Western Digital®

RISC-V SweRV™ EH1
Programmer's Reference Manual

Revision 1.8

September 18, 2020

Document Revision History

Revision	Date	Contents
1.0	Jan 24, 2019	Initial revision

1.1	May 31, 2019	• Updated 'Reference Documents' table:
		  • Updated link and version number of RISC-V ISA spec
		  • Updated link and version number of RISC-V Privileged spec, updated section references throughout text
		  • Added link and version number of last RISC-V Privileged spec with PLIC chapter
		  • Fixed URL and updated version number of RISC-V Debug spec
		• Added core pipeline summary (Section 1.3.1)
		• Corrected load-to-load ordering description (Section 2.5.1)
		• Added section on 'Bus Barrier' mechanism (Section 2.5.3.3) and updated instructions and data fencing sections accordingly (Sections 2.5.3.1 and 2.5.3.2)
		• Added section on 'Memory Protection' mechanism (Section 2.6)
		• Updated note when `mrac` access control bits are ignored (Section 2.8.1)
		• Clarified note how writing illegal value to `mrac` register is handled by hardware (Section 2.8.1)
		• Added region number to field names of `mrac` register to make them unique (Table 2-6)
		• Changed field name *fence.i* in `dmst` register to *fence_i* to avoid potential compatibility issues with tools (Table 2-7)
		• Added section on 'Speculative Bus Accesses' (Section 2.12)
		• Updated DMA QoS description (Section 2.13.3)
		• Added note that applied reset vector must be to valid and enabled memory address (Section 2.14)
		• Updated NMI description and added table of `mcause` values (Section 2.15)
		• Clarified comment about stuck-at bits (Section 3.4)
		• Corrected note regarding correctable error local interrupt not being latched (Sections 3.5.1, 3.5.2, and 3.5.3)
		• Updated Power Management chapter (Chapter 5):
		  • Changed title to 'Power Management and Multi-Core Debug Control'
		  • Added brief descriptions of power management unit (PMU) and multi-processor debug control (MPC) interfaces (Section 5.2)
		  • Clarified that only highest-priority external interrupt wakes up core (Figure 5-1)
		  • Updated note describing 'Core Quiesced' (Section 5.3)
		  • Added notes how to tie off input signals if PMU interface not used (Table 5-3)
		  • Added notes how to tie off input signals if MPC interface not used (Table 5-4)
		  • Updated cross-reference to `mhwakeup` signal description to be more precise (Section 5.4.7)
		• Clarified vectored external interrupt handler selection steps (Section 6.6)
		• Added source ID to field names of `meipX` register to make them unique (Table 6-3)
		• Clarified that event counting of division instructions includes remainder instructions (Table 7-2)
		• Fixed note on tag alignment (Table 8-2)
		• Updated `mfdc` register definition (Table 10-1):
		  • Updated field descriptions

Revision	Date	Contents
		• Assigned names to fields • Added 'DMA QoS control' field • Added 'side effect posted disable' bit • Removed 'PIC multiple interrupts disable' bit (was bit 9) • Removed 'Load miss bypass Write Buffer (WB) disable' bit (was bit 1) • Updated `mcgc` register definition (Table 10-2): • Updated field descriptions • Assigned names to fields • Improved clarity of `mcause` value table (Table 11-3) • Updated asynchronous signals (Table 14-1): • Removed core output signals • Added that JTAG signals are synchronous to TCK • Added asynchronous MPC interface signals • Updated port list (Table 15-1): • Removed '(async)' label from core output signals • Added missing DMA Slave AHB-Lite bus signals • Added MPC interface signals • Updated performance counter activity signals • Added that JTAG signals are synchronous to TCK • Added `jtag_id` port • Added 'Memory Protection Build Arguments' (Section 16.1) • Updated 'Errata' chapter (Chapter 18): • Added 'Back-to-back Write Transactions Not Supported on AHB-Lite Bus' section • Removed 'Core May Handle Write Transactions with Different Transaction IDs Incorrectly on AXI System Bus' section, issue has been fixed
1.2	Aug 13, 2019	• Updated bus barrier description (Section 2.5.3.3) • Updated ICCM/DCCM error detection and handling details (Table 2-4 and Table 2-5) • Added clarification that ordering between core and DMA accesses is not guaranteed (Section 2.13.4) • Updated ICCM/DCCM recovery/logging details (Table 3-2) • Clarified that correctable errors on DMA reads to ICCM/DCCM are counted (Sections 3.5.2 and 3.5.3) • Clarified that correctable DCCM errors counted only for retired load/store instructions (Section 3.5.3) • Changed 'RV_' prefix to '`RV_' (Table 14-1) • Updated port list (Table 15-1): • Changed 'RV_' prefix to '`RV_' • Added 'core_rst_l' signal • Removed 'mbist_mode' signal

Revision	Date	Contents

1.5	Feb 14, 2020	• Added footnote that PIC access errors also included (Table 2-2)
		• Clarified that correctable error local interrupt is level signaled (Sections 3.5.1, 3.5.2, and 3.5.3)
		• Fixed scope of Debug Mode in Core Activity States diagram (Figure 5-1)
		• Added several clarifications on MPC interface restrictions:
		  • Halt/run request typically allowed only when not in requested state already (Section 5.3)
		  • Signaling same request multiple times not allowed (Table 5-4)
		  • Conditions when requests are acknowledged (Section 5.4.2.2)
		  • After reset to Debug Mode, run request only allowed after core is in Debug Mode (Section 5.4.2.2)
		• Added Single Stepping section (Section 5.4.1.1)
		• Amended note regarding signaling PMU halt/run request when already in that state (Section 5.4.2.1)
		• Added note that interrupts must be disabled while changing some interrupt registers (Section 6.5)
		• Updated `mimpid` register value to ‘2’ (Table 12-1)
		• Added standard CSR address map (Table 12-2)
		• Updated port list (Table 15-1):
		  • Added `dbg_rst_l` signal
		  • Removed `core_rst_l` signal (signal on core periphery, but not core complex periphery)
		  • Removed `sb_axi_arsize` bus description comment indicating ‘*hardwired*’
		  • Added `mbist_mode` signal (signal on core complex periphery, but not core periphery)
		• Added ‘Compliance Test Suite Failures’ chapter (Chapter 17)
		• Added erratum for debug access register abstract command issue (Section 16.2)

1.5.1	Feb 28, 2020	• Added note that uninitialized DCCM may cause loads to get incorrect data (Section 3.4) • Added Debug Module reset description (Section 14.3.2) • Added footnote clarifying trace port signals (Table 15-1) • Added erratum for access register abstract command size check issue (Section 16.3)

Revision	Date	Contents

1.6	May 15, 2020	• Added footnote that misaligned accesses to side-effect regions trigger a misaligned exception instead of the recommended access fault exception (Table 2-3)
		• Fixed note how writing illegal value to `mrac` register is handled by hardware (Section 2.8.1)
		• Added Internal Timers chapter and references throughout document (Chapter 4)
		• Added cross-references to debug CSR descriptions (Table 5-2, Table 5-4, Table 12-2, and Sections 7.4 and 14.3.4)
		• Added Debug Support chapter (Chapter 9)
		• Incremented `mimpid` register value from '2' to '3' (Table 12-1)
		• Updated 'Errata' chapter (Chapter 18):
		  • Removed erratum for debug access register abstract command issue (fixed) (was Section 16.2)
		  • Removed erratum for access register abstract command size check issue (fixed) (was Section 16.3)
		  • Added erratum for debug write to `minstret` register issue (Section 18.2)
		  • Added erratum for abstract command register read capability (Section 18.3)

1.7	Jun 25, 2020	• Updated versions of RISC-V Base ISA [1] and Privileged [2] documents (Reference Documents)
		• Added description of SoC access expectation (Section 2.11)
		• Added note that `mitcnt0`/`1` register is cleared if internal timer interrupt coincides with write to it (Section 4.4.1)
		• Amended `debug_mode_status` signal description (Table 5-4)
		• Clarified effect of *sespd* bit of `mfdc` register (Table 10-1)
		• Debug Support chapter updates (Chapter 9):
		  • Fixed 'Access' of JTAG BYPASS register since not directly accessible (Table 9-5)
		  • Fixed abstract command register definition (Table 9-11):
		    • Changed 'W' accesses to 'R0/W'
		    • Fixed *aarsize* and *aamsize* field descriptions that command error is '2'
		    • Updated *aarpostincrement*, *postexec*, *transfer*, and *aampostincrement* bit descriptions including error behavior
		    • Added note to *aamvirtual* bit description that no error is flagged
		  • Fixed reset value of *sbaccess* field (Table 9-14)
		  • Updated description of *sbautoincrement* field that only incrementing for successful accesses (Table 9-14)
		  • Added note that no bus transaction is issued on debug execute address trigger for side-effect load (Table 9-20)
		  • Added footnote that bit 0 is ignored for instruction address matches (Table 9-20 and Table 9-21)
		• Incremented `mimpid` register value from '3' to '4' (Table 12-1)

Revision	Date	Contents

1.8	Sep 18, 2020	• Added note that NMIs are fatal (Section 2.15)
		• Clarified note that debug single-step action is delayed while MPC debug halted (Section 5.3)
		• Added note that debug single-stepping stays pending while MPC debug halted (Section 5.4.1.1)
		• Added note that `mpc_debug_run_req` is required to exit Debug Mode if entered after reset using `mpc_reset_run_req` (Section 5.4.2.2)
		• Added *haltie* control bit to `mpmc` register (Section 5.5.1)
		• Added note that edge-triggered interrupt lines must be tied off to inactive state (Section 6.3.2)
		• Removed outdated 'Full Hardware Implementation of Vectored External Interrupts' section (was Section 6.6.1)
		• Fixed gateway initialization macro example (Section 6.14.2)
		• Added note that `mtime` and `mtimecmp` registers must be provided by SoC (Section 7.2.1)
		• Added note that *index* field does not have WARL behavior (Table 8-1)
		• Added notes that abstract commands may only be executed when core is in debug halt state (Sections 9.1.2 and 9.1.2.5)
		• Added notes that system bus accesses are allowed irrespective of core's state (Sections 9.1.2 and 9.1.2.8)
		• Added description of abstract command enhancements:
		  • Updated notes that SoC memory locations are accessible using access memory abstract command as well (Sections 9.1.2 and 9.1.2.5)
		  • Updated *cmderr* field description (Table 9-10)
		  • Updated abstract command description (Table 9-11):
		    • Clarified that selecting unsupported abstract command causes failure
		    • Updated *aarsize* and *aamsize* field descriptions
		    • Updated *aarpostincrement* and *aampostincrement* bit descriptions
		    • Updated *regno* field description
		  • Added `abstractauto` register (Section 9.1.2.6)
		  • Added note that selecting unmapped access memory abstract command address causes failure (Section 9.1.2.7)
		• Added footnote to *aamvirtual* bit documenting why no command error is reported (Table 9-11)
		• Added description of `sbaddress0` register write access action and error condition behavior (Section 9.1.2.9)
		• Corrected description of `sbdata0` register read and write access action (Section 9.1.2.10)
		• Clarified that triggering on load data or executed instruction opcode not supported (Section 9.1.3.3 and Table 9-20)
		• Clarified that triggers do not fire if *action* is '0' and interrupts disabled (Table 9-20)
		• Updated `tdata2` register description (Table 9-21)
		• Incremented `mimpid` register value from '4' to '5' (Table 12-1)
		• Updated 'Reset to Debug-Mode' description (Section 14.3.4)

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	20190608-Base-Ratified	Specification ratified
2	The RISC-V Instruction Set Manual Volume II: Privileged Architecture	20190608-Priv-MSU-Ratified	Specification ratified
2 (PLIC)	The RISC-V Instruction Set Manual Volume II: Privileged Architecture	1.11-draft December 1, 2018	Last specification version with PLIC chapter
3	RISC-V External Debug Support	0.13.2	Specification 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
RAS	Return Address Stack
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 machine-mode (M-mode) only, 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. 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:

- RV32IMC-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 AXI4 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.

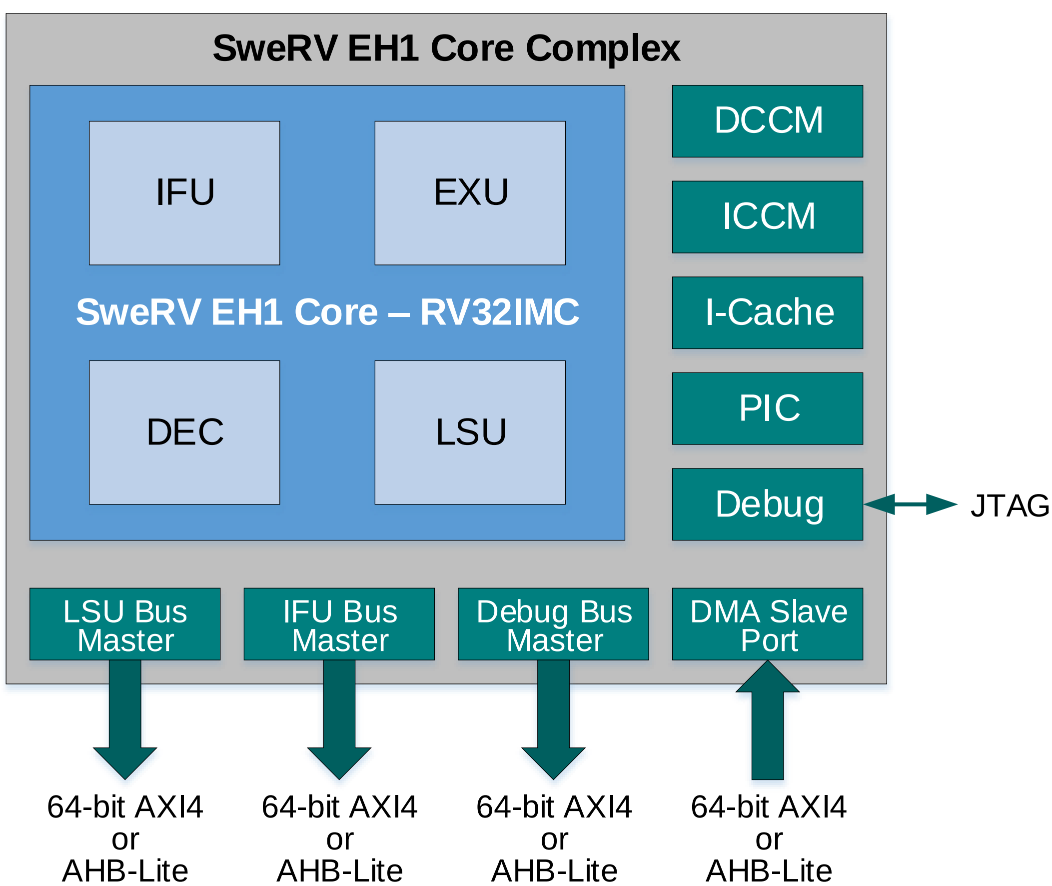


Figure 1-1 SweRV EH1 Core Complex

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 superscalar, dual-issue 9-stage core pipeline supporting four arithmetic logic units (ALUs) labeled EX1 and EX4 in two pipelines I0 and I1, one load/store pipeline, one 3-cycle latency multiplier pipeline, and one out-of-pipeline 34-cycle latency divider. There are four stall points in the pipeline: 'Fetch1', 'Align', 'Decode', and 'Commit'. In the 'Align' stage, instructions are formed from 3 fetch buffers. In the 'Decode' stage, up to 2 instructions from 4 instruction buffers are decoded. In the 'Commit' stage, up to 2 instructions per cycle are committed. Finally, in the 'Writeback' stage, the architectural registers are updated.

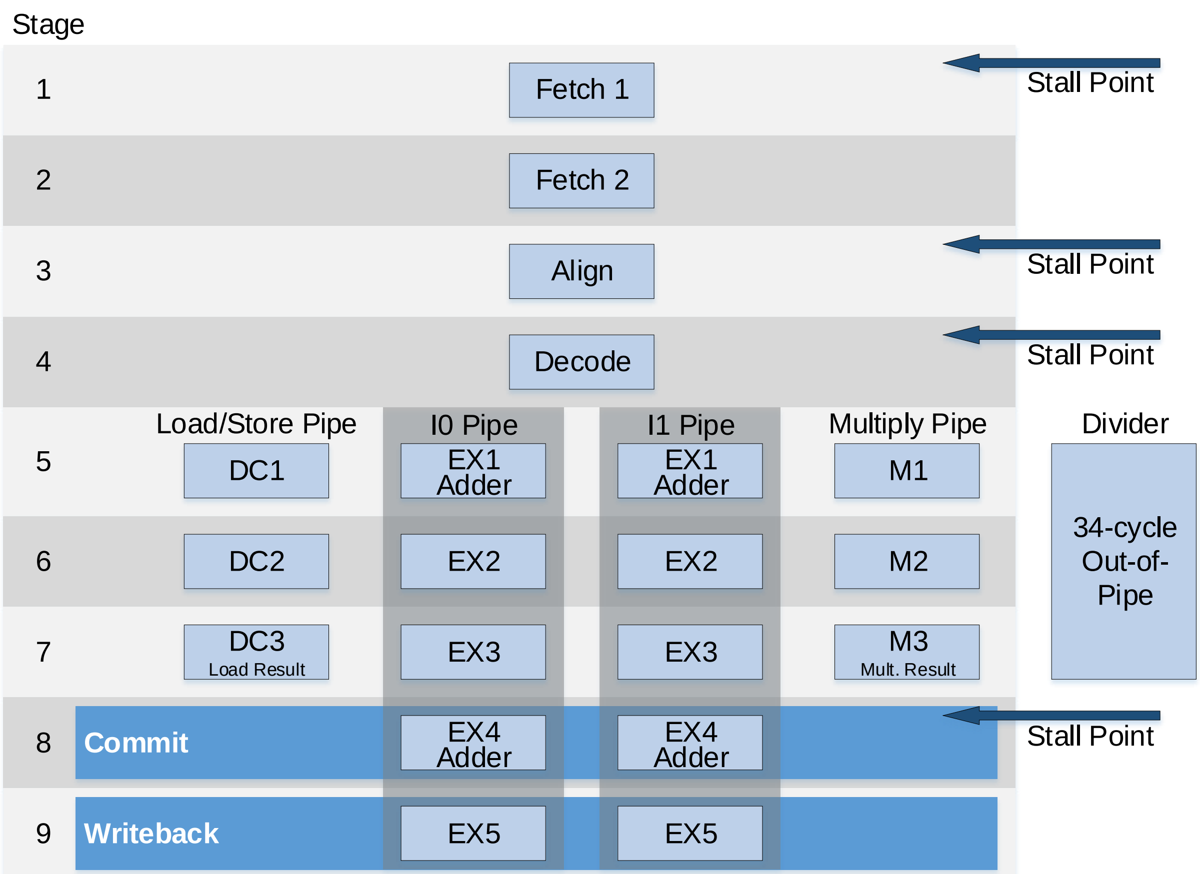


Figure 1-2 SweRV EH1 Core Pipeline

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 (see Section 2.8.1).

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

All loads are sent to the system bus interface in program order. Also, all stores are 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 emptied, 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 neither freeze the pipe nor 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 emptied, 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 Sections 2.5.3.2 and 2.5.3.3).

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, the store and write buffers have been fully drained (i.e., are empty), and the bus barrier (see Section 2.5.3.3) is finished. The `fence` instruction is only committed after all LSU buffers are idle and all outstanding bus transactions are completed.

2.5.3.3 Bus Barrier

SweRV EH1 provides a bus barrier mechanism. Executing a `fence` instruction forces a bus synchronization action which requires all outstanding bus transactions (reads and writes) for the LSU bus master to complete.

Hardware uses an 8-bit counter with which it continuously keeps track of the number of outstanding bus transactions. For every request sent, this counter is incremented; for every response received, this counter is decremented. The maximum number of outstanding bus transactions is 255. If this limit is reached, no further transactions are sent to the bus until the number of outstanding bus transactions is smaller than 255. A bus barrier requires the count to reach 0 before the barrier is finished.

Loads are not allowed to be forwarded across an older bus barrier. The LSU enforces this within the core pipeline. Also, the LSU does not forward from the write buffer if the buffer itself contains a bus barrier.

The `fence` instruction leverages the semantics of the bus barrier. A `fence` instruction waits for all prior bus transactions to finish in addition to the write buffer being fully drained before proceeding. Instructions after a `fence.i` are guaranteed to see previous writes in the case of self-modifying code.

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 2.8.4 for more details), but this may result in unexpected behavior.

2.6 Memory Protection

To eliminate issuing speculative accesses to the IFU and LSU bus interfaces, SweRV EH1 provides a rudimentary memory protection mechanism for instruction and data accesses outside of the ICCM and DCCM memory regions. Separate core build arguments for instructions and data are provided to enable and configure up to 8 address windows each.

An instruction fetch to a non-ICCM region must fall within the address range of at least one instruction access window for the access to be forwarded to the IFU bus interface. If at least one instruction access window is enabled, non-speculative fetch requests which are not within the address range of any enabled instruction access window cause a precise instruction access fault exception. If none of the 8 instruction access windows is enabled, the memory protection mechanism for instruction accesses is turned off. For the ICCM region, accesses within the ICCM's address range are allowed. However, any access not within the ICCM's address range results in a precise instruction access fault exception.

Similarly, a load/store access to a non-DCCM or non-PIC memory-mapped control register region must fall within the address range of at least one data access window for the access to be forwarded to the LSU bus interface. If at least one data access window is enabled, non-speculative load/store requests which are not within the address range of any enabled data access window cause a precise load/store address misaligned or access fault exception. If none of the 8 data access windows is enabled, the memory protection mechanism for data accesses is turned off. For the DCCM and PIC memory-mapped control register region(s), accesses within the DCCM's or the PIC memory-mapped control register's address range are allowed. However, any access not within the DCCM's or PIC memory-mapped control register's address range results in a precise load/store address misaligned or access fault exception.

The configuration parameters for each of the 8 instruction and 8 data access windows are:

- Enable/disable instruction/data access window 0..7,
- a base address of the window (which must be 64B-aligned), and
- a mask specifying the size of the window (which must be an integer-multiple of 64 bytes minus 1).

See Section 16.1 for more information.

