Hazard3

Table of Contents

Chapter 1. Introduction

Hazard3 is a 3-stage RISC-V processor, providing the following architectural support:

- `RV32I`: 32-bit base instruction set
- `M`: integer multiply/divide/modulo
- `A`: atomic memory operations
- `C`: compressed instructions
- `Zba`: address generation
- `Zbb`: basic bit manipulation
- `Zbc`: carry-less multiplication
- `Zbs`: single-bit manipulation
- M-mode privileged instructions `ECALL`, `EBREAK`, `MRET`
- The `WFI` instruction
- `Zicsr`: CSR access
- The machine-mode (M-mode) privilege state, and standard M-mode CSRs
- Debug support, fully compliant with version 0.13.2 of the RISC-V external debug specification

The following are planned for future implementation:

- Trigger unit for debug mode
    - Likely breakpoints only

Chapter 2. Instruction Cycle Counts

All timings are given assuming perfect bus behaviour (no downstream bus stalls).

2.1. RV32I

Instruction	Cycles	Note
Integer Register-register		
`add rd, rs1, rs2`	1	
`sub rd, rs1, rs2`	1	
`slt rd, rs1, rs2`	1	
`sltu rd, rs1, rs2`	1	
`and rd, rs1, rs2`	1	
`or rd, rs1, rs2`	1	
`xor rd, rs1, rs2`	1	
`sll rd, rs1, rs2`	1	
`srl rd, rs1, rs2`	1	
`sra rd, rs1, rs2`	1	
Integer Register-immediate		
`addi rd, rs1, imm`	1	`nop` is a pseudo-op for `addi x0, x0, 0`
`slti rd, rs1, imm`	1	
`sltiu rd, rs1, imm`	1	
`andi rd, rs1, imm`	1	
`ori rd, rs1, imm`	1	
`xori rd, rs1, imm`	1	
`slli rd, rs1, imm`	1	
`srli rd, rs1, imm`	1	
`srai rd, rs1, imm`	1	
Large Immediate		
`lui rd, imm`	1	
`auipc rd, imm`	1	
Control Transfer		
`jal rd, label`	2[1]	
`jalr rd, rs1, imm`	2[1]	
`beq rs1, rs2, label`	1 or 2[1]	1 if nontaken, 2 if taken.

Instruction	Cycles	Note
`bne rs1, rs2, label`	1 or 2[1]	1 if nontaken, 2 if taken.
`blt rs1, rs2, label`	1 or 2[1]	1 if nontaken, 2 if taken.
`bge rs1, rs2, label`	1 or 2[1]	1 if nontaken, 2 if taken.
`bltu rs1, rs2, label`	1 or 2[1]	1 if nontaken, 2 if taken.
`bgeu rs1, rs2, label`	1 or 2[1]	1 if nontaken, 2 if taken.
Load and Store		
`lw rd, imm(rs1)`	1 or 2	1 if next instruction is independent, 2 if dependent.[2]
`lh rd, imm(rs1)`	1 or 2	1 if next instruction is independent, 2 if dependent.[2]
`lhu rd, imm(rs1)`	1 or 2	1 if next instruction is independent, 2 if dependent.[2]
`lb rd, imm(rs1)`	1 or 2	1 if next instruction is independent, 2 if dependent.[2]
`lbu rd, imm(rs1)`	1 or 2	1 if next instruction is independent, 2 if dependent.[2]
`sw rs2, imm(rs1)`	1	
`sh rs2, imm(rs1)`	1	
`sb rs2, imm(rs1)`	1	

2.2. M Extension

Timings assume the core is configured with `MULDIV_UNROLL = 2` and `MUL_FAST = 1`. I.e. the sequential multiply/divide circuit processes two bits per cycle, and a separate dedicated multiplier is present for the `mul` instruction.

