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
- `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:

- `A` extension: atomic memory access
  - `LR`/`SC` fully supported
  - AMONone PMA on all of memory (AMOs are decoded but unconditionally trigger access fault without attempting memory access)
- Trigger unit for debug mode
  - Likely breakpoints only

Chapter 2. Instruction Cycle Counts

All timings are given assuming perfect bus behaviour (no stalls). Stalling of the `I` bus can delay execution indefinitely, as can stalling of the `D` bus during a load or store.

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`	18 or 19	Depending on sign correction
`divu`	18	
`rem`	18 or 19	Depending on sign correction
`remu`	18	

2.3. C Extension

All C extension 16-bit instructions on Hazard3 are aliases of base RV32I instructions. 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.

2.4. 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		
`ecall`	3	Time given is for jumping to `mtvec`
`ebreak`	3	Time given is for jumping to `mtvec`

2.5. Bit Manipulation

Instruction	Cycles	Note
Zba (address generation)		
`sh1add`	1	
`sh2add`	1	
`sh3add`	1	
Zbb (basic bit manipulation)		
`andn`	1	
`clz`	1	
`cpop`	1	
`ctz`	1	
`max`	1	
`maxu`	1	

Instruction	Cycles	Note
`min`	1	
`minu`	1	
`orc.b`	1	
`orn`	1	
`rev8`	1	
`rol`	1	
`ror`	1	
`rori`	1	
`sext.b`	1	
`sext.h`	1	
`xnor`	1	
`zext.h`	1	
`zext.b`	1	`zext.b` is a pseudo-op for `andi rd, rs1, 0xff`
Zbc (carry-less multiply)		
`clmul`	1	
`clmulh`	1	
`clmulr`	1	
Zbs (single-bit manipulation)		
`bclr`	1	
`bclri`	1	
`bext`	1	
`bexti`	1	
`binv`	1	
`binvi`	1	
`bset`	1	
`bseti`	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.

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.

NOTE 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.

All CSRs are 32-bit; MXLEN is fixed at 32 bits on Hazard3.

3.1. Standard M-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.

3.1.1. mvendorid

Address: `0xf11`

Vendor identifier. Read-only, configurable constant.

Bits	Name	Description
31:0	**-**	Should contain either all-zeroes, or a valid JEDEC JEP106 vendor ID using the encoding in the RISC-V specification (*not* the same as a JTAG IDCODE)

3.1.2. marchid

Address: `0xf12`

Architecture identifier for Hazard3. Read-only, configurable constant.

Bits	Name	Description
31:0	**-**	Default is currently all-zeroes as Hazard3 is unregistered.

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. 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.

3.1.6. mstatush

Address: `0x310`

This CSR is present, but it is entirely hardwired to zero. Its presence is required for compliance.

3.1.7. 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.

Bits	Name	Description
12	`m`	1 if the M extension is present, otherwise 0.
2	`c`	1 if the C extension is present, otherwise 0.

3.1.8. medeleg

Address: `0x302`

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

3.1.9. mideleg

Address: `0x303`

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

3.1.10. 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.
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.

3.1.11. 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.

3.1.12. 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.

NOTE Hazard3 has an additional nonstandard vectoring mode, where external interrupts are each separated into distinct vectors and `mcause` values. This is enabled through the implementation-defined control register, midcr, since the RISC-V specification reserves `mtvec.mode == 2, 3` for future standard use.

3.1.13. mcounteren

Address: `0x306`

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

Not to be confused with mcountinhibit.

3.1.14. 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.

3.1.15. 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.

3.1.16. 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`)

Numbers >16 are used for to disambiguate between external IRQs when expanded vectoring is enabled — see midcr.

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.

3.1.17. mtval

Address: `0x343`

Hardwired to 0.

3.1.18. pmpcfg0…3

Address: `0x3a0` through `0x3a3`

Unimplemented. Access will cause an illegal instruction exception.

3.1.19. pmpaddr0…15

Address: `0x3b0` through `0x3bf`

Unimplemented. Access will cause an illegal instruction exception.

3.1.20. 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).