2.7 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 3.1.17 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 2.8.3).

2.7.1 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 2.8.3). 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 2.8.4). 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.

2.7.2 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 11-2) and `mie` (see Table 11-1) registers, respectively. The priority is lower than

RISC-V External interrupt, but higher than RISC-V Timer interrupt (see Table 13-1). The correctable error local interrupt has an `mcause` value of 0x8000_001E (see Table 11-3).

2.7.3 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.

2.7.4 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]	
Load	Core	No	Load access fault exception[4,5]	Precise exception (e.g., address out-of-range)
	Bus	No (for non-blocking load)	Non-blocking load bus error NMI (see Section 2.7.1)	• 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[6] access fault exception[4,5]	Precise exception
	Bus	No	Store bus error NMI (see Section 2.7.1)	• 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] 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.

[4] Exception also flagged for PIC load/store not word-sized or address not word-aligned.

[5] Exception also flagged for loads/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.

[6] AMO refers to the RISC-V "A" (atomics) extension, which is not implemented in SweRV EH1.

2.7.5 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			
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[8]		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[8]		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[9]					

[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.

[8] The RISC-V Privileged specification recommends that misaligned accesses to regions with potential side-effects should trigger an access fault exception, instead of a misaligned exception (see Section 3.5.6 in [2]). Note that SweRV EH1 triggers a misaligned exception in this case. To avoid potential side-effects, the exception handler should not emulate a misaligned access using multiple smaller aligned accesses.

[9] This case is in violation with the write alignment rules specified in Section 2.13.2.

2.7.6 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 (i.e., for oldest instruction in pipeline only)
	Bus	N/A		
Load	Core	No	Load access fault exception	Precise exception (i.e., for non-speculative load only)
		Yes		
	Bus	No (for non-blocking load)	Non-blocking load bus error NMI (see Section 2.7.1)	• Imprecise, fatal • Capture store address in core bus interface
		No (for blocking load)	Load access fault exception	Precise exception
		Yes		
Store	Core	No	Store/AMO access fault exception	Precise exception (i.e., for non-speculative store only)
		Yes		
	Bus	No	Store bus error NMI (see Section 2.7.1)	• 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.

2.7.7 Correctable ECC/Parity Errors

Table 2-5 Handling of Correctable ECC/Parity Errors

<table>
<tr><th>Access</th><th>Core/Bus</th><th>Side Effect</th><th>Action</th><th>Comments</th></tr>
<tr><td rowspan="3">Fetch</td><td rowspan="2">Core</td><td rowspan="2">N/A</td><td>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</td><td>• For all fetches from I-cache (i.e., out of pipeline, independent of actual instruction execution)
• For I-cache with tag/instruction ECC protection, single- and double-bit errors are recoverable</td></tr>
<tr><td>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</td><td>• For all fetches from ICCM (i.e., out of pipeline, independent of actual instruction execution)
• ICCM errors trigger an RFPC (ReFetch PC) flush since in-line correction would require an additional cycle</td></tr>
<tr><td>Bus</td><td>N/A</td><td>• Increment correctable error counter in SoC
• If error threshold reached, signal external interrupt
• Write corrected data back to SoC memory</td><td>Errors in SoC memories are corrected at memory boundary and autonomously written back to memory array</td></tr>
<tr><td rowspan="4">Load</td><td rowspan="2">Core</td><td>No</td><td rowspan="2">• 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</td><td rowspan="2">• For non-speculative accesses only
• DCCM errors are in-line corrected and written back to DCCM</td></tr>
<tr><td>Yes</td></tr>
<tr><td rowspan="2">Bus</td><td>No</td><td rowspan="2">• Increment correctable error counter in SoC
• If error threshold reached, signal external interrupt
• Write corrected data back to SoC memory</td><td rowspan="2">Errors in SoC memories are corrected at memory boundary and autonomously written back to memory array</td></tr>
<tr><td>Yes</td></tr>
</table>

Access	Core/Bus	Side Effect	Action	Comments
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	• For non-speculative accesses only • DCCM errors are in-line 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-line 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-line 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.

2.8 Control/Status Registers

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

- Region Access Control Register (mrac) (see Section 2.8.1)
- Memory Synchronization Trigger Register (dmst) (see Section 2.8.2)
- D-Bus First Error Address Capture Register (mdseac) (see Section 2.8.3)
- D-Bus Error Address Unlock Register (mdeau) (see Section 2.8.4)

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.

2.8.1 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 *sideeffect* access control bits are ignored by the core for load/store accesses to addresses mapped to core-local memories (i.e., DCCM and ICCM) and PIC memory-mapped control registers as well as for all instruction fetch accesses. The *cacheable* access control bits are ignored for instruction fetch accesses from addresses mapped to the ICCM, but not for any other addresses.

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*	*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*	*Y**2	Caching control for region *Y*: 0: Caching not allowed 1: Caching allowed	R/W	0

2.8.2 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

2.8.3 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 2.8.4).

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

2.8.4 D-Bus Error Address Unlock Register (mdeau)

Writing to the `mdeau` register unlocks the `mdseac` register (see Section 2.8.3) 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

2.9 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)

<table>
<tr><th>Region</th><th>Start Address</th><th>End Address</th><th>Memory Type</th></tr>
<tr><td rowspan="5">0x0</td><td>0x0000_0000</td><td>0x0003_FFFF</td><td>Reserved</td></tr>
<tr><td>0x0004_0000</td><td>0x0005_FFFF</td><td>ICCM (region: 0, offset: 0x4000, size: 128KB)</td></tr>
<tr><td>0x0006_0000</td><td>0x0007_FFFF</td><td>Reserved</td></tr>
<tr><td>0x0008_0000</td><td>0x0009_FFFF</td><td>DCCM (region: 0, offset: 0x8000, size: 128KB)</td></tr>
<tr><td>0x000A_0000</td><td>0x0FFF_FFFF</td><td>Reserved</td></tr>
<tr><td>0x1</td><td>0x1000_0000</td><td>0x1FFF_FFFF</td><td>System memory-mapped CSRs</td></tr>
<tr><td>0x2</td><td>0x2000_0000</td><td>0x2FFF_FFFF</td><td rowspan="10">System SRAMs,
system ROMs, and
system memory-mapped I/O device interfaces</td></tr>
<tr><td>0x3</td><td>0x3000_0000</td><td>0x3FFF_FFFF</td></tr>
<tr><td>0x4</td><td>0x4000_0000</td><td>0x4FFF_FFFF</td></tr>
<tr><td>0x5</td><td>0x5000_0000</td><td>0x5FFF_FFFF</td></tr>
<tr><td>0x6</td><td>0x6000_0000</td><td>0x6FFF_FFFF</td></tr>
<tr><td>0x7</td><td>0x7000_0000</td><td>0x7FFF_FFFF</td></tr>
<tr><td>0x8</td><td>0x8000_0000</td><td>0x8FFF_FFFF</td></tr>
<tr><td>0x9</td><td>0x9000_0000</td><td>0x9FFF_FFFF</td></tr>
<tr><td>0xA</td><td>0xA000_0000</td><td>0xAFFF_FFFF</td></tr>
<tr><td>0xB</td><td>0xB000_0000</td><td>0xBFFF_FFFF</td></tr>
</table>

Region	Start Address	End Address	Memory Type
0xC	0xC000_0000	0xCFFF_FFFF	
0xD	0xD000_0000	0xDFFF_FFFF	
0xE	0xE000_0000	0xEFFF_FFFF	
0xF	0xF000_0000	0xFFFF_FFFF	

2.10 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.

2.11 Expected SoC Behavior for Accesses

The SweRV EH1 core expects that the SoC responds to all system bus access requests it receives from the core. System bus accesses include instruction fetches, load/store data accesses as well as debug system bus accesses. A response may either be returning the requested data (e.g., instructions sent back to the core for fetches or data for loads), an acknowledgement indicating the successful completion of a bus transaction (e.g., acknowledging a store), or an error response (e.g., an error indication in response to an attempt to access an unmapped address). If the SoC does not respond to every single bus transaction, the core may hang.

2.12 Speculative Bus Accesses

Deep core pipelines require a certain degree of speculation to maximize performance. The sections below describe instruction and data speculation in the SweRV EH1 core.

Note that speculative accesses to memory addresses with side effects may be entirely avoided by adding the build-argument-selected and -configured memory protection mechanism described in Section 2.6.

2.12.1 Instructions

Instruction cache misses on SweRV EH1 are speculative in nature. The core may issue speculatively fetch accesses on the IFU bus interface for an instruction cache miss in the following cases:

- due to an earlier exception or interrupt,
- due to an earlier branch mispredict,
- due to an incorrect branch prediction, and
- due to an incorrect Return Address Stack (RAS) prediction.

Issuing speculative accesses on the IFU bus interface is benign as long as the platform is able to handle accesses to unimplemented memory and to prevent accesses to SoC components with read side effects by returning random data and/or a bus error condition. The decision of which addresses are unimplemented and which addresses with potential side effects need to be protected is left to the platform.

Instruction fetch speculation can be limited, though not entirely avoided, by turning off the core's branch predictor including the return address stack. Writing a '1' to the *bpd* bit in the `mfdc` register (see Table 10-1) disables branch prediction including RAS.

2.12.2 Data

The SweRV EH1 core does not issue any speculative data accesses on the LSU bus interface.

2.13 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.

2.13.1 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.

2.13.2 Write Alignment Rules

For writes to the ICCM and DCCM through the DMA slave port, accesses 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.

2.13.3 Quality of Service

Accesses to the ICCM and DCCM by the core have higher priority if the DMA FIFO is not full. 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.

The *dqc* field in the `mfdc` register (see Table 10-1) specifies the maximum number of clock cycles a DMA access request waits at the head of the DMA FIFO before requesting a bubble to access the pipe. For example, if *dqc* is 0, a DMA access requests a bubble immediately (i.e., in the same cycle); if *dqc* is 7 (the default value), a waiting DMA access requests a bubble on the 8th cycle. For a DMA access to the ICCM, it may take up to 3 additional cycles[10] before the access is granted. Similarly, for a DMA access to the DCCM, it may take up to 4 additional cycles[11] before the access is granted.

2.13.4 Ordering of Core and DMA Accesses

Accesses to the DCCM or ICCM by the core and the DMA slave port are asynchronous events relative to one another. There are no ordering guarantees between the core and the DMA slave port accessing the same or different addresses.

2.14 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 deasserted (see also Section 14.3). When the core is reset, it fetches the first instruction to be executed from the address provided on the reset vector bus. Note that the applied reset vector must be pointing to the ICCM, if enabled, or a valid memory address, which is within an enabled instruction access window if the memory protection mechanism (see Section 2.6) is used.

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

2.15 Non-Maskable Interrupt (NMI) Signal and Vector

The core provides a 31-bit wide input bus at its periphery for a non-maskable interrupt (NMI) vector. The SoC must provide the NMI vector on the `nmi_vec[31:1]` bus, either hardwired or sourced from a register.

Note: NMI is entirely separate from the other interrupts and not affected by the selection of Direct vs Vectored mode.

The SoC may trigger an NMI by asserting the low-to-high edge-triggered, asynchronous `nmi_int` input signal. This signal 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 deasserted for a

[10] More cycles may be needed in the uncommon case of the pipe currently handling a correctable ECC error for a core fetch request, which needs to be finished first.

[11] If the core pipeline is currently frozen, the DMA access is further delayed until the freeze condition is resolved.

minimum of two full core clock cycles and then reasserted to signal the next NMI request to the core. If the SoC does not use the pin-asserted NMI feature, it must hardwire the `nmi_int` input signal to 0.

In addition to NMIs triggered by the SoC, a core-internal NMI request is signaled when a D-bus store or non-blocking load error has been detected.

When the core receives either an SoC-triggered or a core-internal NMI request, it fetches the next instruction to be executed from the address provided on the NMI vector bus. The reason for the NMI request is reported in the `mcause` register according to Table 2-11.

Table 2-11 Summary of NMI mcause Values

Value mcause[31:0]	Description
0x0000_0000	NMI pin assertion (`nmi_int` input signal, see above)
0xF000_0000	Machine D-bus store error NMI (see Section 2.7.1)
0xF000_0001	Machine D-bus non-blocking load error NMI (see Section 2.7.1)

Note: NMIs are typically fatal! Section 3.4 of the RISC-V Privileged specification [2] states that NMIs are only used for hardware error conditions and cause an immediate jump to the address at the NMI vector running in M-mode regardless of the state of a hart's interrupt enable bits. The NMI can thus overwrite state in an active M-mode interrupt handler and normal program execution cannot resume. Unlike resets, NMIs do not reset hart state, enabling diagnosis, reporting, and possible containment of the hardware error. Because NMIs are not maskable, the NMI handling routine performing diagnosis and reporting is itself susceptible to further NMIs, possibly making any such activity meaningless and erroneous in the face of error storms.

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 6-1 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’.

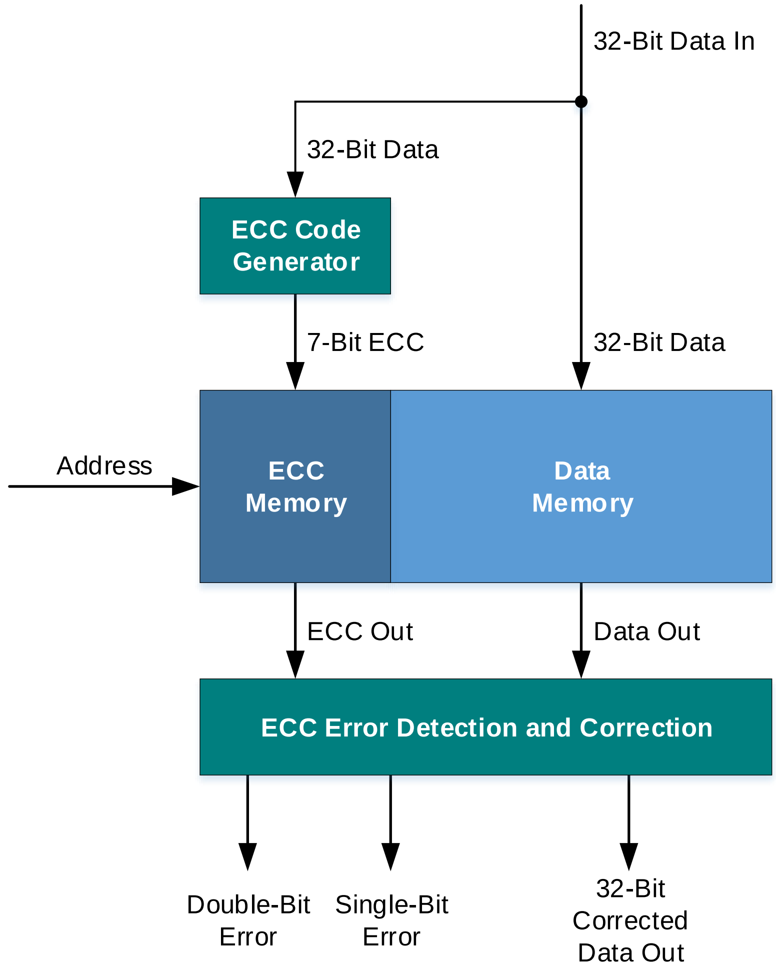


Figure 3-1 Conceptual Block Diagram – ECC in a Memory System

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

<table>
<tr><th>Memory Type</th><th>Abbreviation</th><th>Protection</th><th>Reason/Justification</th></tr>
<tr><td>Instruction Cache</td><td>I-cache</td><td>Parity or SEDDED ECC[12] (data and tag)</td><td>• Instructions can be refetched if error is detected</td></tr>
<tr><td>Instruction Closely-Coupled Memory</td><td>ICCM</td><td rowspan="3">SECDED ECC</td><td rowspan="3">• Large SRAM arrays
• Data could be modified and is only valid copy</td></tr>
<tr><td>Data Closely-Coupled Memory</td><td>DCCM</td></tr>
<tr><td>Core-complex-external Memories</td><td>SoC memories</td></tr>
</table>

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).

Note: If the DCCM is uninitialized, a load following a store to the same DCCM address may get incorrect data. If firmware initializes the DCCM, aligned word-sized stores should be used (because they don't check ECC), followed by a fence, before any load instructions to DCCM addresses are executed.

[12] 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 EH1 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[13]
• 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[14]</td><td colspan="2">• Increment I-cache correctable error counter[13]
• 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[15]:
• Write corrected data/ECC back to ICCM
• Refetch instruction from ICCM[14]</td><td rowspan="2">Fatal error[16] (uncorrectable)</td><td rowspan="2">• Increment[15] 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[16]:
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>
</table>

[13] 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.

[14] 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.

[15] All single-bit errors detected on fetches are corrected, written back to the ICCM, and counted, independent of actual instruction execution.

[16] For oldest instruction in pipeline only.

Memory Type	Detection	Recovery		Logging	
		Single-bit Error	Double-bit Error	Single-bit Error	Double-bit Error
DCCM	• Each 32-bit chunk protected with 7 ECC bits • ECC checked in pipeline	• Correct error in-line • Write[17] corrected data/ECC back to DCCM	Fatal error[18] (uncorrectable)	• Increment[17] DCCM single-bit error counter • If error counter has reached threshold, signal correctable error local interrupt (see Section 3.5.3)	For loads[18]: Load access fault exception For stores[18]: Store/AMO access fault exception For DMA reads: Send error response on DMA slave bus to master
SoC memories	ECC checked at SoC 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 2.7.1)

General comments:

- No address information of each individual correctable error is captured.
- Stuck-at faults:
  - Stuck-at bits would cause the correctable error threshold to be reached relatively quickly but are only reported if interrupts are enabled.
  - Use MBIST to determine exact location of the bad bit.
  - Because ICCM single-bit errors on fetches are not in-line corrected, a stuck-at bit may cause the core to hang.

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.

[17] For load/store accesses, the corrected data is written back to the DCCM and counted only if the load/store instruction retires (i.e., access is non-speculative and has no exception).

[18] For non-speculative accesses only.

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 is '1', a correctable error local interrupt (see Section 2.7.2) 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 2^{thresh} errors. If the error rate is high and the threshold is set to a low value, the interrupt may be missed but the counter value is not lost. 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]* is '1', signal correctable error local interrupt (see Section 2.7.2).	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 either an instruction fetch or a DMA read 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 is '1', a correctable error local interrupt (see Section 2.7.2) is signaled.

Hardware increments the *count* field on a detected single-bit 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 2^{thresh} errors. If the error rate is high and the threshold is set to a low value, the interrupt may be missed but the counter value is not lost. 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]* is '1', signal correctable error local interrupt (see Section 2.7.2).	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 either a retired load/store instruction or a DMA read access 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 is '1', a correctable error local interrupt (see Section 2.7.2) is signaled.

Hardware increments the *count* field on a detected single-bit error for a retired load or store instruction (i.e., a non-speculative access with no exception) or a DMA read. 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 2^{thresh} errors. If the error rate is high and the threshold is set to a low value, the interrupt may be missed but the counter value is not lost. 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]* is '1', signal correctable error local interrupt (see Section 2.7.2).	R/W	0

4 Internal Timers

This chapter describes the internal timer feature of the SweRV EH1 core.

4.1 Features

The SweRV EH1's internal timer features are:

- Two independently controlled 32-bit timers
  - Dedicated counter
  - Dedicated bound
  - Dedicated control to enable/disable incrementing generally, during power management Sleep, and while executing PAUSE
  - Enable/disable local interrupts (in standard RISC-V `mie` register)

4.2 Description

The SweRV EH1 core implements two internal timers. The `mitcnt0` and `mitcnt1` registers (see Section 4.4.1) are 32-bit unsigned counters. Each counter also has a corresponding 32-bit unsigned bound register (i.e., `mitb0` and `mitb1`, see Section 4.4.2) and control register (i.e., `mitctl0` and `mitctl1`, see Section 4.4.3).

All registers are cleared at reset unless otherwise noted. After reset, the counters start incrementing the next clock cycle if the increment conditions are met. All registers can be read as well as written at any time. The `mitcnt0/1` and `mitb0/1` registers may be written to any 32-bit value. If the conditions to increment are met, the corresponding counter `mitcnt0/1` increments every clock cycle.

For each timer, a local interrupt (see Section 4.3) is triggered when that counter is at or above its bound. When a counter is at or above its bound, it gets cleared the next clock cycle (i.e., the interrupt condition is not sticky).

Note: If the core is in Debug Mode and being single-stepped, it may take multiple clock cycles to execute a single instruction. If the conditions to increment are met, the counter increments for every clock cycle it takes to execute a single instruction. Therefore, every executed single-stepped instruction in Debug Mode may result in multiple counter increments.

Note: If the core is in the Debug Mode's Halted (i.e., db-halt) state, an internal timer interrupt does not transition the core back to the Active (i.e., Running) state.

4.3 Internal Timer Local Interrupts

Local-to-the-core interrupts for internal timer 0 and 1 have pending[19] (*mitip0*/*1*) and enable (*mitie0*/*1*) bits in bit positions 29 (for internal timer 0) and 28 (for internal timer 1) of the standard RISC-V `mip` (see Table 11-2) and `mie` (see Table 11-1) registers, respectively. The priority is lower than the RISC-V External, Software, and Timer interrupts (see Table 13-1). The internal timer 0 and 1 local interrupts have an `mcause` value of 0x8000_001D (for internal timer 0) and 0x8000_001C (for internal timer 1) (see Table 11-3).

Note: If both internal timer interrupts occur in the same cycle, internal timer 0's interrupt has higher priority than internal timer 1's interrupt.

Note: A common interrupt service routine may be used for both interrupts. The `mcause` register value differentiates the two local interrupts.

4.4 Control/Status Registers

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

- Internal Timer Counter 0 / 1 Register (mitcnt0/1) (see Section 4.4.1)
- Internal Timer Bound 0 / 1 Register (mitb0/1) (see Section 4.4.2)

[19] Since internal timer interrupts are not latched (i.e., not "sticky") and these local interrupts are only signaled for one core clock cycle, it is unlikely that they are detected by firmware in the `mip` register.

- Internal Timer Control 0 / 1 Register (mitctl0/1) (see Section 4.4.3)

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 Internal Timer Counter 0 / 1 Register (mitcnt0/1)

The `mitcnt0` and `mitcnt1` registers are the counters of the internal timer 0 and 1, respectively.