Instruction	Cycles	Note
32 × 32 → 32 Multiply		
`mul rd, rs1, rs2`	1 or 2	1 if next instruction is independent, 2 if dependent.
32 × 32 → 64 Multiply, Upper Half		
`mulh rd, rs1, rs2`	18 to 20	Depending on sign correction
`mulhsu rd, rs1, rs2`	18 to 20	Depending on sign correction
`mulhu rd, rs1, rs2`	18	
Divide and Remainder		
`div rd, rs1, rs2`	18 or 19	Depending on sign correction
`divu rd, rs1, rs2`	18	
`rem rd, rs1, rs2`	18 or 19	Depending on sign correction
`remu rd, rs1, rs2`	18	

2.3. A Extension

Instruction	Cycles	Note
Load-Reserved/Store-Conditional		
`lr.w rd, (rs1)`	1 or 2	2 if next instruction is dependent[2], or an `lr.w`, `sc.w` or `amo*.w`.[3]
`sc.w rd, rs2, (rs1)`	1 or 2	2 if next instruction is an `lr.w`, `sc.w` or `amo*.w`.[3]
Atomic Memory Operations		
`amoswap.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amoadd.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amoxor.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amoand.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amoor.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amomin.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amomax.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amominu.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]
`amomaxu.w rd, rs2, (rs1)`	4+	4 per attempt. Multiple attempts if reservation is lost.[4]

2.4. C Extension

All C extension 16-bit instructions are aliases of base RV32I instructions. On Hazard3, they perform identically to their 32-bit counterparts.

A consequence of the C extension is that 32-bit instructions can be non-naturally-aligned. This has no penalty during sequential execution, but branching to a 32-bit instruction that is not 32-bit-aligned carries a 1 cycle penalty, because the instruction fetch is cracked into two naturally-aligned bus accesses.

2.5. Privileged Instructions (including Zicsr)

Instruction	Cycles	Note
CSR Access		
`csrrw rd, csr, rs1`	1	
`csrrc rd, csr, rs1`	1	
`csrrs rd, csr, rs1`	1	
`csrrwi rd, csr, imm`	1	
`csrrci rd, csr, imm`	1	
`csrrsi rd, csr, imm`	1	
Trap Request		

Instruction	Cycles	Note
`ecall`	3	Time given is for jumping to `mtvec`
`ebreak`	3	Time given is for jumping to `mtvec`

2.6. Bit Manipulation

Instruction	Cycles	Note
Zba (address generation)		
`sh1add rd, rs1, rs2`	1	
`sh2add rd, rs1, rs2`	1	
`sh3add rd, rs1, rs2`	1	
Zbb (basic bit manipulation)		
`andn rd, rs1, rs2`	1	
`clz rd, rs1`	1	
`cpop rd, rs1`	1	
`ctz rd, rs1`	1	
`max rd, rs1, rs2`	1	
`maxu rd, rs1, rs2`	1	
`min rd, rs1, rs2`	1	
`minu rd, rs1, rs2`	1	
`orc.b rd, rs1`	1	
`orn rd, rs1, rs2`	1	
`rev8 rd, rs1`	1	
`rol rd, rs1, rs2`	1	
`ror rd, rs1, rs2`	1	
`rori rd, rs1, imm`	1	
`sext.b rd, rs1`	1	
`sext.h rd, rs1`	1	
`xnor rd, rs1, rs2`	1	
`zext.h rd, rs1`	1	
`zext.b rd, rs1`	1	`zext.b` is a pseudo-op for `andi rd, rs1, 0xff`
Zbc (carry-less multiply)		
`clmul rd, rs1, rs2`	1	
`clmulh rd, rs1, rs2`	1	
`clmulr rd, rs1, rs2`	1	

Instruction	Cycles	Note
Zbs (single-bit manipulation)		
`bclr rd, rs1, rs2`	1	
`bclri rd, rs1, imm`	1	
`bext rd, rs1, rs2`	1	
`bexti rd, rs1, imm`	1	
`binv rd, rs1, rs2`	1	
`binvi rd, rs1, imm`	1	
`bset rd, rs1, rs2`	1	
`bseti rd, rs1, imm`	1	

[1] A branch to a 32-bit instruction which is not 32-bit-aligned requires one additional cycle, because two naturally aligned bus cycles are required to fetch the target instruction.

[2] If an instruction uses load data (from stage 3) in stage 2, a 1-cycle bubble is inserted after the load. Load-data to store-data dependency does not experience this, because the store data is used in stage 3. However, load-data to store-address (or e.g. load-to-add) does qualify.

