0]	out	Burst type *(hardwired to 0b000)*
lsu_hmastlock	out	Locked transfer *(hardwired to 0)*
lsu_hprot[3:0]	out	Protection control

Signal	Dir	Description
lsu_hsize[2:0]	out	Transfer size
lsu_htrans[1:0]	out	Transfer type
lsu_hwdata[63:0]	out	Write data
lsu_hwrite	out	Write transfer
Slave signals		
lsu_hrdata[63:0]	in	Read data
lsu_hready	in	Transfer finished
lsu_hresp	in	Slave transfer response
System Bus (Debug) Master AHB-Lite		
Master signals		
sb_haddr[31:0]	out	System address
sb_hburst[2:0]	out	Burst type *(hardwired to 0b000)*
sb_hmastlock	out	Locked transfer *(hardwired to 0)*
sb_hprot[3:0]	out	Protection control
sb_hsize[2:0]	out	Transfer size
sb_htrans[1:0]	out	Transfer type
sb_hwdata[63:0]	out	Write data
sb_hwrite	out	Write transfer
Slave signals		
sb_hrdata[63:0]	in	Read data
sb_hready	in	Transfer finished
sb_hresp	in	Slave transfer response
DMA Slave AHB-Lite		
Master signals		
dma_haddr[31:0]	in	System address
dma_hburst[2:0]	in	Burst type
dma_hmastlock	in	Locked transfer
dma_hprot[3:0]	in	Protection control
dma_hsize[2:0]	in	Transfer size
dma_htrans[1:0]	in	Transfer type
dma_hwdata[63:0]	in	Write data
dma_hwrite	in	Write transfer
Slave signals		
dma_hrdata[63:0]	out	Read data
dma_hready	out	Transfer finished
dma_hresp	out	Slave transfer response

Signal	Dir	Description
Power Management Interface		
i_cpu_halt_req	in	Halt request to core (async)
o_cpu_halt_ack	out	Core acknowledgement for halt request (async)
o_cpu_halt_status	out	Core halted indication (async)
o_debug_mode_status	out	Core in debug mode indication (async)
i_cpu_run_req	in	Run request to core (async)
o_cpu_run_ack	out	Core acknowledgement for run request (async)
Performance Counter Activity		
dec_tlu_perfcnt0	out	Performance counter 0 incrementing
dec_tlu_perfcnt1	out	Performance counter 1 incrementing
dec_tlu_perfcnt2	out	Performance counter 2 incrementing
dec_tlu_perfcnt3	out	Performance counter 3 incrementing
Trace Port		
trace_rv_i_insn_ip[63:0]	out	Instruction opcode
trace_rv_i_address_ip[63:0]	out	Instruction address
trace_rv_i_valid_ip[2:0]	out	Instruction trace valid
trace_rv_i_exception_ip[2:0]	out	Exception
trace_rv_i_ecause_ip[4:0]	out	Exception cause
trace_rv_i_interrupt_ip[2:0]	out	Interrupt exception
trace_rv_i_tval_ip[31:0]	out	Exception trap value
JTAG Port		
jtag_tck	in	JTAG Test Clock (async)
jtag_tms	in	JTAG Test Mode Select (async)
jtag_tdi	in	JTAG Test Data In (async)
jtag_trst_n	in	JTAG Test Reset (async)
jtag_tdo	out	JTAG Test Data Out (async)
Memory Testing		
scan_mode	in	Enable MBIST for internal memories
mbist_mode	in	Chip select of all DCCM banks (for debug at SoC level)

14 SweRV EH1 Core Build Arguments

14.1 Core Memory-Related Build Arguments

14.1.1 Core Memories and Memory-Mapped Register Blocks Alignment Rules

Placement of SweRV EH1's core memories and memory-mapped register blocks in the 32-bit address range is very flexible. Each memory or register block may be assigned to any region and within the region's 28-bit address range to any start address on a naturally aligned power-of-two address boundary relative to its own size (i.e., *start_address* = *n* × *size*, whereas *n* is a positive integer number).

For example, the start address of an 8KB-sized DCCM may be 0x0000_0000, 0x0000_2000, 0x0000_4000, 0x0000_6000, etc. A memory or register block with a non-power-of-two size must be aligned to the next bigger power-of-two size. For example, the starting address of a 48KB-sized DCCM must aligned to a 64KB boundary, i.e., it may be 0x0000_0000, 0x0001_0000, 0x0002_0000, 0x0003_0000, etc.

Also, no two memories or register blocks may overlap each other, and no memory or register block may cross a region boundary.

The start address of the memory or register block is specified with an offset relative to the start address of the region. This offset must follow the rules described above.

14.1.2 Memory-Related Build Arguments

- **ICCM**
  - Enable (RV_ICCM_ENABLE): 0, 1 *(0 = no ICCM; 1 = ICCM enabled)*
  - Region (RV_ICCM_REGION): 0..15
  - Offset (RV_ICCM_OFFSET): *(offset in bytes from start of region satisfying rules in Section 14.1.1)*
  - Size (RV_ICCM_SIZE): 4, 8, 16, 32, 64, 128, 256, 512 *(in KB)*
- **DCCM**
  - Region (RV_DCCM_REGION): 0..15
  - Offset (RV_DCCM_OFFSET): *(offset in bytes from start of region satisfying rules in Section 14.1.1)*
  - Size (RV_DCCM_SIZE): 4, 8, 16, 32, 48, 64, 128, 256, 512 *(in KB)*
- **I-Cache**
  - Enable (RV_ICACHE_ENABLE): 0, 1 *(0 = no I-cache; 1 = I-cache enabled)*
  - Size (RV_ICACHE_SIZE): 16, 32, 64, 128, 256 *(in KB)*
  - Protection (RV_ICACHE_ECC): 0, 1 *(0 = parity; 1 = ECC)*
- **PIC Memory-mapped Control Registers**
  - Region (RV_PIC_REGION): 0..15
  - Offset (RV_PIC_OFFSET): *(offset in bytes from start of region satisfying rules in Section 14.1.1)*
  - Size (RV_PIC_SIZE): 32, 64, 128, 256 *(in KB)*

15 SweRV EH1 Errata

15.1 Core May Handle Write Transactions with Different Transaction IDs Incorrectly on AXI System Bus

Description:

The AXI protocol requires all transactions with a given transaction ID to remain ordered, but there is no ordering requirement for transactions with different transaction IDs. The core implementation currently does not correctly handle AXI write transactions with different transaction IDs.

Symptoms:

Two writes to the same address or overlapping addresses with different transaction IDs may not be handled correctly by the core.

Workaround:

Design fix is in progress. Expect release of design fix in early February, 2019.