The conditions to increment a counter are:

- The *enable* bit in the corresponding `mitctl0/1` register is ‘1’,
- if the core is in Sleep (i.e., pmu/fw-halt) state, the *halt_en* bit in the corresponding `mitctl0/1` register is ‘1’,
- if the core is paused, the *pause_en* bit in the corresponding `mitctl0/1` register is ‘1’, and
- the core is not in Debug Mode, except while executing a single-stepped instruction.

A counter is cleared if its value is greater than or equal to its corresponding `mitb0/1` register.

Note: If a write to the `mitcnt0/1` register is committed in the same clock cycle as the timer interrupt condition is met, the internal timer local interrupt is triggered, if enabled, and the counter is cleared. In this case, the counter is not set to the written value.

These registers are mapped to the non-standard read/write CSR address space.

Table 4-1 Internal Timer Counter 0 / 1 Register (mitcnt0/1, at CSR 0x7D2 / 0x7D5)

Field	Bits	Description	Access	Reset
count	31:0	Counter	R/W	0

4.4.2 Internal Timer Bound 0 / 1 Register (mitb0/1)

The `mitb0` and `mitb1` registers hold the upper bounds of the internal timer 0 and 1, respectively.

These registers are mapped to the non-standard read/write CSR address space.

Table 4-2 Internal Timer Bound 0 / 1 Register (mitb0/1, at CSR 0x7D3 / 0x7D6)

Field	Bits	Description	Access	Reset
bound	31:0	Bound	R/W	0xFFFF_FFFF

4.4.3 Internal Timer Control 0 / 1 Register (mitctl0/1)

The `mitctl0` and `mitctl1` registers provide the control bits of the internal timer 0 and 1, respectively.

These registers are mapped to the non-standard read/write CSR address space.

Table 4-3 Internal Timer Control 0 / 1 Register (mitctl0/1, at CSR 0x7D4 / 0x7D7)

Field	Bits	Description	Access	Reset
Reserved	31:3	Reserved	R	0
pause_en	2	Enable/disable incrementing timer counter while executing PAUSE: 0: Disable incrementing (default) 1: Enable incrementing **Note:** If ‘1’ and the core is pausing (see Section 5.5.2), an internal timer interrupt terminates PAUSE and regular execution is resumed.	R/W	0

Field	Bits	Description	Access	Reset
halt_en	1	Enable/disable incrementing timer counter while in Sleep (i.e., pmu/fw-halt) state: 0: Disable incrementing (default) 1: Enable incrementing **Note:** If ‘1’ and the core is in Sleep (i.e., pmu/fw-halt) state, an internal timer interrupt transitions the core back to the Active (i.e., Running) state and regular execution is resumed.	R/W	0
enable	0	Enable/disable incrementing timer counter: 0: Disable incrementing 1: Enable incrementing (default)	R/W	1

5 Power Management and Multi-Core Debug Control

This chapter specifies the power management and multi-core debug control functionality provided or supported by the SweRV EH1 core. Also documented in this chapter is how debug may interfere with core power management.

5.1 Features

SweRV EH1 supports and provides the following power management and multi-core debug control 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/from SoC Power Management Unit (PMU)
- Signal indicating that core is halted
- Halt/run control interface to/from SoC debug Multi-Processor Controller (MPC) to enable cross-triggering in multi-core chips
- Signals indicating that core is in Debug Mode and core hit a breakpoint
- PAUSE feature to help avoid firmware spinning

5.2 Core Control Interfaces

SweRV EH1 provides two control interfaces, one for power management and one for multi-core debug control, which enable the core to be controlled by other SoC blocks.

5.2.1 Power Management

The power management interface enables an SoC-based Power Management Unit (PMU) to:

- Halt (i.e., enter low-power sleep state) or restart (i.e., resume execution) the core, and
- get an indication when the core has gracefully entered the sleep state.

The power management interface signals are described in Table 5-3.

5.2.2 Multi-Core Debug Control

The multi-core debug control interface enables an SoC-based Multi-Processor Controller (MPC) to:

- Control the reset state of the core (i.e., either start executing or enter Debug Mode),
- halt (i.e., enter Debug Mode) or restart (i.e., resume execution) the core,
- get an indication when the core is in Debug Mode, and
- cross-trigger other cores when this core has entered Debug Mode due to a software or a hardware breakpoint.

The multi-core debug control interface signals are described in Table 5-4.

5.3 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 5-1 depicts and Table 5-2 describes the core activity states as well as the events to transition between them.

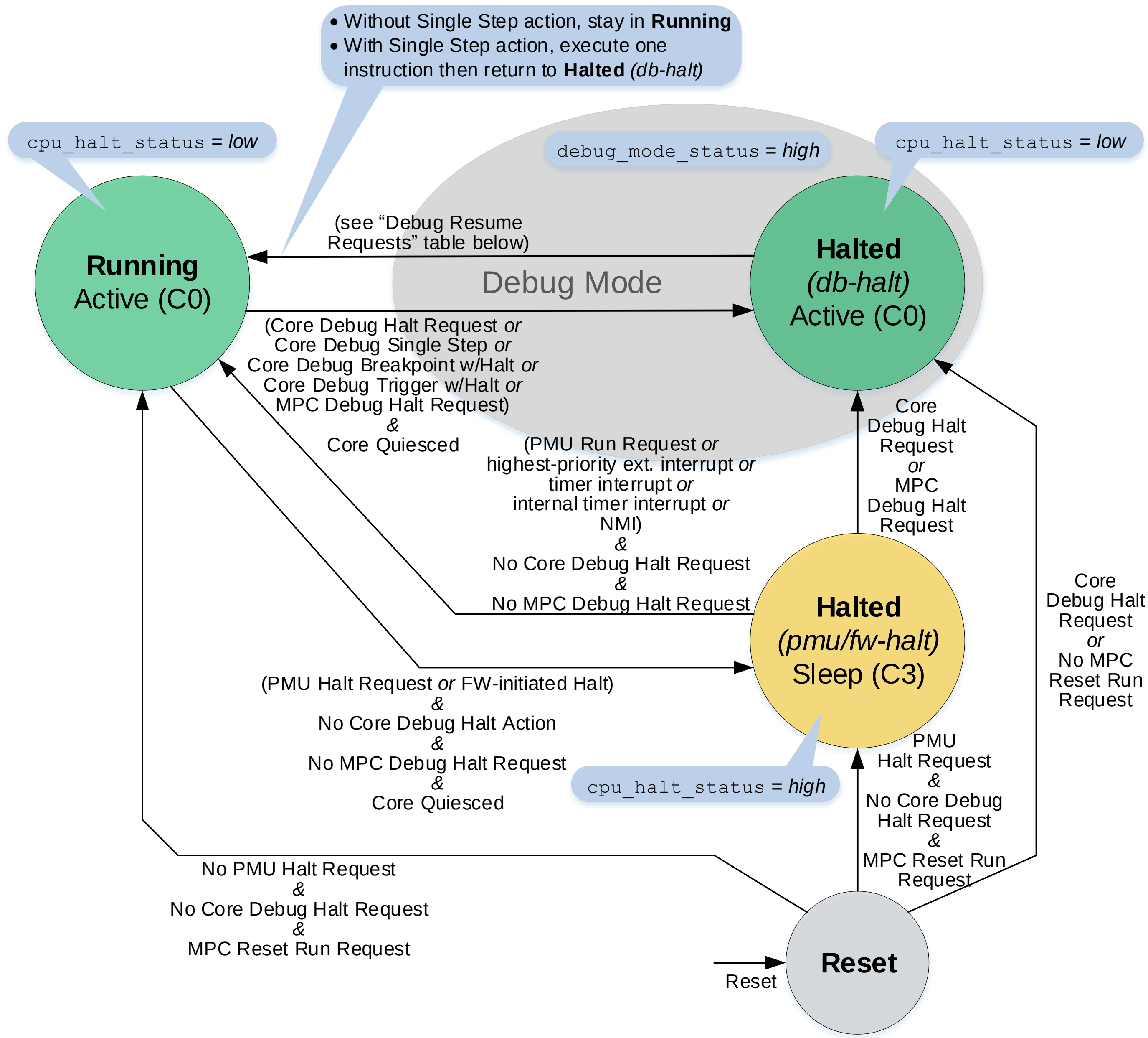


Figure 5-1 SweRV EH1 Core Activity States

Note: 'Core Quiesced' implies that no new instructions are executed and all outstanding core-initiated bus transactions are completed (i.e., the read buffer and the write buffer are empty, and all outstanding I-cache misses are finished). Note that the store queue and the DMA FIFO might not be empty due to on-going DMA transactions.

Table 5-1 Debug Resume Requests

Core-Internal State						Comments
Debug Resume	**Debug Halt**	**MPC Halt**	**MPC Run**	**Halted (This Cycle)**	**Halted (Next Cycle)**	
0	0	0	0	0	0	No request for Debug Mode entry
0	0	0	1			No action required from core (requires coordination outside of core)
0	0	1	0	1	1	Waiting for MPC Run (core remains in 'db-halt' state)
0	0	1	1	1	0	MPC Run Ack
0	1	0	0	1	1	Waiting for Debug Resume (core remains in 'db-halt' state)
0	1	0	1			No action required from core (requires coordination outside of core)
0	1	1	0	1	1	Waiting for both MPC Run and Debug Resume (core remains in 'db-halt' state)
0	1	1	1	1	1	Waiting for Debug Resume (core remains in 'db-halt' state)
1	0	0	0			No action required from core (requires coordination outside of core)
1	0	0	1			No action required from core (requires coordination outside of core)
1	0	1	0			No action required from core (requires coordination outside of core)
1	0	1	1			No action required from core (requires coordination outside of core)
1	1	0	0	1	0	Debug Resume Ack
1	1	0	1			No action required from core (requires coordination outside of core)
1	1	1	0	1	1	Waiting for MPC Run (core remains in 'db-halt' state)
1	1	1	1	1	0	Debug Resume Ack and MPC Run Ack

Note: While in 'db-halt' state, hardware ignores Debug Resume requests if the corresponding 'Debug Halt' state is not '1'. Likewise, hardware ignores MPC Debug Run requests if the corresponding 'MPC Halt' state is not '1'.

Note: The core-internal state bits are cleared upon exiting Debug Mode.

Note: In the time period between an MPC Debug Halt request and an MPC Debug Run request, a core debug single-step action is stalled but stays pending.

Note: Even if the core is already in Debug Mode due to a previous MPC Debug Halt request, a core debugger must initiate a debug halt (i.e., Core Debug Halt request) before it may start issuing other debug commands. However, if Debug Mode was entered due to a core debug breakpoint, a Core Debug Halt request is not required.

Note: An MPC Debug Halt request may only be signaled when the core is either not in Debug Mode or is already in Debug Mode due to a previous Core Debug Halt request or a debug breakpoint or trigger. Also, an MPC Debug Run request may only be signaled when the core is in Debug Mode due to either a previous MPC Debug Halt request, a previous Core Debug Halt request, or a debug breakpoint or trigger. Issuing more than one MPC Debug Halt requests in succession or more than one MPC Debug Run requests in succession is a protocol violation.

Table 5-2 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 Core 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*
Internal Timer Counters	`mitcnt0/1` incremented every core clock cycle (also during execution of instructions while single-stepping in Debug Mode)	`mitcnt0/1` not incremented	Depends on *halt_en* bit in `mitctl0/1` registers: 0: `mitcnt0/1` not incremented 1: `mitcnt0/1` incremented every core clock cycle
Machine Cycle Performance-Monitoring Counter	`mcycle` incremented every core clock cycle	Depends on *stopcount* bit of `dcsr` register (see Section 9.1.3.5): 0: `mcycle` incremented every core clock cycle 1: `mcycle` not incremented	`mcycle` not incremented

5.4 Power Control

The priority order of simultaneous halt requests is as follows:

1. Any core debug halt action:
    a. Core debug halt request
    b. Core debug single step
    c. Core debug breakpoint
    d. Core debug trigger

    or MPC debug halt request
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.

Important Note: There are two separate sources of debug operations: the core itself which conforms to the standard RISC-V Debug specification [3], and the Multi-Processor Controller (MPC) block which provides multi-core debug capabilities. These two sources may interfere with each other and need to be carefully coordinated on a higher level outside the core. Unintended behavior might occur if simultaneous debug operations from these two sources are not synchronized (e.g., MPC requesting a resume during the execution of an abstract command initiated by the debugger attached to the JTAG port).

5.4.1 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:

- Core debug halt request
- Core debug single step
- Core debug breakpoint with halt action
- Core debug trigger with halt action
- Multi-core debug halt request (from MPC)

Debug Mode is exited with:

- Core debug resume request with no single step action
- Multi-core debug run request (from MPC)

The state 'db-halt' is the only halt state allowed while in Debug Mode.

5.4.1.1 Single Stepping

A few notes about executing single-stepped instructions:

- Executing instructions which attempt to exit Debug Mode are ignored (e.g., writing to the `mpmc` register requesting to halt the core does not transition the core to the pmu/fw-halt state).
- Accesses to D-mode registers are illegal, even though the core is in Debug Mode.
- A core debug single-step action initiated in the time period between an MPC Debug Halt request and an MPC Debug Run request is stalled but stays pending until an MPC Debug Run request is issued.

5.4.2 Core Power and Multi-Core Debug Control and Status Signals

Figure 5-2 depicts the power and multi-core debug control and status signals which connect the SweRV EH1 core to the PMU and MPC blocks. Signals from the PMU and MPC to the core are asynchronous and must be synchronized to the core clock domain. Similarly, signals from the core are asynchronous to the PMU and MPC clock domains and must be synchronized to the PMU's or MPC's clock, respectively.

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.

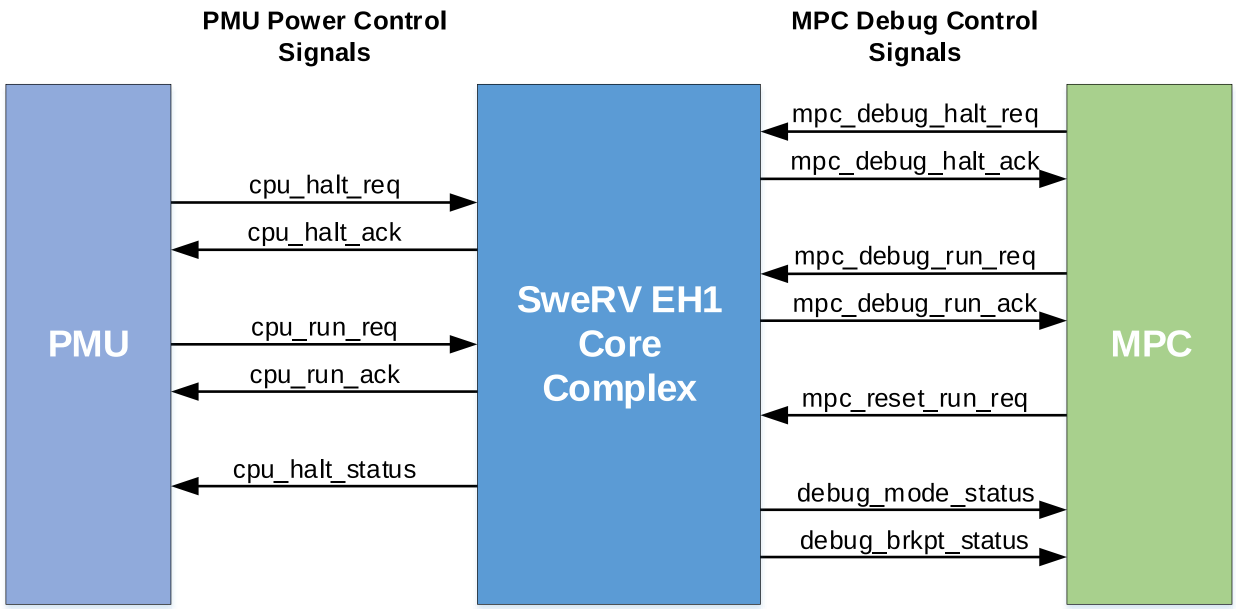


Figure 5-2 SweRV EH1 Power and Multi-Core Debug Control and Status Signals

5.4.2.1 Power Control and Status Signals

There are three types of signals between the Power Management Unit and the SweRV EH1 core, as described in Table 5-3. All signals are active-high.

Table 5-3 SweRV EH1 Power Control and Status Signals

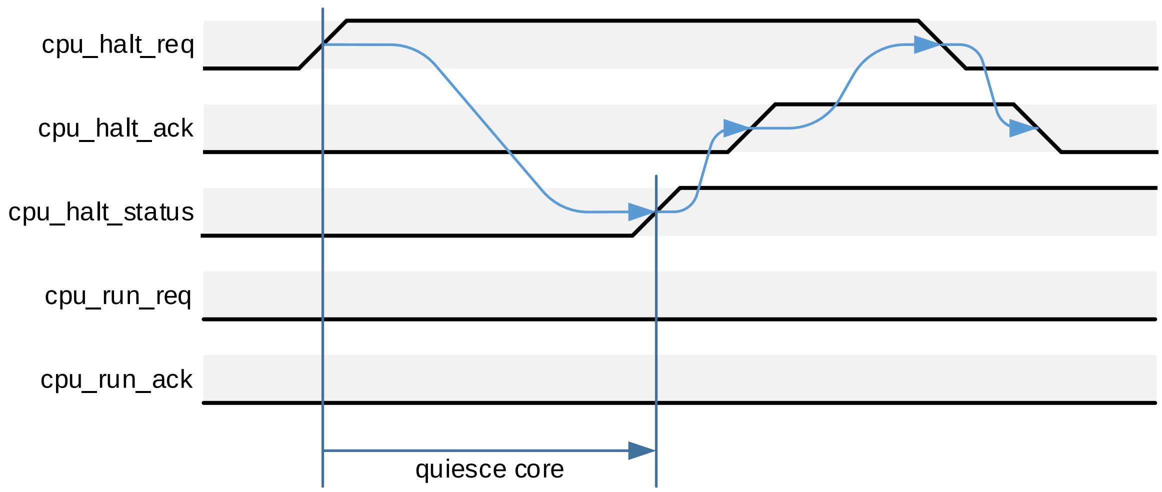
Signal(s)	Description
`cpu_halt_req` and `cpu_halt_ack`	Full handshake to request the core to halt. The PMU requests the core to halt (i.e., enter pmu/fw-halt) by asserting the `cpu_halt_req` signal. The core is quiesced before halting. The core then asserts the `cpu_halt_ack` signal. When the PMU detects 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. **Note:** `cpu_halt_req` must be tied to '0' if PMU interface is not used.
`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. **Note:** `cpu_run_req` must be tied to '0' if PMU interface is not used.
`cpu_halt_status`	Indication from the core to the PMU that the core has been gracefully halted.

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), an acknowledgement may not be signaled (i.e., the `cpu_halt_ack` or `cpu_run_ack` signal, respectively, may not be asserted). In general, requesting a state the core is already in should be avoided, and recovering from this condition needs to be handled at the SoC level.

Figure 5-3 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:

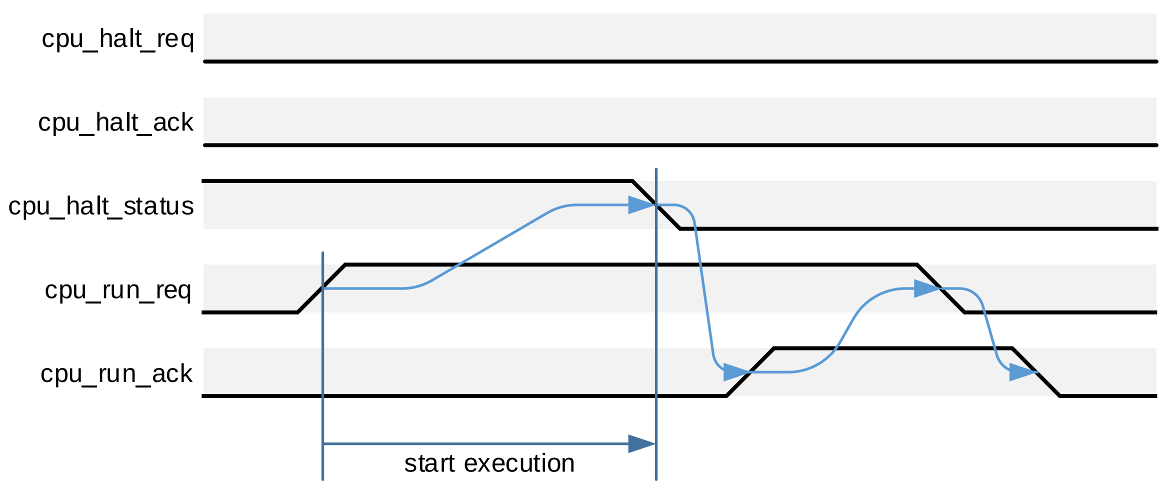


Figure 5-3 SweRV EH1 Power Control and Status Interface Timing Diagrams

5.4.2.2 Multi-Core Debug Control and Status Signals

There are five types of signals between the Multi-Processor Controller and the SweRV EH1 core, as described in Table 5-4. All signals are active-high.