[3] A pipeline bubble is inserted between `lr.w`/`sc.w` and an immediately-following `lr.w`/`sc.w`/`amo*`, because the AHB5 bus standard does not permit pipelined exclusive accesses. A stall would be inserted between `lr.w` and `sc.w` anyhow, so the local monitor can be updated based on the `lr.w` data phase in time to suppress the `sc.w` address phase.

[4] AMOs are issued as a paired exclusive read and exclusive write on the bus, at the maximum speed of 2 cycles per access, since the bus does not permit pipelining of exclusive reads/writes. If the write phase fails due to the global monitor reporting a lost reservation, the instruction loops at a rate of 4 cycles per loop, until success. If the read reservation is refused by the global monitor, the instruction generates a Store/AMO Fault exception, to avoid an infinite loop.

Chapter 3. CSRs

The RISC-V privileged specification affords flexibility as to which CSRs are implemented, and how they behave. This section documents the concrete behaviour of Hazard3's standard and nonstandard M-mode CSRs, as implemented.

All CSRs are 32-bit; MXLEN is fixed at 32 bits on Hazard3. All CSR addresses not listed in this section are unimplemented. Accessing an unimplemented CSR will cause an illegal instruction exception (`mcause` = 2). This includes all U-mode and S-mode CSRs.

IMPORTANT

The RISC-V Privileged Specification should be your primary reference for writing software to run on Hazard3. This section specifies those details which are left implementation-defined by the RISC-V Privileged Specification, for sake of completeness, but portable RISC-V software should not rely on these details.

3.1. Standard M-mode Identification CSRs

3.1.1. mvendorid

Address: `0xf11`

Vendor identifier. Read-only, configurable constant. Should contain either all-zeroes, or a valid JEDEC JEP106 vendor ID using the encoding in the RISC-V specification.

Bits	Name	Description
31:7	`bank`	The number of continuation codes in the vendor JEP106 ID. *One less than the JEP106 bank number.*
6:0	`offset`	Vendor ID within the specified bank. LSB (parity) is not stored.

3.1.2. marchid

Address: `0xf12`

Architecture identifier for Hazard3. Read-only, constant.

Bits	Name	Description
31	-	0: Open-source implementation
30:0	-	0x1b (decimal 27): the registered architecture ID for Hazard3

3.1.3. mimpid

Address: `0xf13`

Implementation identifier. Read-only, configurable constant.

Bits	Name	Description
31:0	**-**	Should contain the git hash of the Hazard3 revision from which the processor was synthesised, or all-zeroes.

3.1.4. mhartid

Address: `0xf14`

Hart identification register. Read-only, configurable constant.

Bits	Name	Description
31:0	**-**	Hazard3 cores possess only one hardware thread, so this is a unique per-core identifier, assigned consecutively from 0.

3.1.5. mconfigptr

Address: `0xf15`

Pointer to configuration data structure. Read-only, configurable constant.

Bits	Name	Description
31:0	**-**	Either pointer to configuration data structure, containing information about the harts and system, or all-zeroes. At least 4-byte-aligned.

3.1.6. misa

Address: `0x301`

Read-only, constant. Value depends on which ISA extensions Hazard3 is configured with. The table below lists the fields which are *not* always hardwired to 0:

Bits	Name	Description
31:30	`mxl`	Always `0x1`. Indicates this is a 32-bit processor.
23	`x`	1 if the core is configured to support trap-handling, otherwise 0. Hazard3 has nonstandard CSRs to enable/disable external interrupts on a per-interrupt basis, see meie0 and meip0. The `misa.x` bit must be set to indicate their presence. Hazard3 does not implement any custom instructions.
12	`m`	1 if the M extension is present, otherwise 0.
2	`c`	1 if the C extension is present, otherwise 0.
0	`a`	1 if the A extension is present, otherwise 0.

3.2. Standard M-mode Trap Handling CSRs

3.2.1. mstatus

Address: `0x300`

The below table lists the fields which are *not* hardwired to 0:

Bits	Name	Description
12:11	`mpp`	Previous privilege level. Always `0x3`, indicating M-mode.
7	`mpie`	Previous interrupt enable. Readable and writable. Is set to the current value of `mstatus.mie` on trap entry. Is set to 1 on trap return.
3	`mie`	Interrupt enable. Readable and writable. Is set to 0 on trap entry. Is set to the current value of `mstatus.mpie` on trap return.