3.1.21. 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.

3.1.22. 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).

3.1.23. 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.

3.1.24. mhpmcounter3…31

Address: `0xb03` through `0xb1f`

Hardwired to 0.

3.1.25. mhpmcounter3…31h

Address: `0xb83` through `0xb9f`

Hardwired to 0.

3.1.26. 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

3.1.27. mhpmevent3…31

Address: `0x323` through `0x33f`

Hardwired to 0.

3.1.28. tselect

Address: `0x7a0`

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

3.1.29. tdata1…3

Address: `0x7a1` through `0x7a3`

Unimplemented. Access will cause an illegal instruction exception.

3.2. Standard Debug Mode CSRs

This section describes the Debug Mode CSRs, which are 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.

3.2.1. 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.

Bits	Name	Description
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?)

3.2.2. 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.

3.2.3. dscratch0

Address: `0x7b2`

Not implemented. However, the Debug Module's internal `data0` register is mapped to this CSR address under the following conditions:

- The core is in Debug Mode
- The Debug Module is *currently executing an abstract command on this core*

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.

The Debug Module lists the number of scratch registers as 0 in `hartinfo.dscratch`.

3.2.4. dscratch1

Not implemented. Access will cause an illegal instruction exception.

3.3. 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.

3.3.1. midcr

Address: `0xbc0`

Implementation-defined control register. Miscellaneous nonstandard controls.

Bits	Name	Description
31:1	-	RES0
0	`eivect`	Modified external interrupt vectoring. If 0, use standard behaviour: all external interrupts set interrupt `mcause` of 11 and vector to `mtvec + 0x2c`. If 1, external interrupts use distinct interrupt `mcause` numbers 16 upward, and distinct vectors `mtvec + (irq + 16) * 4`. Resets to 0. Has no effect when `mtvec[0]` is 0.

3.3.2. 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.

3.3.3. 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) and modified external IRQ vectoring is disabled (`midcr.eivect` is 0)
- `mtvect + (mlei + 16) * 4`, if vectoring is enabled (`mtvec[0]` is 1) and modified external IRQ vectoring is enabled (`midcr.eivect` is 1). `
    - `mlei` is a read-only CSR containing the lowest-numbered pending-and-enabled external interrupt.

3.3.4. 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`. Can be used for faster software vectoring when modified external interrupt vectoring (`midcr.eivect = 1`) is not in use.

Bits	Name	Description
31:5	-	RES0
4:0	-	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.

3.3.5. Maybe-adds

An option to clear a bit in `meie0` when that interrupt is taken, and set it when an `mret` has a matching `mcause` for that interrupt. Makes preemption support easier.

Chapter 4. Debug

Hazard3, along with its external debug components, implements version 0.13.2 of the RISC-V debug specification. The goals of this implementation are:

- Minimal impact on core timing when present
- No external components which need integrating at the other end of your bus fabric — just slap the Debug Module onto the core and away you go
- Efficient block data transfers to target RAM for faster edit-compile-run cycle

Hazard3's debug support implements the following:

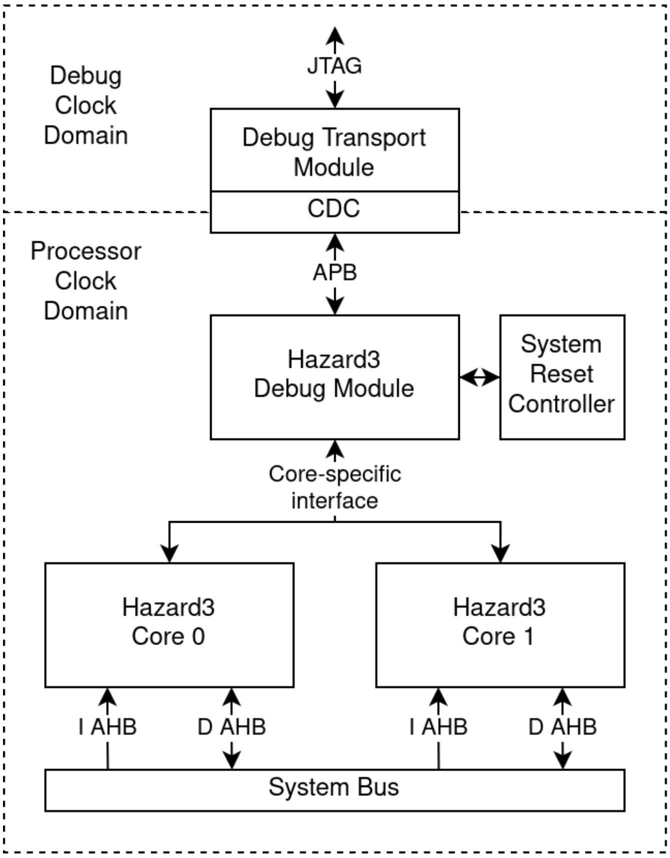
- Run/halt/reset control as required
- Abstract GPR access as required
- Program Buffer, 2 words plus `impebreak`
- Automatic trigger of abstract command (`abstractauto`) on `data0` or Program Buffer access for efficient memory block transfers from the host
- (TODO) Some minimum useful trigger unit — 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

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. This is shown in the example topology below.



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. An APB arbiter could be inserted here, to allow multiple transports to be used, provided the host(s) avoid using multiple transports concurrently.

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.

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. 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.

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 it offers no better throughput than Program Buffer execution with `abstractauto` for bulk transfers, and non-bulk transfers 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. Implementation-defined behaviour

Features implemented by DM (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

Core behaviour:

- 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
- External `data0` register is exposed as a dummy CSR mapped at `0x7b2` (the location of `dscratch0`), readable and writable by the DM.
  - This is a debug mode CSR, so raises an illegal instruction exception when accessed in machine mode
  - The DM ignores writes unless it is currently executing an abstract command on this core (`hartsel` = this core, `abstractcs.busy` = 1)
- `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)

4.4. UART DTM

Hazard3 defines a minimal UART Debug Transport Module, which allows the Debug Module to be accessed via a standard 8n1 asynchronous serial port. The UART DTM is always accessed by the host using a two-wire serial interface (TXD RXD) running at 1 Mbaud. The interface between the DTM and DM is an AMBA 3 APB port with a 32-bit data bus and 8-bit address bus.

This is a quick hack, and not suitable for production systems:

- Debug hardware should not expect a frequency reference for a UART to be present
- The UART DTM does not implement any flow control or error detection/correction

The host may send the following commands:

Command	To DTM	From DTM
`0x00` **NOP**	-	-
`0x01` **Read ID**	-	4-byte ID, same format as JTAG-DTM ID (JEP106-compatible)
`0x02` **Read DMI**	1 address byte	4 data bytes
`0x03` **Write DMI**	1 address byte, 4 data bytes	data bytes echoed back
`0xa5` **Disconnect**	-	-

Initially after power-on the DTM is in the Dormant state, and will ignore any commands. The host sends the magic sequence `"SUP?"` (`0x53, 0x55, 0x50, 0x3f`) to wake the DTM, and then issues a Read ID command to check the link is up. The DTM can be returned to the Dormant state at any time using the `0xa5` Disconnect command.

So that the host can queue up batches of commands in its transmit buffer, without overrunning the DTM's transmit bandwidth, it's recommended to pad each command with NOPs so that it is strictly larger than the response. For example, a Read ID should be followed by four NOPs, and a Read DMI should be followed by 3 NOPs.

To recover command framing, write 6 NOP commands (the length of the longest commands). This will be interpreted as between 1 and 6 NOPs depending on the DTM's state.

This interface assumes the DMI data transfer takes very little time compared with the UART access (typically less than one baud period). When the host-to-DTM bandwidth is kept greater than the DTM-to-host bandwidth, thanks to appropriate NOP padding, the host can queue up batches of commands in its transmit buffer, and this should never overrun the DTM's response channel. So, the 1 Mbaud 8n1 UART link provides 67 kB/s of half-duplex data bandwidth between host and DM, which is enough to get your system off the ground.