Table 5-4 SweRV EH1 Multi-Core Debug Control and Status Signals

Signal(s)	Description
`mpc_debug_halt_req` and `mpc_debug_halt_ack`	Full handshake to request the core to debug halt. The MPC requests the core to halt (i.e., enter 'db-halt') by asserting the `mpc_debug_halt_req` signal. The core is quiesced before halting. The core then asserts the `mpc_debug_halt_ack` signal. When the MPC detects the asserted `mpc_debug_halt_ack` signal, it deasserts the `mpc_debug_halt_req` signal. Finally, when the core detects the deasserted `mpc_debug_halt_req` signal, it deasserts the `mpc_debug_halt_ack` signal. For as long as the `mpc_debug_halt_req` signal is asserted, the core must assert and hold the `mpc_debug_halt_ack` signal whether it was already in 'db-halt' or just transitioned into 'db-halt' state. **Note:** The *cause* field of the core's `dcsr` register (see Section 9.1.3.5) is set to 3 (i.e., the same value as a debugger-requested entry to Debug Mode due to a Core Debug Halt request). Similarly, the `dpc` register (see Section 9.1.3.6) is updated with the address of the next instruction to be executed at the time that Debug Mode was entered. **Note:** Signaling more than one MPC Debug Halt request in succession is a protocol violation. **Note:** `mpc_debug_halt_req` must be tied to '0' if MPC interface is not used.
`mpc_debug_run_req` and `mpc_debug_run_ack`	Full handshake to request the core to run. The MPC requests the core to run by asserting the `mpc_debug_run_req` signal. The core exits the halt state and starts execution again. The core then asserts the `mpc_debug_run_ack` signal. When the MPC detects the asserted `mpc_debug_run_ack` signal, it deasserts the `mpc_debug_run_req` signal. Finally, when the core detects the deasserted `mpc_debug_run_req` signal, it deasserts the `mpc_debug_run_ack` signal. For as long as the `mpc_debug_run_req` signal is asserted, the core must assert and hold the `mpc_debug_run_ack` signal whether it was already in 'Running' or after transitioning into 'Running' state. **Note:** The core remains in the 'db-halt' state if a core debug request is also still active. **Note:** Signaling more than one MPC Debug Run request in succession is a protocol violation**.** **Note:** `mpc_debug_run_req` must be tied to '0' if MPC interface is not used.
`mpc_reset_run_req`	Core start state control out of reset: 1: Normal Mode ('Running' or 'pmu/fw-halt' state) 0: Debug Mode halted ('db-halt' state) **Note:** The core complex does not implement a synchronizer for this signal because the timing of the first clock is critical. It must be synchronized to the core clock domain outside the core in the SoC. **Note:** `mpc_reset_run_req` must be tied to '1' if MPC interface is not used.
`debug_mode_status`	Indication from the core to the MPC that it is currently transitioning to or already in Debug Mode.
`debug_brkpt_status`	Indication from the core to the MPC that a software (i.e., `ebreak` instruction) or hardware (i.e., trigger hit) breakpoint has been triggered in the core. The breakpoint signal is only asserted for breakpoints and triggers with debug halt action. The signal is deasserted on exiting Debug Mode.

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

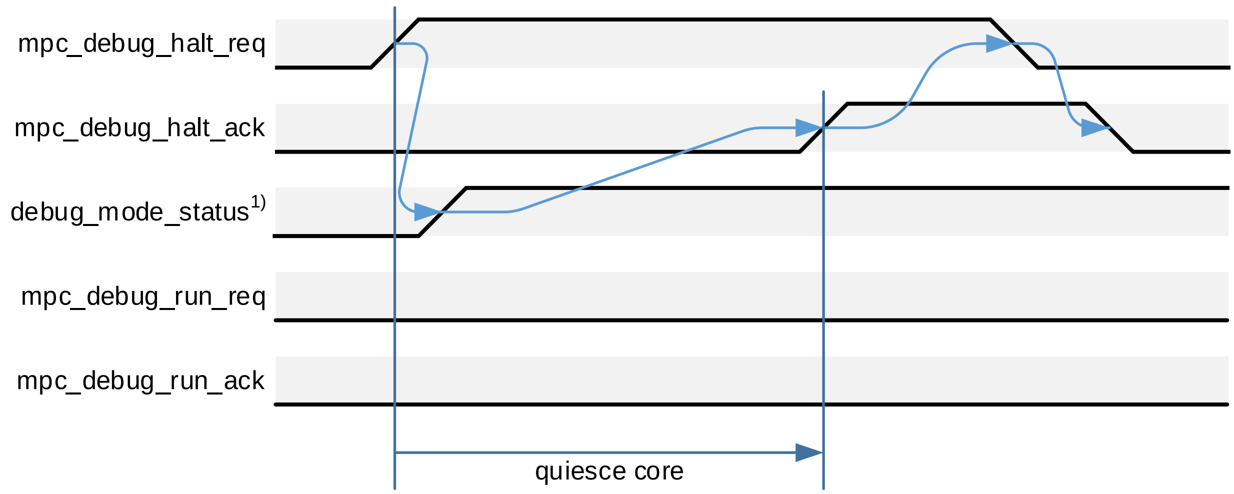
Note: If the core is either not in the db-halt state (i.e., `debug_mode_status` indication is not asserted) or is already in the db-halt state due to a previous Core Debug Halt request or a debug breakpoint or trigger (i.e., `debug_mode_status` indication is already asserted), asserting the `mpc_debug_halt_req` signal is allowed and acknowledged with the assertion of the `mpc_debug_halt_ack` signal. Also, asserting the `mpc_debug_run_req` signal is only allowed if the core is in the db-halt state (i.e., `debug_mode_status` indication is asserted), but the core asserts the `mpc_debug_run_ack` signal only after the `cpu_run_req` signal on the PMU interface has been asserted as well, if a PMU Halt request was still pending.

Note: If the MPC is requesting the core to enter Debug Mode out of reset by activating the `mpc_reset_run_req` signal, the `mpc_debug_run_req` signal may not be asserted until the core is out of reset and has entered Debug Mode. Violating this rule may lead to unexpected core behavior.

Note: If Debug Mode is entered at reset by setting the `mpc_reset_run_req` signal to ‘0’, only a run request issued on the `mpc_debug_run_req`/`ack` interface allows the core to exit Debug Mode. A core debug resume request issued by the debugger does not transition the core out of Debug Mode.

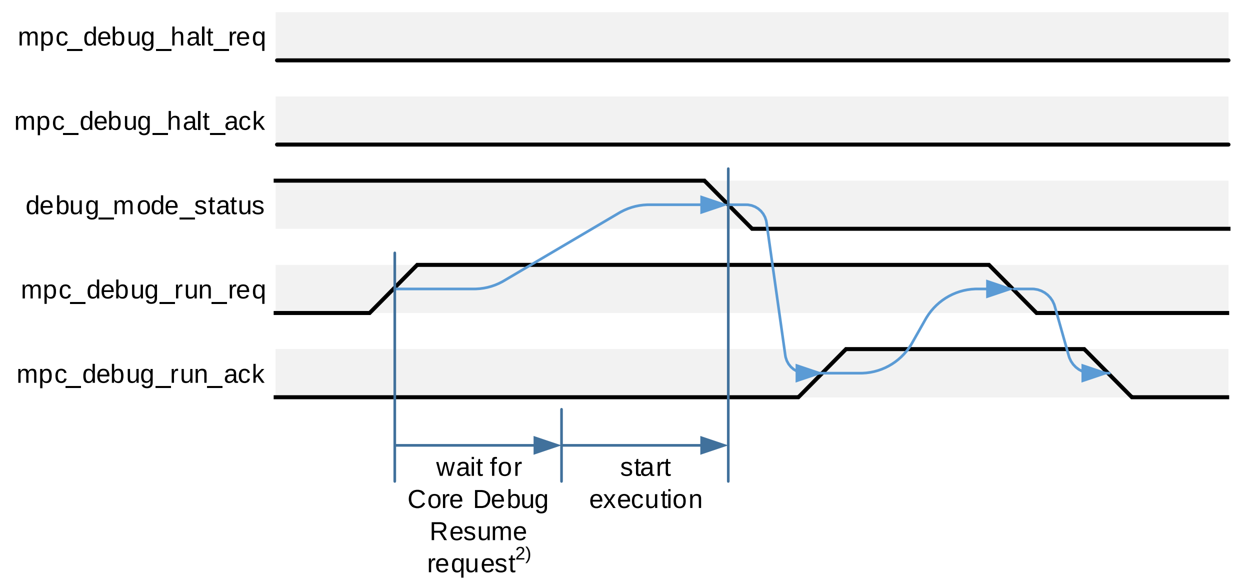
Figure 5-4 depicts conceptual timing diagrams of a halt and a run request.

MPC Halt Request:



1) if core not already quiesced and in Debug Mode due to earlier Core Debug Halt request (i.e., in active core debug session)

MPC Run Request:

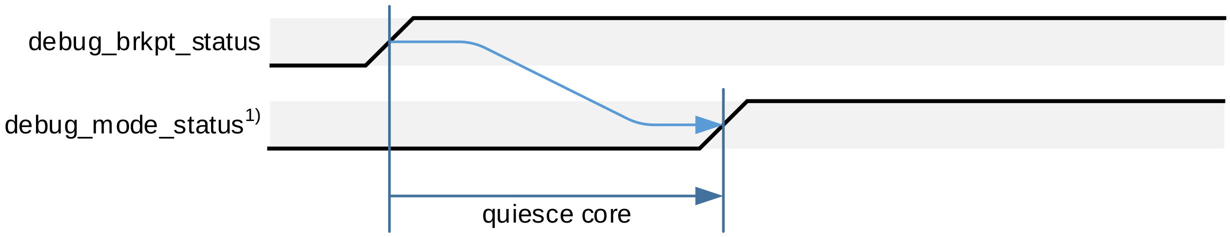


2) if in active core debug session

Figure 5-4 SweRV EH1 Multi-Core Debug Control and Status Interface Timing Diagrams

Figure 5-5 depicts conceptual timing diagrams of the breakpoint indication.

Breakpoint Signal Assertion:



1) if core not already quiesced and in Debug Mode due to earlier Core Debug Halt request (i.e., in active core debug session)

Breakpoint Signal Deassertion:

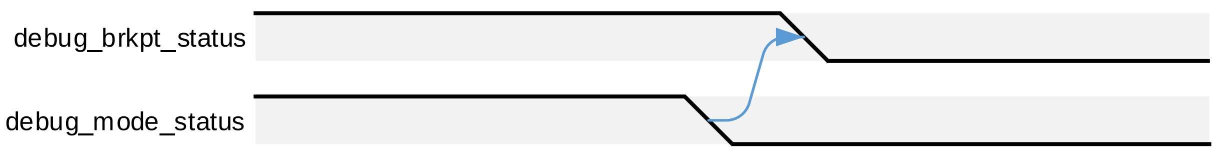


Figure 5-5 SweRV EH1 Breakpoint Indication Timing Diagrams

5.4.3 Debug Scenarios

The following mixed core debug and MPC debug scenarios are supported by the core:

5.4.3.1 Scenario 1: Core Halt → MPC Halt → MPC Run → Core Resume

1. Core debugger asserts a Debug Halt request which results in the core transitioning into Debug Halt state (db-halt).
2. In the system, another processor hits a breakpoint. The MPC signals a Debug Halt request to all processors to halt.
3. Core acknowledges this Debug Halt request as it is already in Debug Halt state (db-halt).
4. MPC signals a Debug Run request, but core is in the middle of a core debugger operation (e.g., an Abstract Command-based access) which requires it to remain in Debug Halt state.
5. Core completes debugger operation and waits for Core Debug Resume request from the core debugger.
6. When core debugger sends a Debug Resume request, the core then transitions to the Running state and deasserts the `debug_mode_status` signal.
7. Finally, core acknowledges MPC Debug Run request.

5.4.3.2 Scenario 2: Core Halt → MPC Halt → Core Resume → MPC Run

1. Core debugger asserts a Debug Halt request which results in the core transitioning into Debug Halt state (db-halt).
2. In the system, another processor hits a breakpoint. The MPC signals Debug Halt request to all processors to halt.
3. Core acknowledges this Debug Halt request as it is already in Debug Halt state (db-halt).
4. Core debugger completes its operations and sends a Debug Resume request to the core.
5. Core remains in Halted state as MPC has not yet asserted its Debug Run request. The `debug_mode_status` signal remains asserted.
6. When MPC signals a Debug Run request, the core then transitions to the Running state and deasserts the `debug_mode_status` signal.
7. Finally, core acknowledges MPC Debug Run request.

5.4.3.3 Scenario 3: MPC Halt → Core Halt → Core Resume → MPC Run

1. MPC asserts a Debug Halt request which results in the core transitioning into Debug Halt state (db-halt).
2. Core acknowledges this Debug Halt request.
3. Core debugger signals a Debug Halt request to the core. Core is already in Debug Halt state (db-halt).
4. Core debugger completes its operations and sends a Debug Resume request to the core.
8. Core remains in Halted state as MPC has not yet asserted its Debug Run request. The `debug_mode_status` signal remains asserted.
5. When MPC signals a Debug Run request, the core then transitions to the Running state and deasserts the `debug_mode_status` signal.
6. Finally, core acknowledges MPC Debug Run request.

5.4.3.4 Scenario 4: MPC Halt → Core Halt → MPC Run → Core Resume

1. MPC asserts a Debug Halt request which results in the core transitioning into Debug Halt state (db-halt).
2. Core acknowledges this Debug Halt request.
3. Core debugger signals a Debug Halt request to the core. Core is already in Debug Halt state (db-halt).
4. MPC signals a Debug Run request, but core debugger operations are still in progress. Core remains in Halted state. The `debug_mode_status` signal remains asserted.
5. Core debugger completes operations and signals a Debug Resume request to the core.
6. The core then transitions to the Running state and deasserts the `debug_mode_status` signal.
7. Finally, core acknowledges MPC Debug Run request.

5.4.3.5 Summary

For the core to exit out of Debug Halt state (db-halt) in cases where it has received debug halt requests from both core debugger and MPC, it must receive debug run requests from both the core debugger as well as the MPC, irrespective of the order in which debug halt requests came from both sources. Until then, the core remains halted and the `debug_mode_status` signal remains asserted.

5.4.4 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
- Highest-priority external interrupt (`mhwakeup` signal from PIC) and core interrupts are enabled
- Timer interrupt
- Internal 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 internal timers are part of the core and internally connected as well. 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.

5.4.5 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 5.5.1). The core is quiesced before indicating that it has gracefully halted.

5.4.6 DMA Operations While Halted

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

5.4.7 External Interrupts While Halted

All non-highest-priority external interrupts are temporarily ignored while halted. Only external interrupts which activate the `mhwakeup` signal (see Section 6.5.2, Steps 13 and 14) 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 sufficiently high priority to be signaled to the core are serviced once the core is back in the Running state.

5.5 Control/Status Registers

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

- Power Management Control Register (mpmc) (see Section 5.5.1)
- Core Pause Control Register (mcpc) (see Section 5.5.2)

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.

5.5.1 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. While entering the Halted state, interrupts may optionally be enabled atomically.

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

Note: Writing a '1' to the *haltie* field of the `mpmc` register without also setting the *halt* field has no immediate effect on the *mie* bit of the `mstatus` register. However, the *haltie* field of the `mpmc` register is updated accordingly.

Note: Once the *mie* bit of the `mstatus` register is set via the *haltie* field of the `mpmc` register, it remains set until other operations clear it. Exiting the Halted (pmu/fw-halt) state does not clear the *mie* bit of the `mstatus` register set by entering the Halted state.

Note: In Debug Mode, writing (i.e., setting or clearing) *haltie* has no effect on the `mstatus` register's *mie* bit since the core does not transition to the Halted (pmu/fw-halt) state.

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

Table 5-5 Power Management Control Register (mpmc, at CSR 0x7C6)

Field	Bits	Description	Access	Reset
Reserved	31:2	Reserved	R	0
haltie	1	Control interrupt enable (i.e., *mie* bit of `mstatus` register) when transitioning to Halted (pmu/fw-halt) state by setting *halt* bit below: 0: Don't change *mie* bit of `mstatus` register 1: Set *mie* bit of `mstatus` register (i.e., atomically enable interrupts)	R/W	1
halt	0	Initiate core halt (i.e., transition to Halted (pmu/fw-halt) state) **Note:** Write ignored if in Debug Mode	R0/W1	0

5.5.2 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[20] 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.

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

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: Depending on the *pause_en* bit of the `mitctl0/1` registers, the internal timers might be incremented while executing PAUSE. If an internal timer interrupt is signaled, PAUSE is terminated and normal execution resumes.

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 5-6 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

6 External Interrupts

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

6.1 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

6.2 Naming Convention

6.2.1 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.

6.2.2 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.

6.3 Overview of Major Functional Units

6.3.1 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.

6.3.2 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.

Implementation Note: The gateway does not implement any edge-detection logic (e.g., an edge-triggered flop) to convert the interrupt request to a level-triggered interrupt signal (see Figure 6-3). Therefore, the interrupt request input signal must be set to the inactive level (i.e., to ‘0’ for an active-high interrupt and to ‘1’ for an active-low interrupt) to avoid an interrupt request being continuously reported as pending, even after the gateway’s IP latch has been cleared. Consequently, if the gateway of an unused interrupt request input is programmed to an “active-high” polarity, the interrupt input signal must be tied off to ‘0’. Similarly, if the polarity is programmed to “active-low”, the interrupt input signal must be tied off to ‘1’.

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.

6.3.3 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.

6.3.4 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.

6.4 PIC Block Diagram

Figure 6-1 depicts a high-level view of the PIC. A simple gateway for asynchronous, level-triggered interrupt sources is shown in Figure 6-2, whereas Figure 6-3 depicts conceptually the internal functional blocks of a configurable gateway. Figure 6-4 shows a single comparator which is the building block to form the evaluation tree logic in the PIC core.

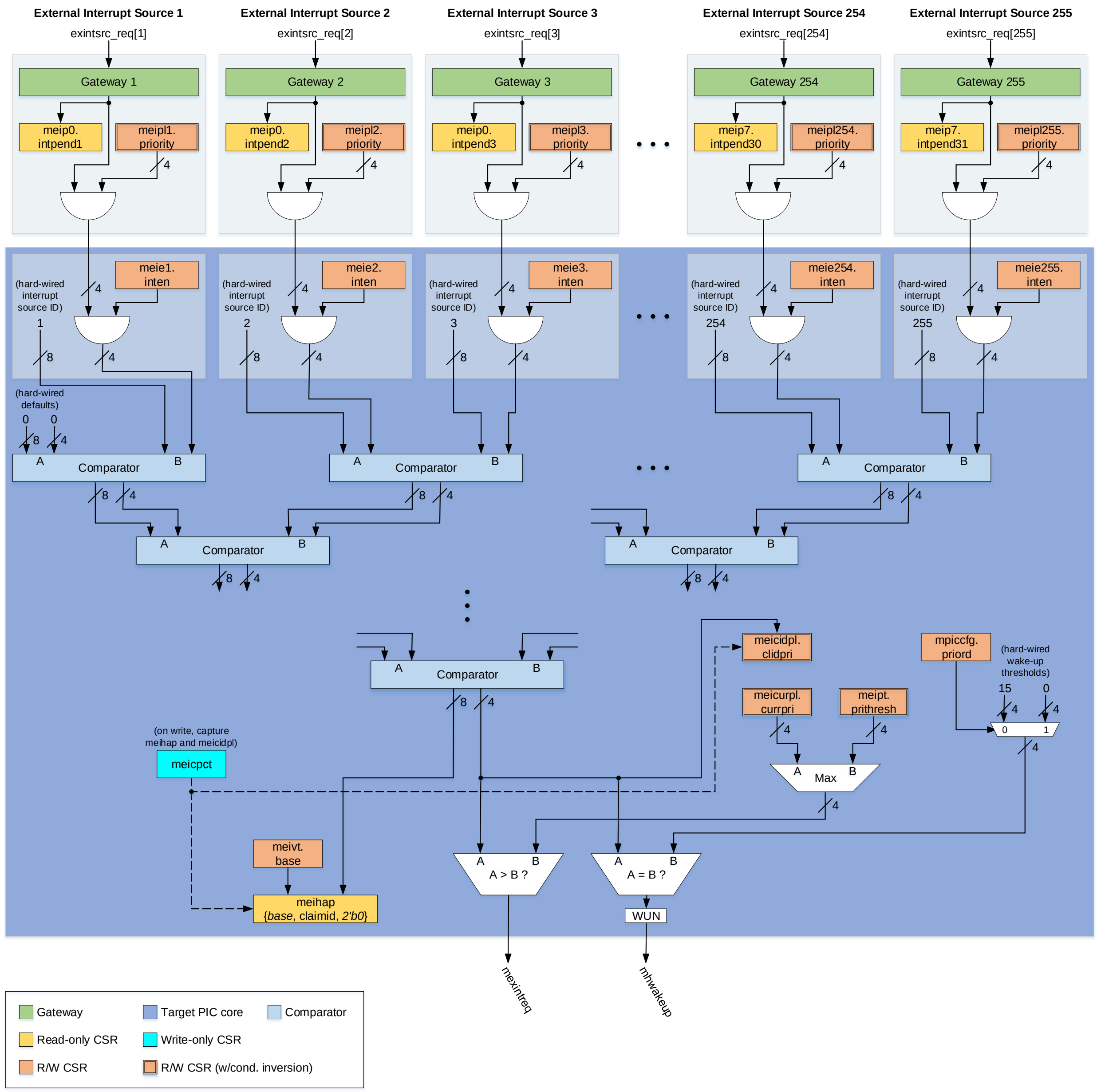


Figure 6-1 PIC Block Diagram

Implementation Note: For R/W control/status registers with double-borders in Figure 6-1, 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.