3.2.2. mstatush

Address: `0x310`

Hardwired to 0.

3.2.3. medeleg

Address: `0x302`

Unimplemented, as only M-mode is supported. Access will cause an illegal instruction exception.

3.2.4. mideleg

Address: `0x303`

Unimplemented, as only M-mode is supported. Access will cause an illegal instruction exception.

3.2.5. mie

Address: `0x304`

Interrupt enable register. Not to be confused with `mstatus.mie`, which is a global enable, having the final say in whether any interrupt which is both enabled in `mie` and pending in `mip` will actually cause the processor to transfer control to a handler.

The table below lists the fields which are *not* hardwired to 0:

Bits	Name	Description
11	`meie`	External interrupt enable. Hazard3 has internal custom CSRs to further filter external interrupts, see meie0.

Bits	Name	Description
7	mtie	Timer interrupt enable. A timer interrupt is requested when mie.mtie, mip.mtip and mstatus.mie are all 1.
3	msie	Software interrupt enable. A software interupt is requested when mie.msie, mip.mtip and mstatus.mie are all 1.

NOTE RISC-V reserves bits 16+ of mie/mip for platform use, which Hazard3 could use for external interrupt control. On RV32I this could only control 16 external interrupts, so Hazard3 instead adds nonstandard interrupt enable registers starting at meie0, and keeps the upper half of mie reserved.

3.2.6. mip

Address: 0x344

Interrupt pending register. Read-only.

NOTE The RISC-V specification lists mip as a read-write register, but the bits which are writable correspond to lower privilege modes (S- and U-mode) which are not implemented on Hazard3, so it is documented here as read-only.

The table below lists the fields which are *not* hardwired to 0:

Bits	Name	Description
11	meip	External interrupt pending. When 1, indicates there is at least one interrupt which is asserted (hence pending in meip0) and enabled in meie0.
7	mtip	Timer interrupt pending. Level-sensitive interrupt signal from outside the core. Connected to a standard, external RISC-V 64-bit timer.
3	msip	Software interrupt pending. In spite of the name, this is not triggered by an instruction on this core, rather it is wired to an external memory-mapped register to provide a cross-hart level-sensitive doorbell interrupt.

NOTE Hazard3 assumes interrupts to be level-sensitive at system level. Bits in mip are cleared by servicing the requestor and causing it to deassert its interrupt request.

3.2.7. mtvec

Address: 0x305

Trap vector base address. Read-write. Exactly which bits of mtvec can be modified (possibly none) is configurable when instantiating the processor, but by default the entire register is writable. The reset value of mtvec is also configurable.

Bits	Name	Description
31:2	base	Base address for trap entry. In Vectored mode, this is *OR'd* with the trap offset to calculate the trap entry address, so the table must be aligned to its total size, rounded up to a power of 2. In Direct mode, base is word-aligned.
0	mode	0 selects Direct mode — all traps (whether exception or interrupt) jump to base. 1 selects Vectored mode — exceptions go to base, interrupts go to base | mcause << 2.

NOTE In the RISC-V specification, mode is a 2-bit write-any read-legal field in bits 1:0. Hazard3 implements this by hardwiring bit 1 to 0.

3.2.8. mscratch

Address: 0x340

Read-write 32-bit register. No specific hardware function — available for software to swap with a register when entering a trap handler.

3.2.9. mepc

Address: 0x341

Exception program counter. When entering a trap, the current value of the program counter is recorded here. When executing an mret, the processor jumps to mepc. Can also be read and written by software.

On Hazard3, bits 31:1 of mepc are capable of holding all 31-bit values. Bit 0 is hardwired to 0, as per the specification.

All traps on Hazard3 are precise. For example, a load/store bus error will set mepc to the exact address of the load/store instruction which encountered the fault.

3.2.10. mcause

Address: 0x342

Exception cause. Set when entering a trap to indicate the reason for the trap. Readable and writable by software.

NOTE On Hazard3, most bits of mcause are hardwired to 0. Only bit 31, and enough least-significant bits to index all exception and all interrupt causes (at least four bits), are backed by registers. Only these bits are writable; the RISC-V specification only requires that mcause be able to hold all legal cause values.