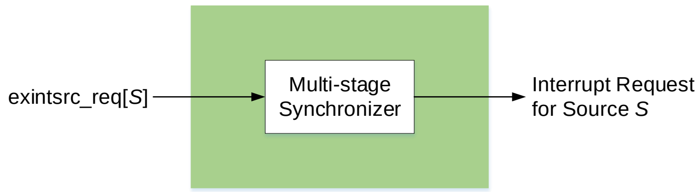


Figure 6-2 Gateway for Asynchronous, Level-triggered Interrupt Sources

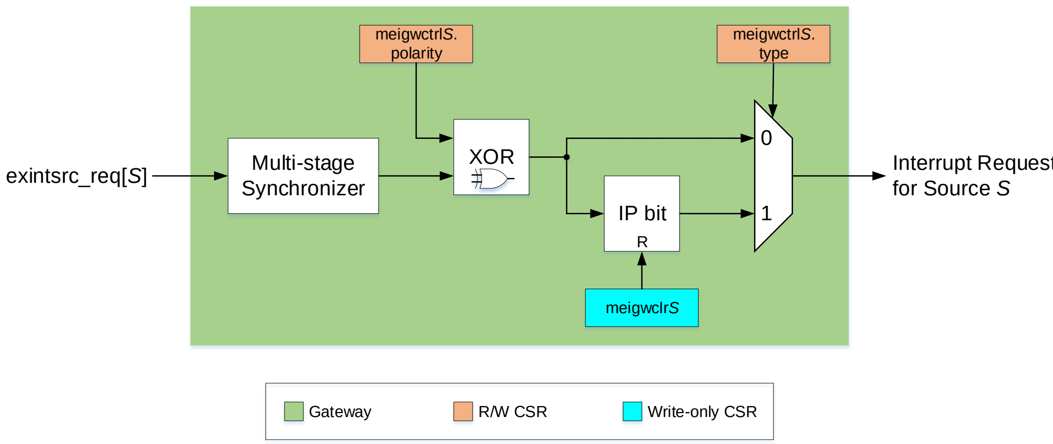


Figure 6-3 Conceptual Block Diagram of a Configurable Gateway

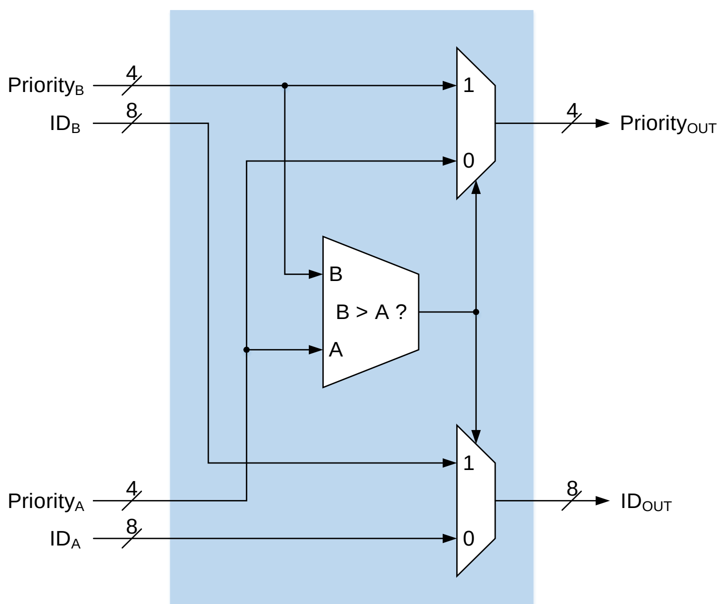


Figure 6-4 Comparator

6.5 Theory of Operation

Note: Interrupts must be disabled (i.e., the *mie* bit in the standard RISC-V `mstatus` register must be cleared) before changing the standard RISC-V `mtvec` register or the PIC's `meicurpl` and `meipt` registers, or unexpected behavior may occur.

6.5.1 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 `meigwctrlS` register and clear the IP bit by writing to the gateway's `meigwclrS` 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 `meiplS` 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 `meieS` registers for each interrupt source *S*.

6.5.2 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 `meipX` register.
6. With the pending interrupt, the source's interrupt priority (indicated by the *priority* field of the `meiplS` register) is forwarded to the evaluation logic.
7. If the corresponding interrupt enable (i.e., *inten* bit of the `meieS` 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 deasserted.
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[21].
15. When the target hart takes the external interrupt, it disables all interrupts (i.e., clears the *mie* bit of the RISC-V hart's `mstatus` register) and jumps to the external interrupt handler.
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 `meigwclrS` register.
19. The clearing deasserts 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 deasserted.
21. Firmware may update the content of the `meihap` and `meicidpl` registers by writing to the `meicpct` register to trigger a new capture.

6.6 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 independent 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 3.1.7 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.

[21] 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.

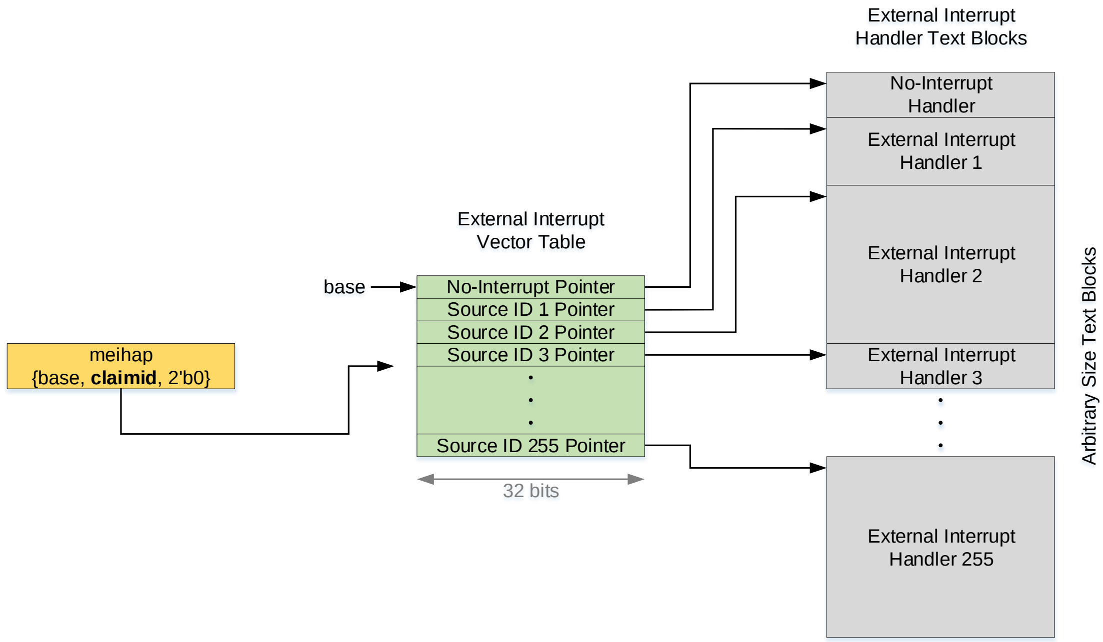


Figure 6-5 Vectored External Interrupts

Figure 6-5 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 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 be used.

Note: The interrupt source-specific handler must restore the register(s) saved in step 1. above before executing the `mret` instruction.

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 6-5 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.

6.7 Interrupt Chaining

Figure 6-6 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.

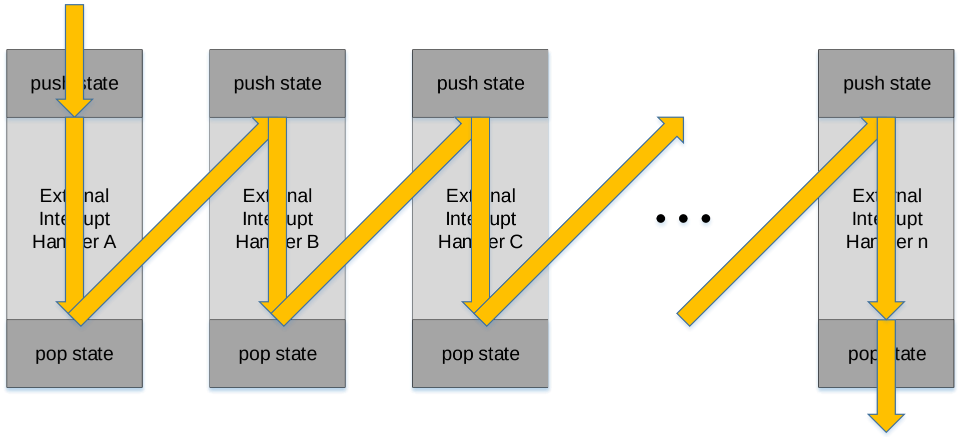


Figure 6-6 Concept of Interrupt Chaining

6.8 *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 6.5.2 *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 6.6:

```
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.

6.9 Performance Targets

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

6.10 Configurability

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

6.10.1 Rules

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

6.10.2 Build Arguments

The PIC build arguments are:

- **PIC base address for memory-mapped control/status registers (PIC_base_addr)**
  - See Section 16.2.2
- **Number of external interrupt sources**
  - Total interrupt sources (RV_PIC_TOTAL_INT): 2..255

6.10.3 Impact on Generated Code

6.10.3.1 External Interrupt Sources

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

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

6.10.3.2 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.

6.11 PIC Control/Status Registers

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

- External Interrupt Priority Threshold Register (meipt) (see Section 6.11.5)
- External Interrupt Vector Table Register (meivt) (see Section 6.11.6)
- External Interrupt Handler Address Pointer Register (meihap) (see Section 6.11.7)
- External Interrupt Claim ID / Priority Level Capture Trigger Register (meicpct) (see Section 6.11.8)
- External Interrupt Claim ID’s Priority Level Register (meicidpl) (see Section 6.11.9)

- External Interrupt Current Priority Level Register (meicurpl) (see Section 6.11.10)

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

- PIC Configuration Register (mpiccfg) (see Section 6.11.1)
- External Interrupt Priority Level Registers (meipl*S*) (see Section 6.11.2)
- External Interrupt Pending Registers (meip*X*) (see Section 6.11.3)
- External Interrupt Enable Registers (meie*S*) (see Section 6.11.4)
- External Interrupt Gateway Configuration Registers (meigwctrl*S*) (see Section 6.11.11)
- External Interrupt Gateway Clear Registers (meigwclr*S*) (see Section 6.11.12)

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.

6.11.1 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 6-1 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

6.11.2 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 `meiplS` 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 6-2 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

Field	Bits	Description	Access	Reset
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

6.11.3 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 6-3 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*X*32+Y*	*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

6.11.4 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 6-4 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 enable for interrupt source ID *S:* 0: Interrupt disabled 1: Interrupt enabled	R/W	0

6.11.5 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 6-5 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

6.11.6 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 6-6 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

6.11.7 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 6-7 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

6.11.8 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 6-8 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

6.11.9 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 6-9 External Interrupt Claim ID’s Priority Level Register (meicidpl, at CSR 0xBCB)

Field	Bits	Description	Access	Reset
Reserved	31:4	Reserved	R	0

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

6.11.10 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 6-10 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

6.11.11 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 6-11 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
polarity	0	External interrupt polarity for interrupt source ID *S:* 0: Active-high interrupt 1: Active-low interrupt	R/W	0

6.11.12 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 `meigwclrS` register.

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

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

Implementation Note: The `meigwclrS` 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 6-12 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

6.12 PIC CSR Address Map

Table 6-13 summarizes the PIC non-standard RISC-V CSR address map.

Table 6-13 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

6.13 PIC Memory-mapped Register Address Map

Table 6-14 summarizes the PIC memory-mapped register address map.

Table 6-14 PIC Memory-mapped Register Address Map

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

Address Offset from PIC_base_addr		Name	Description
Start	End		
+ 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

6.14 Interrupt Enable/Disable Code Samples

6.14.1 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
```