The most significant bit of mcause is set to 1 to indicate an interrupt cause, and 0 to indicate an exception cause. The following interrupt causes may be set by Hazard3 hardware:

Cause	Description
3	Software interrupt (`mip.msip`)
7	Timer interrupt (`mip.mtip`)
11	External interrupt (`mip.meip`)

The following exception causes may be set by Hazard3 hardware:

Cause	Description
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store/AMO address misaligned
7	Store/AMO access fault
11	Environment call

NOTE Not every instruction fetch bus cycle which returns a bus error leads to an exception. Hazard3 prefetches instructions ahead of execution, and associated bus errors are speculated through to the point the processor actually attempts to decode the instruction. Until this point, the error can be flushed by a branch, with no ill effect.

3.2.11. mtval

Address: `0x343`

Hardwired to 0.

3.2.12. mcounteren

Address: `0x306`

Unimplemented, as only M-mode is supported. Access will cause an illegal instruction exception.

Not to be confused with mcountinhibit.

3.3. Standard Memory Protection

3.3.1. pmpcfg0…3

Address: `0x3a0` through `0x3a3`

Unimplemented. Access will cause an illegal instruction exception.

3.3.2. pmpaddr0...15

Address: `0x3b0` through `0x3bf`

Unimplemented. Access will cause an illegal instruction exception.

3.4. Standard M-mode Performance Counters

3.4.1. mcycle

Address: `0xb00`

Lower half of the 64-bit cycle counter. Readable and writable by software. Increments every cycle, unless `mcountinhibit.cy` is 1, or the processor is in Debug Mode (as dcsr.`stopcount` is hardwired to 1).

If written with a value `n` and read on the very next cycle, the value read will be exactly `n + 1` (ignoring wrapping).

3.4.2. mcycleh

Address: `0xb80`

Upper half of the 64-bit cycle counter. Readable and writable by software. Increments every time `mcycle` wraps from `0xffffffff` to `0x00000000` upon increment.

3.4.3. minstret

Address: `0xb02`

Lower half of the 64-bit instruction retire counter. Readable and writable by software. Increments with every instruction exectued, unless `mcountinhibit.ir` is 1, or the processor is in Debug Mode (as dcsr.`stopcount` is hardwired to 1).

3.4.4. minstreth

Address: `0xb82`

Upper half of the 64-bit instruction retire counter. Readable and writable by software. Increments every time `minstret` wraps from `0xffffffff` to `0x00000000` upon increment.

3.4.5. mhpmcounter3...31

Address: `0xb03` through `0xb1f`

Hardwired to 0.

3.4.6. mhpmcounter3...31h

Address: `0xb83` through `0xb9f`

Hardwired to 0.

3.4.7. mcountinhibit

Address: `0x320`

Counter inhibit. Read-write. The table below lists the fields which are *not* hardwired to 0:

Bits	Name	Description
2	`ir`	When 1, inhibit counting of `minstret`/`minstreth`
0	`cy`	When 1, inhibit counting of `mcycle`/`mcycleh`

3.4.8. mhpmevent3…31

Address: `0x323` through `0x33f`

Hardwired to 0.

3.5. Standard Trigger CSRs

3.5.1. tselect

Address: `0x7a0`

Unimplemented. Reads as 0, write causes illegal instruction exception.

3.5.2. tdata1…3

Address: `0x7a1` through `0x7a3`

Unimplemented. Access will cause an illegal instruction exception.

3.6. Standard Debug Mode CSRs

This section describes the Debug Mode CSRs, which follow the 0.13.2 RISC-V debug specification. The Debug section gives more detail on the remainder of Hazard3's debug implementation, including the Debug Module.

All Debug Mode CSRs are 32-bit; DXLEN is always 32.

3.6.1. dcsr

Address: `0x7b0`

Debug control and status register. Access outside of Debug Mode will cause an illegal instruction exception. Relevant fields are implemented as follows:

Bits	Name	Description
31:28	xdebugver	Hardwired to 4: external debug support as per RISC-V 0.13.2 debug specification.
15	ebreakm	When 1, ebreak instructions will break to Debug Mode instead of trapping in M mode.
11	stepie	Hardwired to 0: no interrupts are taken during hardware single-stepping.
10	stopcount	Hardwired to 1: mcycle/mcycleh and minstret/minstreth do not increment in Debug Mode.
9	stoptime	Hardwired to 1: core-local timers don't increment in debug mode. This requires cooperation of external hardware based on the halt status to implement correctly.
8:6	cause	Read-only, set by hardware — see table below.
2	step	When 1, re-enter Debug Mode after each instruction executed in M-mode.
1:0	prv	Hardwired to 3, as only M-mode is implemented.