6.14.2 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, ((\type<<1) | \polarity)
    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
```

7 Performance Monitoring

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

7.1 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

7.2 Control/Status Registers

7.2.1 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 3.1.11 in [2])
- Machine Timer Registers (`mtime` and `mtimecmp`) (see Section 3.1.10 in [2])
  Note: `mtime` and `mtimecmp` are memory-mapped registers which must be provided by the SoC.

7.2.2 Platform-specific Control/Status Registers

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

- Group Performance Monitor Control Register (mgpmc) (see Section 7.2.2.1)

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.

7.2.2.1 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 7-1 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

7.3 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’.

7.4 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 of the `dcsr` register (see Section 9.1.3.5) 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.

7.5 Events

Table 7-2 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 7-2 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+32b) 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 and remainders committed (IP, 0/1)
11	loads committed	Number of loads committed (IP, 0/1)
12	stores committed	Number of stores committed (IP, 0/1)

Event No	Event Name	Description
13	misaligned loads	Number of misaligned loads (IP, 0/1)
14	misaligned stores	Number of misaligned stores (IP, 0/1)
15	alus committed	Number of ALU[22] 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[23] interrupts taken (IP)
39	external interrupts taken	Number of external interrupts taken (IP)

[22] NOP is an ALU operation. WFI is implemented as a NOP in SweRV EH1 and, hence, counted as an ALU operation was well.

[23] Events counted include interrupts triggered by the standard RISC-V platform-level timer as well as by the two internal timers.

Event No	Event Name	Description
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)
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 AXI4 or AHB-Lite I-bus busy (OOP)
48	cycles stalled due to D-bus busy	Number of cycles stalled due to AXI4 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)

8 Cache Control

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

8.1 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.

8.2 Feature Descriptions

8.2.1 Cache Flushing

As described in Section 2.8.2, 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).

8.2.2 Enabling/Disabling I-Cache

As described in Section 2.8.1, 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.

8.2.3 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 8.5.1 - 8.5.4 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.

8.3 Use Cases

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

1. **Debug Support**

   A few examples how diagnostic accesses (Section 8.2.3) 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 8.2.2) is an essential feature for this.

8.4 Theory of Operation

8.4.1 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), and
   - *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).

8.4.2 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), and
   - *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.

8.4.3 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 8.4.1 to read or Section 8.4.2 to write.
3. Increment the index.
4. Go back to step 2.) for a total of 16 iterations.

8.4.4 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), and
   - *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).

8.4.5 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), and
- *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.

8.5 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 8.5.1)
- I-Cache Array Data 0 Register (dicad0) (see Section 8.5.2)
- I-Cache Array Data 1 Register (dicad1) (see Section 8.5.3)
- I-Cache Array Go Register (dicago) (see Section 8.5.4)

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.

8.5.1 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 8.5.2 and 8.5.3) hold the information read from or to be written to the specified location, and the `dicago` register (see Section 8.5.4) 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 8-1 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[24]	15:2	Index address bits select **Notes:** • Index bits are right-justified; for I-cache sizes smaller than 256KB, unused upper bits are 0 • For tag array and status, bits 5..2 are ignored by hardware • This field does not have WARL behavior	R/W	0
Reserved	1:0	Reserved	R	0

8.5.2 I-Cache Array Data 0 Register (dicad0)

The `dicad0` register, in combination with the `dicad1` register (see Section 8.5.3), is used to store information read from or to be written to the I-cache array location specified with the `dicawics` register (see Section 8.5.1). Triggering a read or write access of the I-cache array is controlled by the `dicago` register (see Section 8.5.4). The layout of the `dicad0` register is different for the data array and the tag array / status, as described in Table 8-2 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 8-2 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 right-justified; for I-cache sizes larger than 16KB, unused higher bits are 0	R/W	0
Unused	11:7	Unused	R/W	0

[24] SweRV EH1'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

8.5.3 I-Cache Array Data 1 Register (dicad1)

The `dicad1` register, in combination with the `dicad0` register (see Section 8.5.38.5.2), is used to store information read from or to be written to the I-cache array location specified with the `dicawics` register (see Section 8.5.1). Triggering a read or write access of the I-cache array is controlled by the `dicago` register (see Section 8.5.4). The layout of the `dicad1` register is described in Table 8-3 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 8-3 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

8.5.4 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 8.5.1). Reading the `dicago` register populates the `dicad0`/`dicad1` registers (see Sections 8.5.2 and 8.5.3) 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 8-4 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 8-4 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

9 SweRV EH1 Debug Support

The SweRV EH1 core conforms to the "RISC-V Debug Specification 0.13.2, with JTAG DTM" document [3]. This chapter provides a description of the implemented debug-related control and status register definitions. For a RISC-V debug overview and detailed feature descriptions, refer to corresponding sections in [3].

9.1 Control/Status Registers

The RISC-V Debug architecture defines three separate address spaces: JTAG, Debug Module Interface, and RISC-V CSR. The registers associated with these three address spaces are described in the following sections:

- Control/Status Registers in JTAG Address Space (see Section 9.1.1)
- Control/Status Registers in Debug Module Interface Address Space (see Section 9.1.2)
- Control/Status Registers in RISC-V CSR Address Space (see Section 9.1.3)

9.1.1 Control/Status Registers in JTAG Address Space

Table 9-1 summarizes the control/status registers in the JTAG Debug Transport Module address space.

Addresses shown below are in the 5-bit JTAG address space. A control/status register is addressed by setting the 5-bit JTAG IR register.

Note: The core complex clock (`clk`) frequency must be at least twice the JTAG clock (`jtag_tck`) frequency for the JTAG data to pass correctly through the clock domain crossing synchronizers.

Table 9-1 Registers in JTAG Debug Transport Module Address Space

JTAG DTM Address	Name	Description	Section
0x01	IDCODE	JTAG IDCODE	9.1.1.1
0x10	dtmcs	DTM control and status	9.1.1.2
0x11	dmi	Debug module interface access	9.1.1.3
0x1F	BYPASS	JTAG BYPASS	9.1.1.4

9.1.1.1 IDCODE Register (IDCODE)

The `IDCODE` register is a standard JTAG register. It is selected in the JTAG TAP controller's IR register when the TAP state machine is reset. The `IDCODE` register's definition is exactly as defined in IEEE Std 1149.1-2013.

This register is read-only.

This register is mapped to the 5-bit JTAG address space.

Table 9-2 IDCODE Register (IDCODE, at JTAG 0x01)

Field	Bits	Description	Access	Reset
version	31:28	Identifies release version of this part	R	`jtag_id[31:28]` value (see Table 15-1)
partnum	27:12	Identifies designer's part number of this part	R	`jtag_id[27:12]` value (see Table 15-1)
manufid	11:1	Identifies designer/manufacturer of this part	R	`jtag_id[11:1]` value (see Table 15-1)
1	0	Must be '1'	R	1

9.1.1.2 DTM Control and Status Register (dtmcs)

The `dtmcs` register controls and provides status of the Debug Transport Module (DTM).

This register is mapped to the 5-bit JTAG address space.

Table 9-3 DTM Control and Status Register (dtmcs, at JTAG 0x10)

Field	Bits	Description	Access	Reset
Reserved	31:18	Reserved	R	0
dmihardreset	17	Not implemented **Note:** Hard reset of DTM not required in SweRV EH1 because DMI accesses always succeed. Writes to this bit ignored.	R	0
dmireset	16	Not implemented **Note:** Reset of DTM's error state not required in SweRV EH1 because DMI accesses always succeed. Writes to this bit ignored.	R	0
Reserved	15	Reserved	R	0
idle	14:12	Hint to debugger of minimum number of cycles debugger should spend in Run-Test/Idle after every DMI scan to avoid a 'busy' return code (*dmistat* of 3). Debugger must still check *dmistat* when necessary: 0: Not necessary to enter Run-Test/Idle at all. Other values not implemented.	R	0
dmistat	11:10	DMI status: 0: No error 1: Reserved 2..3: Not implemented (DMI accesses always succeed)	R	0
abits	9:4	Size of *address* field in `dmi` register (see Table 9-4)	R	7
version	3:0	Conforming to RISC-V Debug specification Version 0.13.2	R	1

9.1.1.3 Debug Module Interface Access Register (dmi)

The `dmi` register allows access to the Debug Module Interface (DMI).

In the JTAG TAP controller's Update-DR state, the DTM starts the operation specified in the *op* field.

In the JTAG TAP controller's Capture-DR state, the DTM updates the *data* field with the result from that operation.

Note: No status is reported in the *op* field. Therefore, debuggers should refrain from batching together multiple scans.

This register is mapped to the 5-bit JTAG address space.

Table 9-4 Debug Module Interface Access Register (dmi, at JTAG 0x11)

Field	Bits	Description	Access	Reset
address	40:34	Address used for DMI access. In Update-DR, value used to access DM over DMI.	R/W	0
data	33:2	Data to send to DM over DMI during Update-DR, and data returned from DM as result of previous operation.	R/W	0
op	1:0	For write: 0: Ignore data and address (nop) 1: Read from address (read) 2: Write data to address (write) 3: Not implemented (do not use) For read: 0: Previous operation completed successfully 1..3: Not implemented (DMI accesses always succeed)	R/W	0

9.1.1.4 BYPASS Register (BYPASS)

The `BYPASS` register is a standard JTAG register. It is implemented as a 1-bit register which has no functional effect, except adding a 1-bit delay. It allows a debugger to not communicate with this TAP (i.e., bypass it).

Note: All unused addresses in the 5-bit JTAG address space (i.e., all addresses except 0x01 (`IDCODE`), 0x10 (`dtmcs`), and 0x11 (`dmi`)) select the `BYPASS` register as well.

This register is mapped to the 5-bit JTAG address space.

Table 9-5 BYPASS Register (BYPASS, at JTAG 0x1F)

Field	Bits	Description	Access	Reset
bypass	0	Bypass	---	0

9.1.2 Control/Status Registers in Debug Module Interface Address Space

Table 9-6 summarizes the control/status registers in the Debug Module Interface address space.

Registers in the Debug Module Interface address space are accessed through the `dmi` register in the JTAG address space (see Section 9.1.1.3). The *address* field of the `dmi` register selects the Debug Module Interface register to be accessed, the *data* field either provides the value to be written to the selected register or captures that register's value, and the *op* field selects the operation to be performed.

Addresses shown below are offsets relative to the Debug Module base address. SweRV EH1 supports a single Debug Module with a base address of 0x00.

Table 9-6 Registers in Debug Module Interface Address Space

DMI Address	Name	Description	Section
0x04	data0	Abstract data 0	9.1.2.7
0x05	data1	Abstract data 1	
0x10	dmcontrol	Debug module control	9.1.2.1
0x11	dmstatus	Debug module status	9.1.2.2
0x16	abstractcs	Abstract control and status	9.1.2.4
0x17	command	Abstract command	9.1.2.5
0x18	abstractauto	Abstract command autoexec	9.1.2.6
0x38	sbcs	System bus access control and status	9.1.2.8
0x39	sbaddress0	System bus address 31:0	9.1.2.9
0x3C	sbdata0	System bus data 31:0	9.1.2.10
0x3D	sbdata1	System bus data 63:32	9.1.2.11
0x40	haltsum0	Halt summary 0	9.1.2.3

Note: ICCM, DCCM, and PIC memory ranges are only accessible using the access memory abstract command method. SoC memories are accessible using either the access memory abstract command method or the system bus access method.

Note: Abstract commands may only be executed when the core is in the debug halt (db-halt) state. However, SoC memory locations may be accessed using the system bus access method, irrespective of the core's state.

9.1.2.1 Debug Module Control Register (dmcontrol)

The `dmcontrol` register controls the overall Debug Module as well as the hart.

Note: On any given write, a debugger may only write '1' to either the *resumereq* or *ackhavereset* bit. The other bit must be written to '0'.

This register is mapped to the Debug Module Interface address space.

Table 9-7 Debug Module Control Register (dmcontrol, at Debug Module Offset 0x10)

Field	Bits	Description	Access	Reset
haltreq	31	Halt request: 0: Clears halt request bit **Note:** May cancel outstanding halt request. 1: Sets halt request bit **Note:** Running hart halts whenever halt request bit is set.	R0/W	0
resumereq	30	Resume request: 0: No effect 1: Causes hart to resume, if halted **Note:** Also clears resume ack bit for hart. **Note:** Setting *resumereq* bit is ignored if *haltreq* bit is set.	R0/W1	0
hartreset	29	Not implemented (i.e., 0: Deasserted)	R	0
ackhavereset	28	Reset core-internal, sticky `havereset` state: 0: No effect 1: Clear `havereset` state	R0/W1	0

Field	Bits	Description	Access	Reset
Reserved	27	Reserved	R	0
hasel	26	Selects definition of currently selected harts: 0: Single currently selected hart (SweRV EH1 is single-thread)	R	0
hartsello	25:16	Not implemented (SweRV EH1 is single-thread)	R	0
hartselhi	15:6	Not implemented (SweRV EH1 is single-thread)	R	0
Reserved	5:4	Reserved	R	0
setresethaltreq	3	Not implemented **Note:** *hasresethaltreq* bit in `dmstatus` register (Table 9-8) is '0'.	R	0
clrresethaltreq	2	Not implemented **Note:** *hasresethaltreq* bit in `dmstatus` register (Table 9-8) is '0'.	R	0
ndmreset	1	Controls reset signal from DM to SweRV EH1 core. Signal resets hart, but not DM. To perform a reset, debugger writes '1', and then writes '0' to deassert reset.	R/W	0
dmactive	0	Reset signal for Debug Module (DM): 0: Module's state takes its reset values **Note:** Only *dmactive* bit may be written to value other than its reset value. Writes to all other bits of this register are ignored. 1: Module functions normally Debugger may pulse this bit low to get Debug Module into known state. **Note:** The core complex's `dbg_rst_l` signal (see Table 15-1) resets the Debug Module. It should only be used to reset the Debug Module at power up or possibly with a global reset signal which resets the entire platform.	R/W	0

9.1.2.2 Debug Module Status Register (dmstatus)

The `dmstatus` register reports status for the overall Debug Module as well as the hart.

This register is read-only.

This register is mapped to the Debug Module Interface address space.

Table 9-8 Debug Module Status Register (dmstatus, at Debug Module Offset 0x11)

Field	Bits	Description	Access	Reset
Reserved	31:23	Reserved	R	0
impebreak	22	Not implemented **Note:** SweRV EH1 does not implement a Program Buffer.	R	0
Reserved	21:20	Reserved	R	0
allhavereset	19	'1' when hart has been reset and reset has not been acknowledged	R	--
anyhavereset	18	'1' when hart has been reset and reset has not been acknowledged	R	--
allresumeack	17	'1' when hart has acknowledged last resume request	R	--
anyresumeack	16	'1' when hart has acknowledged last resume request	R	--
allnonexistent	15	Not implemented (SweRV EH1 is single-thread)	R	0

Field	Bits	Description	Access	Reset
anynonexistent	14	Not implemented (SweRV EH1 is single-thread)	R	0
allunavail	13	'1' when hart is unavailable[25]	R	--
anyunavail	12	'1' when hart is unavailable[25]	R	--
allrunning	11	'1' when hart is running	R	--
anyrunning	10	'1' when hart is running	R	--
allhalted	9	'1' when hart is halted	R	--
anyhalted	8	'1' when hart is halted	R	--
authenticated	7	Not implemented (i.e., 1: Always authenticated)	R	1
authbusy	6	Not implemented (i.e., 0: Authentication module never busy)	R	0
hasresethaltreq	5	Not implemented **Note:** SweRV EH1 implements halt-on-reset with *haltreq* set out of reset method.	R	0
confstrptrvalid	4	Not implemented **Note:** SweRV EH1 does not provide information relevant to configuration string.	R	0
version	3:0	Debug Module present, conforming to RISC-V Debug specification Version 0.13.2	R	2

9.1.2.3 Halt Summary 0 Register (haltsum0)

Each bit in the `haltsum0` register indicates whether a specific hart is halted or not. Since SweRV EH1 is single-threaded, only one bit is implemented.

Note: Unavailable/nonexistent harts are not considered to be halted.

This register is read-only.

This register is mapped to the Debug Module Interface address space.

Table 9-9 Halt Summary 0 Register (haltsum0, at Debug Module Offset 0x40)

Field	Bits	Description	Access	Reset
Reserved	31:1	Reserved	R	0
halted	0	'1' when hart halted	R	0

9.1.2.4 Abstract Control and Status Register (abstractcs)

The `abstractcs` register provides status information of the abstract command interface and enables clearing of detected command errors.

Note: Writing this register while an abstract command is executing causes its *cmderr* field to be set to '1' (i.e., 'busy'), if it is '0'.

This register is mapped to the Debug Module Interface address space.

[25] Hart is in reset or *ndmreset* bit of `dmstatus` register is '1'.

Table 9-10 Abstract Control and Status Register (abstractcs, at Debug Module Offset 0x16)

Field	Bits	Description	Access	Reset
Reserved	31:29	Reserved	R	0
progbufsize	28:24	Not implemented **Note:** SweRV EH1 does not implement a Program Buffer.	R	0
Reserved	23:13	Reserved	R	0
busy	12	Abstract command interface activity: 0: Abstract command interface idle 1: Abstract command currently being executed **Note:** 'Busy' indication set when `command` register (see Section 9.1.2.5) is written, cleared after command has completed.	R	0
Reserved	11	Reserved	R	0
cmderr	10:8	Set if abstract command fails. Reason for failure: 0 (none): No error 1 (busy): Abstract command was executing when `command`, `abstractcs`, or `abstractauto` register was written, or when `data0` or `data1` register was read or written 2 (not supported): Requested command or option not supported, regardless of whether hart is running or not (i.e., illegal command, access register command not word-sized or *postexec* bit set, or access memory command size larger than word) 3 (exception): Exception occurred while executing abstract command (i.e., illegal register address, address outside of ICCM/DCCM/PIC memory range but in internal memory region, ICCM/DCCM uncorrectable ECC error, or ICCM/PIC access not word-sized) 4 (halt/resume): Abstract command couldn't execute because hart wasn't in required state (running/halted), or unavailable 5 (bus): Abstract command failed for SoC memory access due to bus error (e.g., unmapped address, uncorrectable error, incorrect alignment, or unsupported access size) 6: Reserved 7 (other): Register or memory access size not 32 bits wide or unaligned **Note:** Bits in this field remain set until cleared by writing '111'. **Note:** Next abstract command not started until value is reset to '0'. **Note:** Only contains valid value if *busy* is '0'.	R/W1C	0
Reserved	7:4	Reserved	R	0
datacount	3:0	2 data registers implemented as part of abstract command interface	R	2

9.1.2.5 Abstract Command Register (command)

Writes to the `command` register cause the corresponding abstract command to be executed.

Writing this register while an abstract command is executing causes the *cmderr* field in the `abstractcs` register (see Section 9.1.2.4) to be set to '1' (i.e., 'busy'), if it is '0'. If the *cmderr* field is non-zero, writes to the `command` register are ignored.

Note: A non-zero *cmderr* field inhibits starting a new abstract command to accommodate debuggers which, for performance reasons, may send several commands to be executed in a row without checking the *cmderr* field in between. Checking the *cmderr* field only at the end of a sequence of commands is safe because later commands which might depend on a previous, but failed command are not executed.

Note: Access register and access memory abstract commands may only be executed when the core is in the debug halt (db-halt) state. If the debugger is requesting the execution of an abstract command while the core is not in the debug halt state, the command is aborted and the *cmderr* field is set to '4' (i.e., 'halt/resume'), if it is '0'.

Note: The access memory abstract command method provides access to ICCM, DCCM, and PIC memory ranges as well as to SoC memories.

This register is mapped to the Debug Module Interface address space.

Table 9-11 Abstract Command Register (command, at Debug Module Offset 0x17)

Field	Bits	Description	Access	Reset
cmdtype	31:24	Abstract command type: 0: Access Register Command 2: Access Memory Command **Note:** Other values not implemented or reserved for future use. Writing this field to value different than '0' or '2' causes abstract command to fail and *cmderr* field of `abstractcs` register to be set to '2'.	R0/W	0
Access Register Command				
Reserved	23	Reserved	R	0
aarsize	22:20	Register access size: 2: 32-bit access **Note:** Other size values not implemented. Writing this field to value different than '2' causes abstract command to fail and *cmderr* field of `abstractcs` register to be set to '2', except if *transfer* is '0'.	R/W	2
aarpostincrement	19	Access register post-increment control: 0: No post-increment 1: After every successful access register command completion, increment *regno* field (wrapping around to 0)	R/W	0
postexec	18	Not implemented (i.e., 0: No effect) **Note:** Writing to '1' causes abstract command to fail and *cmderr* field of `abstractcs` register to be set to '2'.	R	0
transfer	17	Transfer: 0: Do not perform operation specified by *write* **Note:** Selection of unimplemented options (except for *aarsize* and *regno* fields) causes *cmderr* field of `abstractcs` register to be set to '2'. 1: Perform operation specified by *write* **Note:** Selection of unimplemented options causes abstract command to fail and *cmderr* field of `abstractcs` register to be set to '2'.	R	1
write	16	Read or write register: 0 (read): Copy data from register specified in *regno* field into `data0` register (Section 9.1.2.7) 1 (write): Copy data from `data0` register (Section 9.1.2.7) into register specified in *regno* field	R0/W	0

Field	Bits	Description	Access	Reset
regno	15:0	Register access: 0x0000 - 0x0FFF: CSRs 0x1000 - 0x101F: GPRs 0x1020 - 0xFFFF: Not implemented or reserved **Note:** Selecting illegal register address causes abstract command to fail and *cmderr* field of `abstractcs` register to be set to '3', except if *transfer* is '0'.	R0/W	0
Access Memory Command (ICCM, DCCM, PIC, and SoC Memories)				
aamvirtual	23	Not implemented (i.e., 0: Addresses are physical) **Note:** SweRV EH1 supports physical addresses only. Since physical and virtual address are identical, no error is flagged[26] even if written to '1'.	R	0
aamsize	22:20	Memory access size: 0: 8-bit access (for DCCM and SoC memories) 1: 16-bit access (for DCCM and SoC memories) 2: 32-bit access (for ICCM, DCCM, PIC, and SoC memories) **Note:** Writing this field to value '0' or '1' for ICCM or PIC memory access causes abstract command to fail and *cmderr* field of `abstractcs` register to be set to '3'. **Note:** Other size values not implemented. Writing this field to value higher than '2' causes abstract command to fail and *cmderr* field of `abstractcs` register to be set to '2'.	R/W	2
aampostincrement	19	Access memory post-increment control: 0: No post-increment 1: After every successful access memory command completion, increment `data1` register (which contains memory address, see Section 9.1.2.7) by number of bytes encoded in *aamsize* field	R/W	0
Reserved	18:17	Reserved	R	0
write	16	Read or write memory location: 0 (read): Copy data from memory location specified in `data1` register (i.e., address) into `data0` register (i.e., data) (Section 9.1.2.7) 1 (write): Copy data from `data0` register (i.e., data) into memory location specified in `data1` register (i.e., address) (Section 9.1.2.7)	R0/W	0
target-specific	15:14	Not implemented **Note:** SweRV EH1 does not use target-specific bits.	R	0
Reserved	13:0	Reserved	R	0

[26] The RISC-V Debug specification [3] states that an implementation must fail accesses that it does not support. However, the Debug Task Group community agreed in an email exchange on the group's reflector as well as in a group meeting that not reporting an error is acceptable for implementations without address translation (i.e., the physical address equals the virtual address).

9.1.2.6 Abstract Command Autoexec Register (abstractauto)

The `abstractauto` register controls if reading or writing the `data0/1` registers (see Section 9.1.2.7) automatically triggers the next execution of the abstract command in the `command` register (see Section 9.1.2.5). This feature allows more efficient burst accesses.

Writing this register while an abstract command is executing causes the *cmderr* field in the `abstractcs` register (see Section 9.1.2.4) to be set to '1' (i.e., 'busy'), if it is '0'.

This register is mapped to the Debug Module Interface address space.

Table 9-12 Abstract Command Autoexec Register (abstractauto, at Debug Module Offset 0x18)

Field	Bits	Description	Access	Reset
Reserved	31:2	Reserved	R	0
autoexecdata1	1	Auto-execution control for `data1` register: 0: No automatic triggering of abstract command execution 1: Reading or writing `data1` causes abstract command to be executed again	R/W	0
autoexecdata0	0	Auto-execution control for `data0` register: 0: No automatic triggering of abstract command execution 1: Reading or writing `data0` causes abstract command to be executed again	R/W	0

9.1.2.7 Abstract Data 0 / 1 Registers (data0/1)

The `data0/1` registers are basic read/write registers which may be read or changed by abstract commands.

Note: The *datacount* field of the `abstractcs` register (see Table 9-10) indicates that 2 (out of possible 12) registers are implemented in SweRV EH1.

The `data0` register sources the value for and provides the return value of an abstract command. The `data1` register provides the address for an access memory abstract command.

Note: Selecting an address outside of the ICCM, DCCM, or PIC memory range but in one of the core-internal memory regions causes the abstract command to fail and the *cmderr* field of the `abstractcs` register to be set to '3'. Similarly, selecting an unmapped SoC memory address causes the abstract command to fail, provided the SoC responds with a bus error, and the *cmderr* field of the `abstractcs` register to be set to '5'.

Accessing these registers while an abstract command is executing causes the *cmderr* field of the `abstractcs` register (see Table 9-10) to be set to '1' (i.e., 'busy'), if it was '0'.

Attempts to write the `data0/1` registers while the *busy* bit of the `abstractcs` register (see Table 9-10) is set does not change their value.

The values in these registers may not be preserved after an abstract command has been executed. The only guarantees on their contents are the ones offered by the executed abstract command. If the abstract command fails, no assumptions should be made about the contents of these registers.

These registers are mapped to the Debug Module Interface address space.

Table 9-13 Abstract Data 0 / 1 Register (data0/1, at Debug Module Offset 0x04 / 0x05)

Field	Bits	Description	Access	Reset
data	31:0	Abstract command data: data0: data value (access register and access memory command) data1: address (access memory command)	R/W	0

9.1.2.8 System Bus Access Control and Status Register (sbcs)

The `sbcs` register provides controls and status information of the system bus access interface.

Note: The system bus access method provides access to SoC memories only. Access to ICCM, DCCM, and PIC memory ranges is only available using the access memory abstract command method.

Note: The operation of the system bus access method does not depend on the core's state. SoC memory locations may be accessed using this method even when the core is running.

This register is mapped to the Debug Module Interface address space.

Table 9-14 System Bus Access Control and Status Register (sbcs, at Debug Module Offset 0x38)

Field	Bits	Description	Access	Reset
sbversion	31:29	System Bus interface conforms to RISC-V Debug specification, Version 0.13.2	R	1
Reserved	28:23	Reserved	R	0
sbbusyerror	22	Set when debugger attempts to read data while a read is in progress, or when debugger initiates a new access while one is still in progress (i.e., while *sbbusy* bit is set). Remains set until explicitly cleared by debugger. **Note:** When set, Debug Module cannot initiate more system bus accesses.	R/W1C	0
sbbusy	21	System bus master interface status: 0: System bus master idle 1: System bus master busy (Set when read or write access requested, remains set until access fully completed) **Note:** Writes to this register while *sbbusy* bit is set result in undefined behavior. Debugger must not write this register until it reads *sbbusy* bit as '0'. **Note:** Bit reflects if system bus master interface is busy, not status of system bus itself.	R	0
sbreadonaddr	20	Auto-read on address write: 0: No auto-read on address write 1: Every write to `sbaddress0` (see Section 9.1.2.9) automatically triggers system bus read at new address	R/W	0
sbaccess	19:17	Access size for system bus access: 0: 8-bit access 1: 16-bit access 2: 32-bit access 3: 64-bit access **Note:** Other values not supported. No access performed, *sberror* field set to '4'.	R/W	2
sbautoincrement	16	Auto-address increment: 0: No auto-address increment 1: `sbaddress0` register (see Section 9.1.2.9) incremented by access size (in bytes) selected in *sbaccess* field after every successful system bus access	R/W	0

Field	Bits	Description	Access	Reset
sbreadondata	15	Auto-read on data read: 0: No auto-read on data read 1: Every read from `sbdata0` register (see Section 9.1.2.10) automatically triggers new system bus read at (possibly auto-incremented) address	R/W	0
sberror	14:12	Set when Debug Module's system bus master encounters an error: While this field is non-zero, no more system bus accesses can be initiated by the Debug Module. 0: No bus error 1: Not implemented (no timeout) 2: Bad address accessed 3: Alignment error 4: Access of unsupported size requested 5..7: Not implemented (no other error conditions) **Note:** Bits in this field remain set until cleared by writing ‘111’. **Note:** Debug Module may not initiate next system bus access until value is reset to ‘0’.	R/W1C	0
sbasize	11:5	Width of system bus addresses (in bits)	R	32
sbaccess128	4	128-bit system bus accesses not supported	R	0
sbaccess64	3	64-bit system bus accesses supported	R	1
sbaccess32	2	32-bit system bus accesses supported	R	1
sbaccess16	1	16-bit system bus accesses supported	R	1
sbaccess8	0	8-bit system bus accesses supported	R	1

9.1.2.9 System Bus Address 31:0 Register (sbaddress0)

The `sbaddress0` register provides the address of the system bus access.

If the *sbreadonaddr* bit of the `sbcs` register is ‘1’, writing the `sbaddress0` register triggers a system bus read access from the new address.

Note: The *sberror* and *sbbusyerror* fields of the `sbcs` register must both be ‘0’ for a system bus read operation to be performed.

Note: If the system bus master interface is busy (i.e., *sbbusy* bit of the `sbcs` register is ‘1’) when a write access to this register is performed, the *sbbusyerror* bit in the `sbcs` register is set and the access is aborted.

This register is mapped to the Debug Module Interface address space.

Table 9-15 System Bus Address 31:0 Register (sbaddress0, at Debug Module Offset 0x39)

Field	Bits	Description	Access	Reset
address	31:0	System bus address	R/W	0

9.1.2.10 System Bus Data 31:0 Register (sbdata0)

The `sbdata0` register holds the right-justified lower bits for system bus read and write accesses.

A successful system bus read updates the `sbdata0/1` registers with the value read from the system bus at the memory location addressed by the `sbaddress0` register. If the width of the read access is less than 64 bits, the remaining high bits may take on any value.

Reading the `sbdata0` register provides the current value of this register. If the *sbreadondata* bit of the `sbcs` register is '1', reading this register also triggers a system bus read access which updates the `sbdata0/1` registers with the value read from the memory location addressed by the `sbaddress0` register.

Writing the `sbdata0` register triggers a system bus write access which updates the memory location addressed by the `sbaddress0` register with the new values in the `sbdata0/1` registers.

Note: Only the `sbdata0` register has this behavior. Accessing the `sbdata1` register has no side effects. A debugger must access the `sbdata1` register first, before accessing the `sbdata0` register.

Note: The *sberror* and *sbbusyerror* fields of the `sbcs` register must both be '0' for a system bus read or write operation to be performed.

Note: If the system bus master interface is busy (i.e., *sbbusy* bit of the `sbcs` register is '1') when a read or write access to this register is performed, the *sbbusyerror* bit in the `sbcs` register is set and the access is aborted.

This register is mapped to the Debug Module Interface address space.

Table 9-16 System Bus Data 31:0 Register (sbdata0, at Debug Module Offset 0x3C)

Field	Bits	Description	Access	Reset
data	31:0	System bus data[31:0] for system bus read and write accesses	R/W	0

9.1.2.11 System Bus Data 63:32 Register (sbdata1)

The `sbdata1` register holds the upper 32 bits of the 64-bit wide system bus for read and write accesses.

Note: If the system bus master interface is busy (i.e., *sbbusy* bit of the `sbcs` register is '1') when a read or write access to this register is performed, the *sbbusyerror* bit in the `sbcs` register is set and the access is aborted.

This register is mapped to the Debug Module Interface address space.

Table 9-17 System Bus Data 63:32 Register (sbdata1, at Debug Module Offset 0x3D)

Field	Bits	Description	Access	Reset
data	31:0	System bus data[63:32] for system bus read and write accesses	R/W	0

9.1.3 Control/Status Registers in RISC-V CSR Address Space

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

- Trigger Select Register (tselect) (see Section 9.1.3.1)
- Trigger Data 1 Register (tdata1) (see Section 9.1.3.2)
- Match Control Register (mcontrol) (see Section 9.1.3.3)
- Trigger Data 2 Register (tdata2) (see Section 9.1.3.4)
- Debug Control and Status Register (dcsr) (see Section 9.1.3.5)
- Debug PC Register (dpc) (see Section 9.1.3.6)

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.

9.1.3.1 Trigger Select Register (tselect)

Note: Since triggers can be used both by Debug Mode and M-mode, the debugger must restore this register if it modified it.

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

Table 9-18 Trigger Select Register (tselect, at CSR 0x7A0)

Field	Bits	Description	Access	Reset
Reserved	31:2	Reserved	R	0
index	1:0	Index of trigger 0..3 **Note:** Triggers 0 and 2 may be chained, triggers 1 and 3 not.	R/W	0

9.1.3.2 Trigger Data 1 Register (tdata1)

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

Table 9-19 Trigger Data 1 Register (tdata1, at CSR 0x7A1)

Field	Bits	Description	Access	Reset
type	31:28	See Table 9-20, "Match Control Register (mcontrol, at CSR 0x7A1)" below.	R	2
dmode	27			
data	26:0			

9.1.3.3 Match Control Register (mcontrol)

Note: SweRV EH1 does not support triggering on the data of a load or on the opcode of an executed instruction.

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

Table 9-20 Match Control Register (mcontrol, at CSR 0x7A1)

Field	Bits	Description	Access	Reset
type	31:28	Address/data match trigger (= mcontrol)	R	2
dmode	27	Mode write privileges to `tdata1/2` registers (Sections 9.1.3.2 and 9.1.3.4) selected by `tselect` register (Section 9.1.3.1): 0: Both Debug Mode and M-mode may write `tdata1/2` registers selected by `tselect` register 1: Only Debug Mode may write `tdata1/2` registers selected by `tselect` register. Writes from M-mode are ignored. **Note:** Only writable from Debug Mode.	R/W	0
maskmax	26:21	2^{31} bytes is largest naturally aligned powers-of-two (NAPOT) range supported by hardware when *match* field is '1'.	R	31
hit	20	Set by hardware when trigger matches. May be set or cleared by user at any time. Allows to determine which trigger(s) matched.	R/W	0
select	19	Match selection: 0: Perform match on address 1: Perform match on store data value	R/W	0
timing	18	Action for this trigger is taken just before instruction that triggered it is committed, but after all preceding instructions are committed. **Note:** No bus transaction is issued for an execute address trigger hit on a load to a side-effect address.	R	0
sizelo	17:16	Trigger must match exact address (if *select* bit is '0') or data (if *select* bit is '1').	R	0

Field	Bits	Description	Access	Reset
action	15:12	Action to take when trigger fires: 0: Raise breakpoint exception (used when software wants to use trigger module without external debugger attached) 1: Enter Debug Mode (only supported when trigger's *dmode* bit is ‘1’) **Note:** Other values reserved for future use. **Note:** Triggers do not fire if this field is ‘0’ and interrupts are disabled[27] (i.e., *mie* bit of `mstatus` standard RISC-V register is ‘0’).	R/W	0
chain	11	Trigger chaining: 0: Trigger match is independent of other triggers 1: Trigger match is conditional on next sequential trigger matching as well **Note:** Supported for triggers 0 and 2 only, attempts to set this bit for triggers 1 and 3 are ignored. **Note:** In SweRV EH1, only pairs of triggers (i.e., triggers 0/1 and triggers 2/3) are chainable. **Note:** If *chain* bit of trigger 0/2 is ‘1’, it is chained to trigger 1/3. Only *action* field of trigger 1/3 is used (i.e., *action* field of trigger 0/2 is ignored). The action on second trigger is taken if and only if both triggers in chain match at the same time. **Note:** Because the *chain* bit affects the next trigger, hardware must zero it for `mcontrol` register writes with *dmode* bit of ‘0’ if the next trigger has a *dmode* bit of ‘1’. In addition, hardware should ignore writes to the `mcontrol` register which set the *dmode* bit to ‘1’ if the previous trigger has both a *dmode* bit of ‘0’ and a *chain* bit of ‘1’. Debuggers must avoid the latter case by checking the *chain* bit of the previous trigger when writing the `mcontrol` register.	R/W	0
match	10:7	Match size: 0: Matches when value equals `tdata2` register’s (Section 9.1.3.4) value[28] 1: Matches when top *M* bits of value match top *M* bits of `tdata2` register’s (Section 9.1.3.4) value (*M* is 31 minus the index of least-significant bit containing 0 in `tdata2` register) **Note:** Other values not implemented or reserved for future use.	R/W	0
m	6	When set, enable this trigger in M-mode	R/W	0
Reserved	5	Reserved	R	0
s	4	Not implemented (SweRV EH1 is M-mode only)	R	0
u	3	Not implemented (SweRV EH1 is M-mode only)	R	0

[27] To enable native debugging of M-mode code, SweRV EH1 implements the simpler but more restrictive solution of preventing triggers with the *action* field set to ‘0’ (i.e., breakpoint exception) while interrupts are disabled, as described in Section 5.1, ‘Native M-Mode Triggers’ of the RISC-V Debug specification [3].

[28] Bit 0 of `tdata2` register is ignored for instruction address matches.

Field	Bits	Description	Access	Reset
execute	2	When set, trigger fires on address of executed instruction **Note:** For writes, written to '0' if *select* bit is written to '1'.	R/W	0
store	1	When set, trigger fires on address or data of store	R/W	0
load	0	When set, trigger fires on address of load **Note:** For writes, written to '0' if *select* bit is written to '1'.	R/W	0

9.1.3.4 Trigger Data 2 Register (tdata2)

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

Table 9-21 Trigger Data 2 Register (tdata2, at CSR 0x7A2)

Field	Bits	Description	Access	Reset

value	31:0	Match value:
		• Address or data value for match:
		  • Address of load, store, or executed instruction[28]
		  • Data value of store
		• Match mask
		(see *match* field of `mcontrol` register (Table 9-20) set to '1')
			R/W	0

9.1.3.5 Debug Control and Status Register (dcsr)

The `dcsr` register controls the behavior and provides status of the hart in Debug Mode.

The RISC-V Debug specification [3], Section 4.8.1 documents some required and several optional features. Table 9-22 describes the required features, the partial support of optional features in SweRV EH1, and indicates features not supported with "Not implemented".

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 standard read/write CSR address space.

Table 9-22 Debug Control and Status Register (dcsr, at CSR 0x7B0)

Field	Bits	Description	Access	Reset
xdebugver	31:28	External debug support exists as described in this chapter and [3]	R	4
Reserved	27:16	Reserved	R	0
ebreakm	15	0: `ebreak` in M-mode behaves as described in RISC-V Privileged specification [2] 1: `ebreak` in M-mode enters Debug Mode	R/W	0
Reserved	14	Reserved	R	0
ebreaks	13	Not implemented (SweRV EH1 is M-mode only)	R	0
ebreaku	12	Not implemented (SweRV EH1 is M-mode only)	R	0
stepie	11	0: Interrupts disabled during single stepping 1: Interrupts enabled during single stepping **Note:** Debugger must not change value while hart is running.	R/W	0

Field	Bits	Description	Access	Reset
stopcount	10	0: Increment counters as usual 1: Don't increment any counters (incl. `cycle` and `instret`) while in Debug Mode or on `ebreak` entering Debug Mode (referred value for most debugging scenarios)	R/W	0
stoptime	9	Increment timers same as in non-debug mode	R	0
cause	8:6	Reason for Debug Mode entry (if multiple reasons in single cycle, set cause to highest priority): 1: `ebreak` instruction was executed *(priority 3)* 2: Trigger Module caused a breakpoint exception *(priority 4, highest)* 3: Debugger or MPC interface (see Table 5-4) requested entry to Debug Mode using haltreq *(priority 1)* 4: Hart single-stepped because *step* was set *(priority 0, lowest)* 5: Hart halted directly out of reset due to resethaltreq (also acceptable to report '3') *(priority 2)* Other values reserved for future use.	R	0
Reserved	5	Reserved	R	0
mprven	4	Not implemented (i.e., 0: *mprv* field in `mstatus` register ignored in Debug Mode)	R	0
nmip	3	Non-Maskable Interrupt (NMI) pending for hart when set **Note:** NMI may indicate a hardware error condition, reliable debugging may no longer be possible once bit is set.	R	0
step	2	When set and not in Debug Mode, hart only executes single instruction and enters Debug Mode. If instruction does not complete due to exception, hart immediately enters Debug Mode before executing trap handler, with appropriate exception registers set. **Note:** Debugger must not change value while hart is running.	R/W	0
prv	1:0	Indicates privilege level hart was operating in when Debug Mode was entered (3 = M-mode)	R	3

9.1.3.6 Debug PC Register (dpc)

The `dpc` register provides the debugger information about the program counter (PC) when entering Debug Mode and control where to resume (RISC-V Debug specification [3], Section 4.8.2).

Upon entry to Debug Mode, the `dpc` register is updated with the address of the next instruction to be executed. The behavior is described in more detail in Table 9-23 below.

When resuming, the hart's PC is updated to the address stored in the `dpc` register. A debugger may write the `dpc` register to change where the hart resumes.

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 standard read/write CSR address space.

Table 9-23 Debug PC Register (dpc, at CSR 0x7B1)

Field	Bits	Description	Access	Reset
dpc	31:0	Address captured for: ebreak: Address of `ebreak` instruction Single step: Address of instruction which would be executed next if not in Debug Mode (i.e., PC + 4 for 32-bit instructions which don't change program flow, destination PC on taken jumps/branches, etc.) Trigger module: If timing (see *timing* bit in `mcontrol` register in Table 9-20) is: 0: Address of instruction which caused trigger to fire 1: Address of next instruction to be executed when Debug Mode was entered Halt request: Address of next instruction to be executed when Debug Mode was entered	R/W	0

10 Low-Level Core Control

This chapter describes some low-level core control registers.

10.1 Control/Status Registers

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

- Feature Disable Control Register (mfdc) (see Section 10.1.1)
- Clock Gating Control Register (mcgc) (see Section 10.1.2)

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.

10.1.1 Feature Disable Control 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 10-1 Feature Disable Control Register (mfdc, at CSR 0x7F9)

Field	Bits	Description	Access	Reset
Reserved	31:19	Reserved	R	0
dqc	18:16	DMA QoS control (see Section 2.13.3)	R/W	7
Reserved	15:11	Reserved	R	0
did	10	Dual issue disable: 0: dual issue 1: single issue	R/W	0
Reserved	9	Reserved	R	0
cecd	8	Core ECC check disable: 0: ICCM/DCCM ECC checking enabled 1: ICCM/DCCM ECC checking disabled	R/W	0
sad	7	Secondary ALU disable: 0: enable secondary ALU 1: disable secondary ALU	R/W	0
sespd	6	Side effect store pipelining disable: 0: side effect stores are pipelined 1: side effect stores block all subsequent bus transactions until store response with default value received **Note:** Side effect loads always block and freeze pipeline **Note:** Reset value depends on selected bus core build argument	R/W	0 (AHB-Lite) 1 (AXI4)