Fields not mentioned above are hardwired to 0.

Hazard3 may set the following dcsr.cause values:

Cause	Description
1	Processor entered Debug Mode due to an ebreak instruction executed in M-mode.
3	Processor entered Debug Mode due to a halt request, or a reset-halt request present when the core reset was released.
4	Processor entered Debug Mode after executing one instruction with single-stepping enabled.

Cause 5 (resethaltreq) is never set by hardware. This event is reported as a normal halt, cause 3. Cause 2 (trigger) is never used because there are no triggers. (TODO?)

3.6.2. dpc

Address: 0x7b1

Debug program counter. When entering Debug Mode, dpc samples the current program counter, e.g. the address of an ebreak which caused Debug Mode entry. When leaving debug mode, the processor jumps to dpc. The host may read/write this register whilst in Debug Mode.

3.6.3. dscratch0

Address: 0x7b2

Not implemented. Access will cause an illegal instruction exception.

To provide data exchange between the Debug Module and the core, the Debug Module's `data0` register is mapped into the core's CSR space at a read/write M-custom address — see dmdata0.

3.6.4. dscratch1

Address: `0x7b3`

Not implemented. Access will cause an illegal instruction exception.

3.7. Custom CSRs

These are all allocated in the space `0xbc0` through `0xbff` which is available for custom read/write M-mode CSRs, and `0xfc0` through `0xfff` which is available for custom read-only M-mode CSRs.

Hazard3 also allocates a custom *Debug Mode* register dmdata0 in this space.

3.7.1. dmdata0

Address: `0xbff`

The Debug Module's internal `data0` register is mapped to this CSR address when the core is in debug mode. At any other time, access to this CSR address will cause an illegal instruction exception.

NOTE	The 0.13.2 debug specification allows for the Debug Module's abstract data registers to be mapped into the core's CSR address space, but there is no Debug-custom space, so the read/write M-custom space is used instead to avoid conflict with future versions of the debug specification.

The Debug Module uses this mapping to exchange data with the core by injecting `csrr`/`csrw` instructions into the prefetch buffer. This in turn is used to implement the Abstract Access Register command. See Debug.

This CSR address is given by the `dataaddress` field of the Debug Module's `hartinfo` register, and `hartinfo.dataaccess` is set to 0 to indicate this is a CSR mapping, not a memory mapping.

3.7.2. meie0

Address: `0xbe0`

External interrupt enable register 0. Contains a read-write bit for each external interrupt request IRQ0 through IRQ31. A `1` bit indicates that interrupt is currently enabled.

Addresses `0xbe1` through `0xbe3` are reserved for further `meie` registers, supporting up to 128 external interrupts.

An external interrupt is taken when all of the following are true:

- The interrupt is currently asserted in `meip0`
- The matching interrupt enable bit is set in `meie0`

- The standard M-mode interrupt enable `mstatus.mie` is set
- The standard M-mode global external interrupt enable `mie.meie` is set

`meie0` resets to **all-ones**, for compatibility with software which is only aware of `mstatus` and `mie`. Because `mstatus.mie` and `mie.meie` are both initially clear, the core will not take interrupts straight out of reset, but it is strongly recommended to configure `meie0` before setting the global interrupt enable, to avoid interrupts from unexpected sources.

3.7.3. meip0

Address: `0xfe0`

External IRQ pending register 0. Contains a read-only bit for each external interrupt request IRQ0 through IRQ31. A `1` bit indicates that interrupt is currently asserted. IRQs are assumed to be level-sensitive, and the relevant `meip0` bit is cleared by servicing the requestor so that it deasserts its interrupt request.

Addresses `0xfe1` through `0xfe3` are reserved for further `meip` registers, supporting up to 128 external interrupts.

When any bit is set in both `meip0` and `meie0`, the standard external interrupt pending bit `mip.meip` is also set. In other words, `meip0` is filtered by `meie0` to generate the standard `mip.meip` flag. So, an external interrupt is taken when *all* of the following are true:

- An interrupt is currently asserted in `meip0`
- The matching interrupt enable bit is set in `meie0`
- The standard M-mode interrupt enable `mstatus.mie` is set
- The standard M-mode global external interrupt enable `mie.meie` is set

In this case, the processor jumps to either:

- `mtvec` directly, if vectoring is disabled (`mtvec[0]` is 0)
- `mtvec + 0x2c`, if vectoring is enabled (`mtvec[0]` is 1)

3.7.4. mlei

Address: `0xfe4`

Lowest external interrupt. Contains the index of the lowest-numbered external interrupt which is both asserted in `meip0` and enabled in `meie0`, left-shifted by 2 so that it can be used to index an array of 32-bit function pointers.

Bits	Name	Description
31:7	-	RES0
6:2	-	Index of the lowest-numbered active external interrupt. A LSB-first priority encode of `meip0 & meie0`. Zero when no external interrupts are both pending and enabled.

Bits	Name	Description
1:0	**-**	RES0

Chapter 4. Debug

Hazard3, along with its external debug components, implements version 0.13.2 of the RISC-V debug specification. It supports the following:

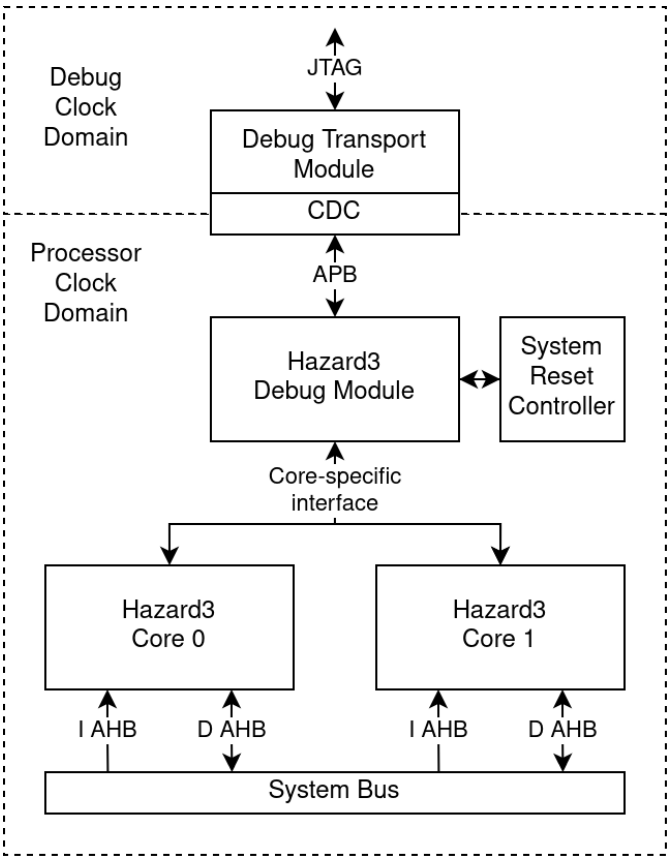
- Run/halt/reset control as required
- Abstract GPR access as required
- Program Buffer, 2 words plus `impebreak`
- Automatic trigger of abstract command (`abstractauto`) on `data0` or Program Buffer access for efficient memory block transfers from the host
- (TODO) Some minimum useful trigger unit — likely just breakpoints, no watchpoints

4.1. Debug Topologies

Hazard3's Debug Module has the following interfaces:

- An upstream AMBA 3 APB port — the "Debug Module Interface" — for host access to the Debug Module
- A downstream Hazard3-specific interface to one or more cores *(multicore support is experimental)*
- Some reset request/acknowledge signals which require careful handshaking with system-level reset logic

This is shown in the example topology below.



The Debug Module *must* be connected directly to the processors without intervening registers. This implies the Debug Module is in the same clock domain as the processors, so multiple processors on the same Debug Module must share a common clock.

Upstream of the Debug Module is at least one Debug Transport Module, which bridges some host-facing interface such as JTAG to the APB Debug Module Interface. Hazard3 provides an implementation of a standard RISC-V JTAG-DTM, but any APB master could be used. The Debug Module requires at least 7 bits of word addressing, i.e. 9 bits of byte address space.