ldnbd	5	LSU/DIV non-blocking disable: 0: enable non-blocking loads/divides 1: disable non-blocking loads/divides	R/W	0

Field	Bits	Description	Access	Reset
fdd	4	Fast divide disable: 0: enable fast divide 1: disable fast divide	R/W	0
bpd	3	Branch prediction disable: 0: enable branch prediction and return address stack 1: disable branch prediction and return address stack	R/W	0
wbcd	2	Write Buffer (WB) coalescing disable: 0: enable Write Buffer coalescing 1: disable Write Buffer coalescing	R/W	0
Reserved	1	Reserved	R	0
pd	0	Pipelining disable: 0: pipelined execution 1: single instruction execution	R/W	0

10.1.2 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 10-2 Clock Gating Control Register (mcgc, at CSR 0x7F8)

Field	Bits	Description	Access	Reset
Reserved	31:9	Reserved	R	0
misc	8	Miscellaneous clock gating override: 0: enable clock gating 1: clock gating override	R/W	0
dec	7	DEC clock gating override: 0: enable clock gating 1: clock gating override	R/W	0
exu	6	EXU clock gating override: 0: enable clock gating 1: clock gating override	R/W	0
ifu	5	IFU clock gating override: 0: enable clock gating 1: clock gating override	R/W	0
lsu	4	LSU clock gating override: 0: enable clock gating 1: clock gating override	R/W	0

Field	Bits	Description	Access	Reset
bus	3	Bus clock gating override: 0: enable clock gating 1: clock gating override	R/W	0
pic	2	PIC clock gating override: 0: enable clock gating 1: clock gating override	R/W	0
dccm	1	DCCM clock gating override: 0: enable clock gating 1: clock gating override	R/W	0
iccm	0	ICCM clock gating override: 0: enable clock gating 1: clock gating override	R/W	0

11 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 11.1.1)
- Machine Cause Register (mcause) (see Section 11.1.2)

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.

11.1.1 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 11-1 and Table 11-2 below).

Table 11-1 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
mitie0	29	Internal timer 0 local interrupt enable	R/W	0
mitie1	28	Internal timer 1 local interrupt enable	R/W	0
Reserved	27:12	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[29]	R/W	0
Reserved	2:0	Reserved	R	0

Table 11-2 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
mitip0	29	Internal timer 0 local interrupt pending	R	0
mitip1	28	Internal timer 1 local interrupt pending	R	0
Reserved	27:12	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

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

Field	Bits	Description	Access	Reset
Reserved	6:0	Reserved	R	0

11.1.2 Machine Cause Register (mcause)

The standard RISC-V `mcause` register indicates the cause for a trap as shown in Table 11-3, 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 11-3 Machine Cause Register (mcause, at CSR 0x342)

Type	Trap Code	Value mcause[31:0]	Description	Section(s)
NMI	N/A	0x0000_0000	NMI pin assertion	2.15
Exception	1	0x0000_0001	Instruction access fault	2.7.4, 2.7.6, and 3.4
	2	0x0000_0002	Illegal instruction	
	3	0x0000_0003	Breakpoint	
	4	0x0000_0004	Load address misaligned	2.7.5
	5	0x0000_0005	Load access fault	2.7.4, 2.7.6, and 3.4
	6	0x0000_0006	Store/AMO address misaligned	2.7.5
	7	0x0000_0007	Store/AMO access fault	2.7.4, 2.7.6, and 3.4
	11	0x0000_000B	Environment call from M-mode	
Interrupt	7	0x8000_0007	Machine timer[30] interrupt	
	11	0x8000_000B	Machine external interrupt	
	28	0x8000_001C	Machine internal timer 1 local interrupt	4.3
	29	0x8000_001D	Machine internal timer 0 local interrupt	4.3
	30	0x8000_001E	Machine correctable error local interrupt	2.7.2
NMI	N/A	0xF000_0000	Machine D-bus store error NMI	2.7.1 and 2.15
		0xF000_0001	Machine D-bus non-blocking load error NMI	2.7.1 and 2.15

Note: All other values are reserved.

[30] Core external timer

12 CSR Address Map

12.1 Standard RISC-V CSRs

Table 12-1 lists the SweRV EH1 core-specific standard RISC-V Machine Information CSRs.

Table 12-1 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_0005
0xF14	MRO	mhartid	Hardware thread ID	0x0000_0000

Table 12-2 lists the SweRV EH1 standard RISC-V CSR address map.

Table 12-2 SweRV EH1 Standard RISC-V CSR Address Map

Number	Privilege	Name	Description	Section
0x300	MRW	mstatus	Machine status	
0x304	MRW	mie	Machine interrupt enable	11.1.1
0x305	MRW	mtvec	Machine trap-handler base address	
0x323	MRW	mhpmevent3	Machine performance-monitoring event selector 3	7.2.1
0x324	MRW	mhpmevent4	Machine performance-monitoring event selector 4	
0x325	MRW	mhpmevent5	Machine performance-monitoring event selector 5	
0x326	MRW	mhpmevent6	Machine performance-monitoring event selector 6	
0x340	MRW	mscratch	Scratch register for machine trap handlers	
0x341	MRW	mepc	Machine exception program counter	
0x342	MRW	mcause	Machine trap cause	11.1.2
0x343	MRW	mtval	Machine bad address or instruction	
0x344	MRW	mip	Machine interrupt pending	11.1.1
0x7A0	MRW	tselect	Debug/Trace trigger register select	9.1.3.1
0x7A1	MRW	tdata1	First Debug/Trace trigger data	9.1.3.2
		mcontrol	Match control	9.1.3.3
0x7A2	MRW	tdata2	Second Debug/Trace trigger data	9.1.3.4
0x7B0	DRW	dcsr	Debug control and status register	9.1.3.5
0x7B1	DRW	dpc	Debug PC	9.1.3.6
0xB00	MRW	mcycle	Machine cycle counter	7.2.1
0xB02	MRW	minstret	Machine instructions-retired counter	7.2.1

Number	Privilege	Name	Description	Section
0xB03	MRW	mhpmcounter3	Machine performance-monitoring counter 3	7.2.1
0xB04	MRW	mhpmcounter4	Machine performance-monitoring counter 4	
0xB05	MRW	mhpmcounter5	Machine performance-monitoring counter 5	
0xB06	MRW	mhpmcounter6	Machine performance-monitoring counter 6	
0xB80	MRW	mcycleh	Upper 32 bits of mcycle, RV32I only	7.2.1
0xB82	MRW	minstreth	Upper 32 bits of minstret, RV32I only	7.2.1
0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter3, RV32I only	7.2.1
0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter4, RV32I only	
0xB85	MRW	mhpmcounter5h	Upper 32 bits of mhpmcounter5, RV32I only	
0xB86	MRW	mhpmcounter6h	Upper 32 bits of mhpmcounter6, RV32I only	

12.2 Non-Standard RISC-V CSRs

Table 12-3 summarizes the SweRV EH1 non-standard RISC-V CSR address map.

Table 12-3 SweRV EH1 Non-Standard RISC-V CSR Address Map

Number	Privilege	Name	Description	Section
0x7C0	MRW	mrac	Region access control	2.8.1
0x7C2	MRW	mcpc	Core pause control	5.5.2
0x7C4	DRW	dmst	Memory synchronization trigger (Debug Mode only)	2.8.2
0x7C6	MRW	mpmc	Power management control	5.5.1
0x7C8	DRW	dicawics	I-cache array/way/index selection (Debug Mode only)	8.5.1
0x7C9	DRW	dicad0	I-cache array data 0 (Debug Mode only)	8.5.2
0x7CA	DRW	dicad1	I-cache array data 1 (Debug Mode only)	8.5.3
0x7CB	DRW	dicago	I-cache array go (Debug Mode only)	8.5.4
0x7D0	MRW	mgpmc	Group performance monitor control	7.2.2.1
0x7D2	MRW	mitcnt0	Internal timer counter 0	4.4.1
0x7D3	MRW	mitb0	Internal timer bound 0	4.4.2
0x7D4	MRW	mitctl0	Internal timer control 0	4.4.3
0x7D5	MRW	mitcnt1	Internal timer counter 1	4.4.1
0x7D6	MRW	mitb1	Internal timer bound 1	4.4.2
0x7D7	MRW	mitctl1	Internal timer control 1	4.4.3
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	10.1.2

Number	Privilege	Name	Description	Section
0x7F9	MRW	mfdc	Feature disable control	10.1.1
0xBC0	MRW	mdeau	D-Bus error address unlock	2.8.4
0xBC8	MRW	meivt	External interrupt vector table	6.11.6
0xBC9	MRW	meipt	External interrupt priority threshold	6.11.5
0xBCA	MRW	meicpct	External interrupt claim ID / priority level capture trigger	6.11.8
0xBCB	MRW	meicidpl	External interrupt claim ID’s priority level	6.11.9
0xBCC	MRW	meicurpl	External interrupt current priority level	6.11.10
0xFC0	MRO	mdseac	D-bus first error address capture	2.8.3
0xFC8	MRO	meihap	External interrupt handler address pointer	6.11.7

13 Interrupt Priorities

Table 13-1 summarizes the SweRV EH1 platform-specific (Local) and standard RISC-V (External and Timer) relative interrupt priorities.

Table 13-1 SweRV EH1 Platform-specific and Standard RISC-V Interrupt Priorities

	Interrupt	**Section**
Highest Interrupt Priority	*Non-Maskable Interrupt (standard RISC-V)*	2.15
	External interrupt (standard RISC-V)	6
	Correctable error (local interrupt)	2.7.2
	Timer interrupt (standard RISC-V)	
	Internal timer 0 (local interrupt)	4.3
Lowest Interrupt Priority	Internal timer 1 (local interrupt)	4.3

14 Clock and Reset

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

14.1 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 complex clock input
  - System bus clock ratios controlled by enable signals
- Single core complex reset signal
  - Ability to reset to Debug Mode
- Separate Debug Module reset signal
  - Allows to interact with Debug Module when core complex is still in reset

14.2 Clocking

14.2.1 Regular Operation

The SweRV EH1 core complex is driven by a single clock (`clk`). All input and output signals, except those listed in Table 14-1, 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 14-1 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.

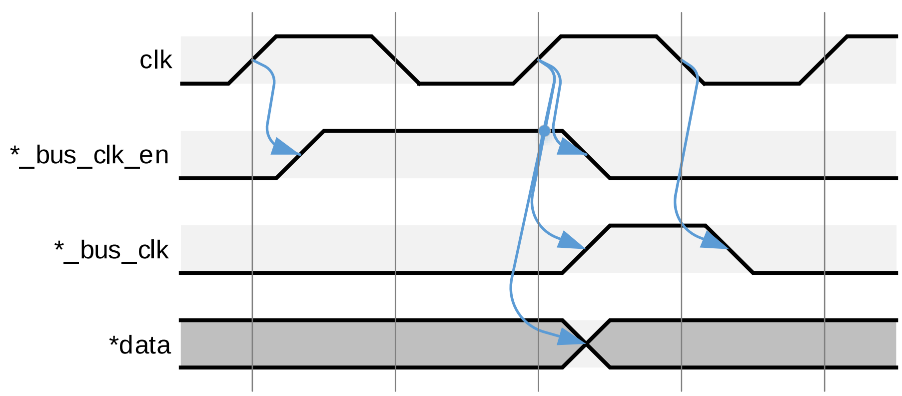


Figure 14-1 Conceptual Clock, Clock-Enable, and Data Timing Relationship

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.

14.2.2 System Bus-to-Core Clock Ratios

Figure 14-2 to Figure 14-9 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).

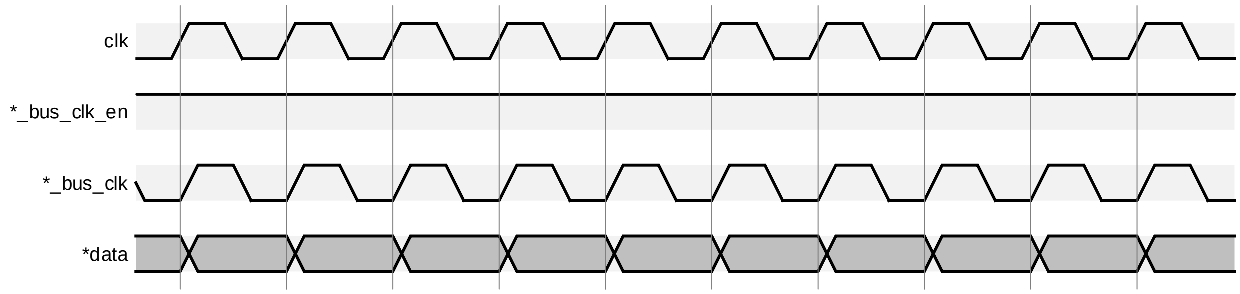


Figure 14-2 1:1 System Bus-to-Core Clock Ratio

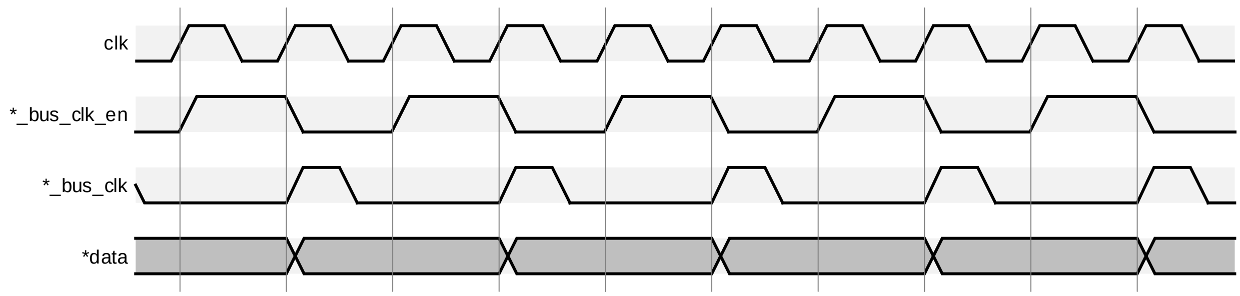


Figure 14-3 1:2 System Bus-to-Core Clock Ratio

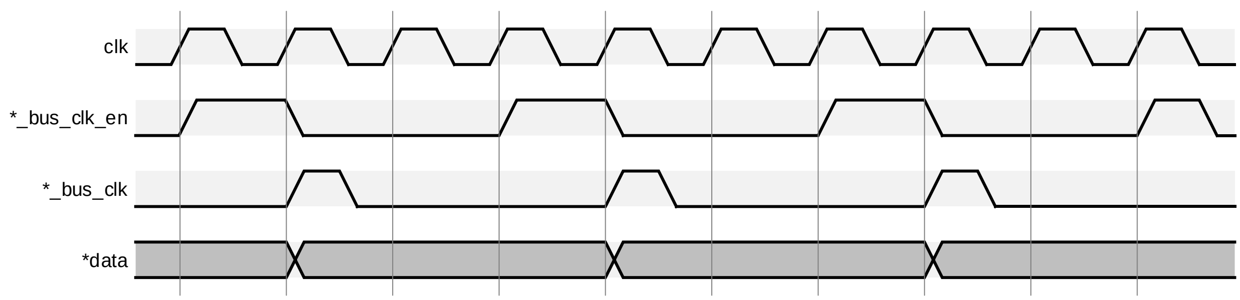


Figure 14-4 1:3 System Bus-to-Core Clock Ratio

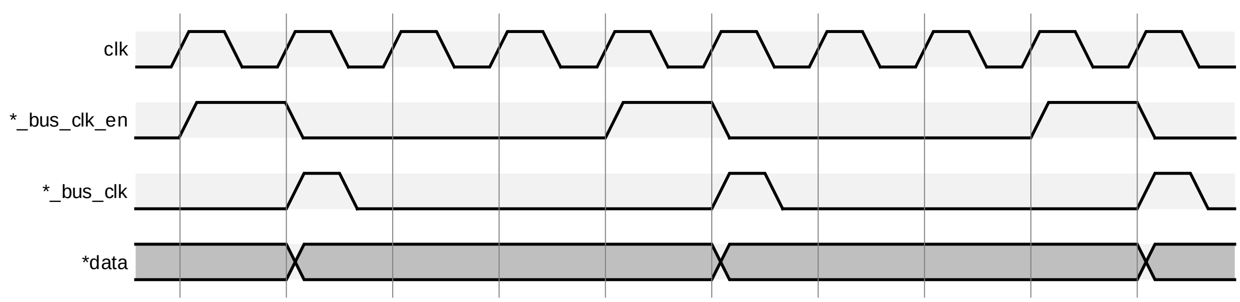


Figure 14-5 1:4 System Bus-to-Core Clock Ratio

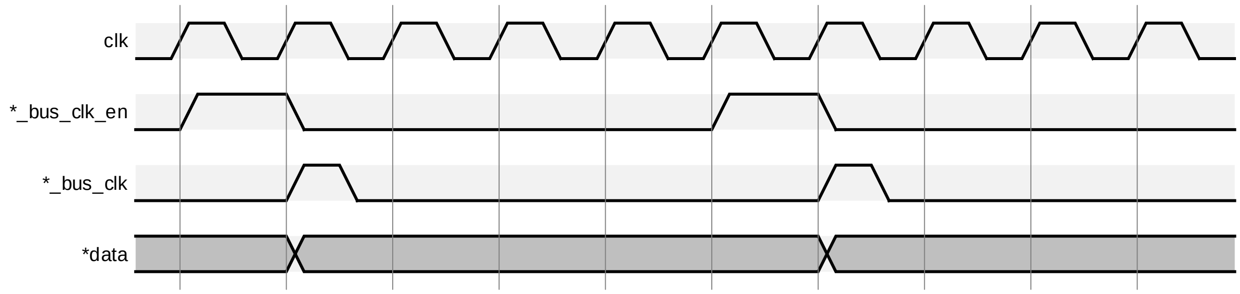


Figure 14-6 1:5 System Bus-to-Core Clock Ratio

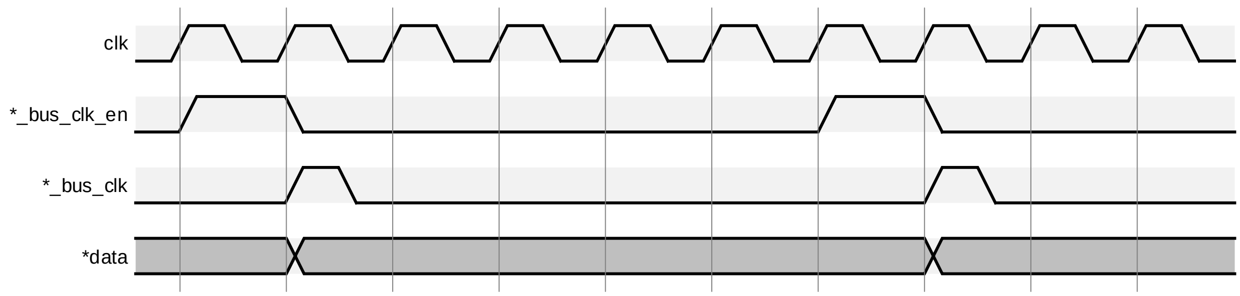


Figure 14-7 1:6 System Bus-to-Core Clock Ratio

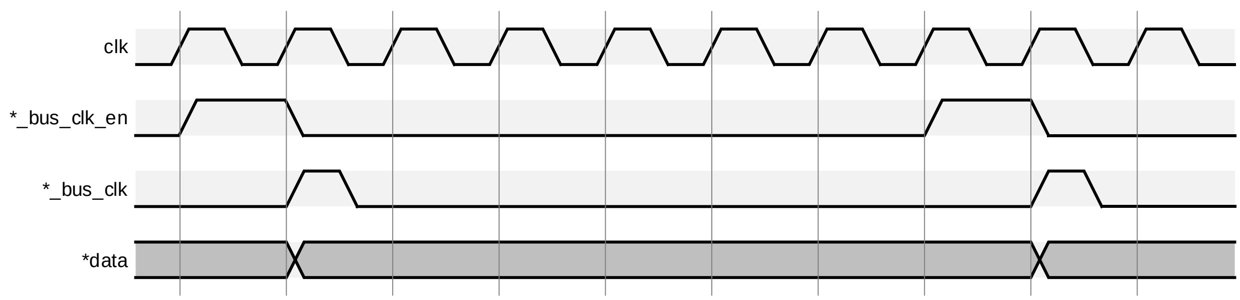


Figure 14-8 1:7 System Bus-to-Core Clock Ratio

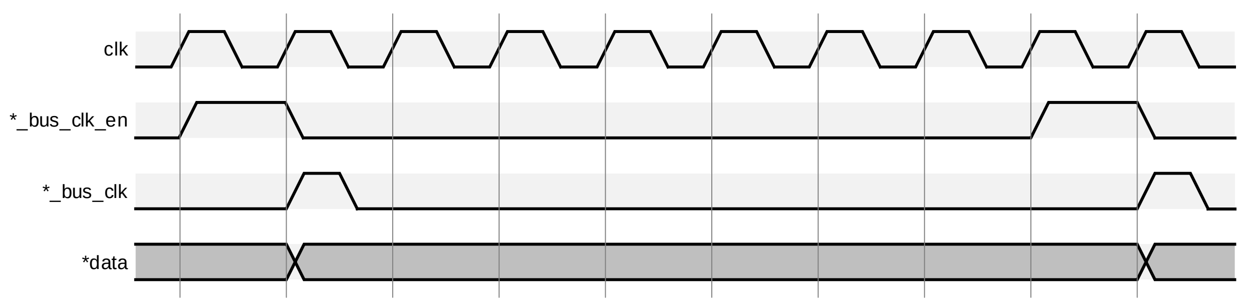


Figure 14-9 1:8 System Bus-to-Core Clock Ratio

14.2.3 Asynchronous Signals

Table 14-1 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 14-1 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 Unit (PMU) Interface		
i_cpu_halt_req	in	PMU halt request to core
i_cpu_run_req	in	PMU run request to core
Multi-Processor Controller (MPC) Debug Interface		
mpc_debug_halt_req	in	MPC debug halt request to core
mpc_debug_run_req	in	MPC debug run request to core
JTAG		
jtag_tck	in	JTAG Test Clock
jtag_tms	in	JTAG Test Mode Select (synchronous to jtag_tck)
jtag_tdi	in	JTAG Test Data In (synchronous to jtag_tck)
jtag_trst_n	in	JTAG Test Reset
jtag_tdo	out	JTAG Test Data Out (synchronous to jtag_tck)

14.3 Reset

The SweRV EH1 core complex provides two reset signals, the core complex reset (see Section 14.3.1) and the Debug Module reset (see Section 14.3.2).

14.3.1 Core Complex Reset (rst_l)

As shown in Figure 14-10, the core complex reset signal (`rst_l`) is active-low, may be asynchronously asserted, but must be synchronously deasserted 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.

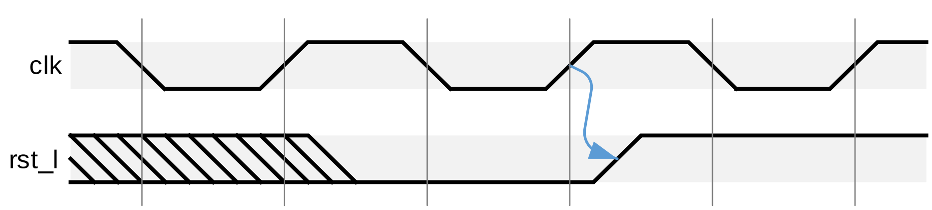


Figure 14-10 Conceptual Clock and Reset Timing Relationship

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.

Note: The core complex reset signal resets the entire SweRV EH1 core complex, except the Debug Module.

14.3.2 Debug Module Reset (dbg_rst_l)

The Debug Module reset signal (`dbg_rst_l`) is an active-low signal which resets the SweRV EH1 core complex's Debug Module as well as the synchronizers between the JTAG interface and the core complex. The Debug Module reset signal may be connected to the power-on reset signal of the SoC. This allows an external debugger to interact with the Debug Module when the core complex reset signal (`rst_l`) is still asserted.

If this layered reset functionality is not required, the `dbg_rst_l` signal may be tied to the `rst_l` signal outside the core complex.

14.3.3 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.

14.3.4 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 `dmcontrol` register (see Section 9.1.2.1) of 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 of `dmcontrol` register (`dmcontrol` = 0x0000_0001)
2. Reset core complex
    - Set *ndmreset* bit of `dmcontrol` register (`dmcontrol` = 0x0000_0003)
3. While in reset, assert halt request with `ndmreset` still asserted
    - Set *haltreq* bit of `dmcontrol` register (`dmcontrol` = 0x8000_0003)
4. Take core complex out of reset with halt request still asserted
    - Clear *ndmreset* bit of `dmcontrol` register (`dmcontrol` = 0x8000_0001)

15 SweRV EH1 Core Complex Port List

Table 15-1 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 15-1 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 (excl. Debug Module)
rst_vec[31:1]	in	Core reset vector
dbg_rst_l	in	Debug Module reset (incl. JTAG synchronizers)
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 AXI4[31]		
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

[31] 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_awprot[2:0]	out	Protection type
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

Signal	Dir	Description
Load/Store Unit Master AXI4		
Write address channel signals		
lsu_axi_awvalid	out	Write address valid
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)*

Signal	Dir	Description
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		
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 AXI4		
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

Signal	Dir	Description
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
sb_axi_arlen[7:0]	out	Burst length *(hardwired to 0b0000_0000)*
sb_axi_arsize[2:0]	out	Burst size
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 AXI4		
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

Signal	Dir	Description
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
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

Signal	Dir	Description
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
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		
Slave 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
dma_hsel	in	Slave select

Signal	Dir	Description
dma_hreadyin	in	Transfer finished in
Master signals		
dma_hrdata[63:0]	out	Read data
dma_hreadyout	out	Transfer finished
dma_hresp	out	Slave transfer response
Power Management Unit (PMU) Interface		
i_cpu_halt_req	in	PMU halt request to core (async)
o_cpu_halt_ack	out	Core acknowledgement for PMU halt request
o_cpu_halt_status	out	Core halted indication
i_cpu_run_req	in	PMU run request to core (async)
o_cpu_run_ack	out	Core acknowledgement for PMU run request
Multi-Processor Controller (MPC) Debug Interface		
mpc_debug_halt_req	in	MPC debug halt request to core (async)
mpc_debug_halt_ack	out	Core acknowledgement for MPC debug halt request
mpc_debug_run_req	in	MPC debug run request to core (async)
mpc_debug_run_ack	out	Core acknowledgement for MPC debug run request
mpc_reset_run_req	in	Core start state control out of reset
o_debug_mode_status	out	Core in Debug Mode indication
debug_brkpt_status	out	Hardware/software breakpoint indication
Performance Counter Activity		
dec_tlu_perfcnt0[1:0]	out	Performance counter 0 incrementing (pipeline I1, I0)
dec_tlu_perfcnt1[1:0]	out	Performance counter 1 incrementing (pipeline I1, I0)
dec_tlu_perfcnt2[1:0]	out	Performance counter 2 incrementing (pipeline I1, I0)
dec_tlu_perfcnt3[1:0]	out	Performance counter 3 incrementing (pipeline I1, I0)
Trace Port[32]		
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

[32] The core provides trace information for a maximum of two instructions and one interrupt/exception per clock cycle. Note that the only information provided for interrupts/exceptions is the cause, the interrupt/exception flag, and the trap value. The core's trace port busses are minimally sized, but wide enough to deliver all trace information the core may produce in one clock cycle. Not provided signals for the upper bits of the interface related to the interrupt slot might have to be tied off in the SoC.

Signal	Dir	Description
JTAG Port		
jtag_tck	in	JTAG Test Clock (async)
jtag_tms	in	JTAG Test Mode Select (async, sync to jtag_tck)
jtag_tdi	in	JTAG Test Data In (async, sync to jtag_tck)
jtag_trst_n	in	JTAG Test Reset (async)
jtag_tdo	out	JTAG Test Data Out (async, sync to jtag_tck)
jtag_id[31:1]	in	JTAG IDCODE register value (bit 0 tied internally to 1)
Testing		
scan_mode	in	May be used to enable logic scan test, if implemented (must be ‘0’ for normal core operation)
mbist_mode	in	May be used to enable MBIST for core-internal memories, if implemented (should be tied to ‘0’ if not used)

16 SweRV EH1 Core Build Arguments

16.1 Memory Protection Build Arguments

16.1.1 Memory Protection Build Argument Rules

The rules for valid memory protection address (INST/DATA_ACCESS_ADDR*x*) and mask (INST/DATA_ACCESS_MASK*x*) build arguments are:

- INST/DATA_ACCESS_ADDR*x* must be 64B-aligned (i.e., 6 least significant bits must be '0')
- INST/DATA_ACCESS_MASK*x* must be an integer multiple of 64B minus 1 (i.e., 6 least significant bits must be '1')
- For INST/DATA_ACCESS_MASK*x*, all '0' bits (if any) must be left-justified and all '1' bits must be right-justified
- No bit in INST/DATA_ACCESS_ADDR*x* may be '1' if the corresponding bit in INST/DATA_ACCESS_MASK*x* is also '1' (i.e., for each bit position, at most one of the bits in INST/DATA_ACCESS_ADDR*x* and INST/DATA_ACCESS_MASK*x* may be '1')

16.1.2 Memory Protection Build Arguments

- **Instructions**
    - Instruction Access Window *x* (*x* = 0..7)
        - Enable (INST_ACCESS_ENABLE*x*): 0,1 *(0 = window disabled; 1 = window enabled)*
        - Base address (INST_ACCESS_ADDR*x*): 0x0000_0000..0xFFFF_FFC0 *(see Section 16.1.1)*
        - Mask (INST_ACCESS_MASK*x*): 0x0000_003F..0xFFFF_FFFF *(see Section 16.1.1)*
- **Data**
    - Data Access Window *x* (*x* = 0..7)
        - Enable (DATA_ACCESS_ENABLE*x*): 0,1 *(0 = window disabled; 1 = window enabled)*
        - Base address (DATA_ACCESS_ADDR*x*): 0x0000_0000..0xFFFF_FFC0 *(see Section 16.1.1)*
        - Mask (DATA_ACCESS_MASK*x*): 0x0000_003F..0xFFFF_FFFF *(see Section 16.1.1)*

16.2 Core Memory-Related Build Arguments

16.2.1 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.

16.2.2 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 16.2.1)*
    - 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 16.2.1)*
  - 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 16.2.1)*
  - Size (RV_PIC_SIZE): 32, 64, 128, 256 *(in KB)*

17 SweRV EH1 Compliance Test Suite Failures

17.1 I-MISALIGN_LDST-01

Test Location:

https://github.com/riscv/riscv-compliance/blob/master/riscv-test-suite/rv32i/src/I-MISALIGN_LDST-01.S

Reason for Failure:

The SweRV EH1 core supports unaligned accesses to memory addresses which are not marked as having side effects (i.e., to idempotent memory). Load and store accesses to non-idempotent memory addresses take misalignment exceptions.

(Note that this is a known issue with the test suite (https://github.com/riscv/riscv-compliance/issues/22) and is expected to eventually be fixed.)

Workaround:

Configure the address range used by this test to "non-idempotent" in the `mrac` register.

17.2 I-MISALIGN_JMP-01

Test Location:

https://github.com/riscv/riscv-compliance/blob/master/riscv-test-suite/rv32i/src/I-MISALIGN_JMP-01.S

Reason for Failure:

The SweRV EH1 core supports the standard "C" 16-bit compressed instruction extension. Compressed instruction execution cannot be turned off. Therefore, branch and jump instructions to 16-bit aligned memory addresses do not trigger misalignment exceptions.

(Note that this is a known issue with the test suite (https://github.com/riscv/riscv-compliance/issues/16) and is expected to eventually be fixed.)

Workaround:

None.

17.3 I-FENCE.I-01 and fence_i

Test Location:

https://github.com/riscv/riscv-compliance/blob/master/riscv-test-suite/rv32Zifencei/src/I-FENCE.I-01.S

and

https://github.com/riscv/riscv-compliance/blob/master/riscv-test-suite/rv32ui/src/fence_i.S

Reason for Failure:

The SweRV EH1 core implements separate instruction and data buses to the system interconnect (i.e., Harvard architecture). The latencies to memory through the system interconnect may be different for the two interfaces and the order is therefore not guaranteed.

Workaround:

Configuring the address range used by this test to "non-idempotent" in the `mrac` register forces the core to wait for a write response before fetching the updated line. Alternatively, the system interconnect could provide ordering guarantees between requests sent to the instruction fetch and load/store bus interfaces (e.g., matching latencies through the interconnect).

17.4 breakpoint

Test Location:

https://github.com/riscv/riscv-compliance/blob/master/riscv-test-suite/rv32mi/src/breakpoint.S

Reason for Failure:

The SweRV EH1 core disables breakpoints when the *mie* bit in the standard `mstatus` register is cleared.

(Note that this behavior is compliant with the RISC-V External Debug Support specification, Version 0.13.2. See Section 5.1, 'Native M-Mode Triggers' in [3] for more details.)

Workaround:

None.

18 SweRV EH1 Errata

18.1 Back-to-back Write Transactions Not Supported on AHB-Lite Bus

Description:

The AHB-Lite bus interface for LSU is not optimized for write performance. Each aligned store is issued to the bus as a single write transaction followed by an idle cycle. Each unaligned store is issued to the bus as multiple back-to-back byte write transactions followed by an idle cycle. These idle cycles limit the achievable bus utilization for writes.

Symptoms:

Potential performance impact for writes with AHB-Lite bus.

Workaround:

None.

18.2 Debug Write to minstret Register Stores Incremented Value

Description:

A debugger may attempt to initialize the `minstret` register to a specific value by using the access register abstract command. The abstract command's write operation itself is incorrectly counted as a retired instruction and causes the actually value written to the `minstret` register to be one higher than the intended value.

Symptoms:

When initializing the `minstret` register to a specific value from a debugger using the access register abstract command, then reading back this register indicates that the actual written value is one higher than the intended value.

Workaround:

When issuing an access register abstract command from a debugger to write the `minstret` register, the written value should be one less than the intended value to compensate for the incorrect increment. To initialize the 64-bit `minstret` counter to '0', the value 0xFFFF_FFFF must be written to the `minstreth` register first, followed by writing 0xFFFF_FFFF to the `minstret` register.

18.3 Debug Abstract Command Register May Return Non-Zero Value on Read

Description:

The RISC-V External Debug specification specifies the abstract command (`command`) register as write-only (see Section 3.14.7 in [3]). However, the SweRV EH1 implementation supports write as well as read operations to this register. This may help a debugger's feature discovery process, but is not fully compliant with the RISC-V External Debug specification. Because the expected return value for reading this register is always zero, it is unlikely that a debugger expecting a zero value would attempt to read it.

Symptoms:

Reading the debug abstract command (`command`) register may return a non-zero value.

Workaround:

A debugger should avoid reading the abstract command register if it cannot handle non-zero data.