An APB arbiter could be inserted here, to allow multiple transports to be used, provided the host(s) avoid using multiple transports concurrently. This also admits simple implementation of self-hosted debug, by mapping the Debug Module to a system-level peripheral address space.

The clock domain crossing (if any) occurs on the downstream port of the Debug Transport Module. Hazard3's JTAG-DTM implementation runs entirely in the TCK domain, and instantiates a bus clock-crossing module internally to bridge a TCK-domain internal APB bus to an external bus in the processor clock domain.

It is possible to instantiate multiple Debug Modules, one per core, and attach them to a single Debug Transport Module. This is not the preferred topology, but it does allow multiple cores to be independently clocked.

4.2. Implementation-defined behaviour

Features implemented by the Hazard3 Debug Module (beyond the mandatory):

- Halt-on-reset, selectable per-hart
- Program Buffer, size 2 words, `impebreak` = 1.
- A single data register (`data0`) is implemented as a per-hart CSR accessible by the DM
- `abstractauto` is supported on the `data0` register
- Up to 32 harts selectable via `hartsel`

Not implemented:

- Hart array mask selection
- Abstract access memory
- Abstract access CSR
- Post-incrementing abstract access GPR
- System bus access

The core behaves as follows:

- Branch, `jal`, `jalr` and `auipc` are illegal in debug mode, because they observe PC: attempting to execute will halt Program Buffer execution and report an exception in `abstractcs.cmderr`
- The `dret` instruction is not implemented (a special purpose DM-to-core signal is used to signal resume)

- The `dscratch` CSRs are not implemented
- The DM's `data0` register is mapped into the core as a CSR, dmdata0, address `0xbff`.
  - Raises an illegal instruction exception when accessed outside of Debug Mode
  - The DM ignores attempted core writes to the CSR, unless the DM is currently executing an abstract command on that core
  - Used by the DM to implement abstract GPR access, by injecting CSR read/write instructions
- `dcsr.stepie` is hardwired to 0 (no interrupts during single stepping)
- `dcsr.stopcount` and `dcsr.stoptime` are hardwired to 1 (no counter or internal timer increment in debug mode)
- `dcsr.mprven` is hardwired to 0
- `dcsr.prv` is hardwired to 3 (M-mode)

See also Standard Debug Mode CSRs for more details on the core-side Debug Mode registers.

The debug host must use the Program Buffer to access CSRs and memory. This carries some overhead for individual accesses, but is efficient for bulk transfers: the `abstractauto` feature allows the DM to trigger the Program Buffer and/or a GPR tranfer automatically following every `data0` access, which can be used for e.g. autoincrementing read/write memory bursts. Program Buffer read/writes can also be used as `abstractauto` triggers: this is less useful than the `data0` trigger, but takes little extra effort to implement, and can be used to read/write a large number of CSRs efficiently.

Abstract memory access is not implemented because, for bulk transfers, it offers no better throughput than Program Buffer execution with `abstractauto`. Non-bulk transfers, while slower, are still instantaneous from the perspective of the human at the other end of the wire.

The Hazard3 Debug Module has experimental support for multi-core debug. Each core possesses exactly one hardware thread (hart) which is exposed to the debugger. The RISC-V specification does not mandate what mapping is used between the Debug Module hart index `hartsel` and each core's `mhartid` CSR, but a 1:1 match of these values is the least likely to cause issues. Each core's `mhartid` can be configured using the `MHARTID_VAL` parameter during instantiation.

4.3. Debug Module to Core Interface

The DM can inject instructions directly into the core's instruction prefetch buffer. This mechanism is used to execute the Program Buffer, or used directly by the DM, issuing hardcoded instructions to manipulate core state.

The DM's `data0` register is exposed to the core as a debug mode CSR. By issuing instructions to make the core read or write this dummy CSR, the DM can exchange data with the core. To read from a GPR `x` into `data0`, the DM issues a `csrw data0, x` instruction. Similarly `csrr x, data0` will write `data0` to that GPR. The DM always follows the CSR instruction with an `ebreak`, just like the implicit `ebreak` at the end of the Program Buffer, so that it is notified by the core when the GPR read instruction sequence completes.

TODO reset interface description