Hazard3

Updated: 2024-Aug-07

Table of Contents

1. Introduction 1
  1.1. Architectural Overview 1
    1.1.1. Pipeline Stages 1
    1.1.2. Bus Interfaces 2
    1.1.3. Multiply/Divide 2
  1.2. List of RISC-V Specifications 3
2. Configuration and Integration 4
  2.1. Hazard3 Source Files 4
  2.2. Top-level Modules 4
  2.3. FPGA Synthesis 5
  2.4. ASIC Synthesis 5
  2.5. Interfaces (Top-level Ports) 5
    2.5.1. Interfaces Common to All Wrappers 5
    2.5.2. Interfaces for 1-port AHB5 CPU 9
    2.5.3. Interfaces for 2-port AHB5 CPU 11
  2.6. Configuration Parameters 13
    2.6.1. Reset state configuration 13
    2.6.2. Standard RISC-V ISA support 13
    2.6.3. Custom Hazard3 Extensions 15
    2.6.4. CSR support 16
    2.6.5. External interrupt support 18
    2.6.6. Identification Registers 18
    2.6.7. Performance/size options 19
3. CSRs 21
  3.1. Standard M-mode Identification CSRs 21
    3.1.1. mvendorid 21
    3.1.2. marchid 21
    3.1.3. mimpid 21
    3.1.4. mhartid 22
    3.1.5. mconfigptr 22
    3.1.6. misa 22
  3.2. Standard M-mode Trap Handling CSRs 22
    3.2.1. mstatus 23
    3.2.2. mstatush 23
    3.2.3. medeleg 23
    3.2.4. mideleg 23
    3.2.5. mie 23
    3.2.6. mip 24

3.2.7. mtvec 25
3.2.8. mscratch 25
3.2.9. mepc 25
3.2.10. mcause 25
3.2.11. mtval 26
3.2.12. mcounteren 26
3.3. Standard Memory Protection CSRs 27
3.3.1. pmpcfg0…3 27
3.3.2. pmpaddr0…15 27
3.4. Standard M-mode Performance Counters 28
3.4.1. mcycle 28
3.4.2. mcycleh 28
3.4.3. minstret 28
3.4.4. minstreth 29
3.4.5. mhpmcounter3…31 29
3.4.6. mhpmcounter3…31h 29
3.4.7. mcountinhibit 29
3.4.8. mhpmevent3…31 29
3.5. Standard Trigger CSRs 29
3.5.1. tselect 29
3.5.2. tdata1…3 29
3.6. Standard Debug Mode CSRs 30
3.6.1. dcsr 30
3.6.2. dpc 31
3.6.3. dscratch0 31
3.6.4. dscratch1 31
3.7. Custom Debug Mode CSRs 31
3.7.1. dmdata0 31
3.8. Custom Interrupt Handling CSRs 32
3.8.1. meiea 32
3.8.2. meipa 32
3.8.3. meifa 33
3.8.4. meipra 34
3.8.5. meinext 34
3.8.6. meicontext 35
3.9. Custom Memory Protection CSRs 38
3.9.1. pmpcfgm0 38
3.10. Custom Power Control CSRs 38
3.10.1. msleep 38
4. Custom Extensions 40
4.1. Xh3irq: Hazard3 interrupt controller 40

4.2. Xh3pmpm: M-mode PMP regions . . . 40
4.3. Xh3power: Hazard3 power management . . . 40
4.3.1. h3.block . . . 41
4.3.2. h3.unblock . . . 41
4.4. Xh3bextm: Hazard3 bit extract multiple . . . 42
4.4.1. h3.bextm . . . 42
4.4.2. h3.bextmi . . . 43
5. Debug . . . 45
5.1. Debug Topologies . . . 45
5.2. Implementation-defined behaviour . . . 46
5.3. Debug Module to Core Interface . . . 48
Appendix A: Instruction Cycle Counts . . . 49
A.1. RV32I . . . 49
A.2. M Extension . . . 50
A.3. A Extension . . . 51
A.4. C Extension . . . 51
A.5. Privileged Instructions (including Zicsr) . . . 51
A.6. Bit Manipulation . . . 52
A.7. Zcb Extension . . . 53
A.8. Zcmp Extension . . . 53
A.9. Branch Predictor . . . 54

Chapter 1. Introduction

Hazard3 is a configurable 3-stage RISC-V processor, implementing:

- `RV32I`: 32-bit base instruction set
- `M`: integer multiply/divide/modulo
- `A`: atomic memory operations, with AHB5 global exclusives
- `C`: compressed instructions
- `Zicsr`: CSR access
- `Zba`: address generation
- `Zbb`: basic bit manipulation
- `Zbc`: carry-less multiplication
- `Zbs`: single-bit manipulation
- `Zbkb`: basic bit manipulation for scalar cryptography
- `Zcb`: basic additional compressed instructions
- `Zcmp`: push/pop and double-move compressed instructions
- Debug, Machine and User privilege/execution modes
- Privileged instructions `ECALL`, `EBREAK`, `MRET` and `WFI`
- External debug support
- Instruction address trigger unit (hardware breakpoints)

1.1. Architectural Overview

1.1.1. Pipeline Stages

The three stages are:

- `F`: Fetch
  - Contains the data phase for instruction fetch
  - Contains the instruction prefetch buffer
  - Predecodes register numbers `rs1`/`rs2`, for faster register file read and register bypass
  - Contains the address match logic for the optional branch predictor
- `X`: Execute
  - Decodes and execute instructions
  - Drives the address phase for load/store/AMO
  - Generates jump/branch addresses
  - Contains the read and write ports for the CSR file
  - Unbypassed register values are available at the beginning of stage `X`

  - The ALU result is valid by the end of stage `X`
- `M`: Memory
  - Contains the data phase for load/store/AMO
  - Generates exception addresses
  - Register writeback is at the end of stage `M`

The instruction fetch address phase is best thought of as residing in stage `X`. The 2-cycle feedback loop between jump/branch decode into address issue in stage `X`, and the fetch data phase in stage `F`, is what defines Hazard3's jump/branch performance.

This document often refers to `F`, `X` and `M` as stages 1, 2 and 3 respectively. This numbering is useful when describing dependencies between values held in different pipeline stages, as it makes the direction and distance of the dependency more apparent.

1.1.2. Bus Interfaces

Hazard3 implements either one or two AHB5 bus manager ports. Use the single-port configuration when ease of integration is a priority, since it supports simpler bus topologies. The dual-port configuration adds a dedicated port for instruction fetch. Use the dual-port configuration for maximum frequency and the best clock-for-clock performance.

Hazard3 uses AHB5 specifically, rather than older versions of the AHB standard, because of its support for global exclusives. This is a bus feature that allows a processor to perform an ordered read-modify-write sequence with a guarantee that no other processor has written to the same address range in between. Hazard3 uses this to implement multiprocessor support for the A (atomics) extension. Single-processor support for the A extension does not require these additional signals.

AHB5 is one of the two protocols described in the AMBA 5 AHB protocol specification. Its full name is (perhaps surprisingly) AMBA 5 AHB5. Refer to the protocol specification for more information about this standard bus protocol.

1.1.3. Multiply/Divide

For minimal M-extension support, as enabled by EXTENSION_M, Hazard3 instantiates a sequential multiply/divide circuit (restoring divide, naive repeated-addition multiply). Instructions stall in stage `X` until the multiply/divide completes. Optionally, the circuit can be unrolled by a small factor to produce multiple bits ber clock. A throughput of one, two or four bits per cycle is achievable in practice, with the internal logic delay becoming quite significant at four.

Set MUL_FAST to instantiate the single-cycle multiplier circuit. The fast multiplier returns results either to stage 3 or stage 2, depending on the MUL_FASTER parameter.

By default the single-cycle multiplier only supports 32-bit `mul`, which is by far the most common of the four multiply instructions. The remaining instructions still execute on the sequential multiply/divide circuit. Set the MULH_FAST parameter to add single-cycle support for the high-half instructions (`mulh`, `mulhu` and `mulhsu`), at the cost of additional logic delay and area.

The single-cycle multiplier is implemented as a simple `*` behavioural multiply, so that your tools can infer the best multiply circuit for your platform. For example, Yosys infers DSP tiles on iCE40 UP5k FPGAs. The multiplier is a self-contained module (in `hdl/arith/hazard3_mul_fast.v`), so you can replace its implementation if you know of a faster or lower-area method for your platform.

1.2. List of RISC-V Specifications

These are links to the ratified versions of the base instruction set and extensions implemented by Hazard3.

Extension	Specification
`RV32I` v2.1	Unprivileged ISA 20191213
`M` v2.0	Unprivileged ISA 20191213
`A` v2.1	Unprivileged ISA 20191213
`C` v2.0	Unprivileged ISA 20191213
`Zicsr` v2.0	Unprivileged ISA 20191213
`Zifencei` v2.0	Unprivileged ISA 20191213
`Zba` v1.0.0	Bit Manipulation ISA extensions 20210628
`Zbb` v1.0.0	Bit Manipulation ISA extensions 20210628
`Zbc` v1.0.0	Bit Manipulation ISA extensions 20210628
`Zbs` v1.0.0	Bit Manipulation ISA extensions 20210628
`Zbkb` v1.0.1	Scalar Cryptography ISA extensions 20220218
`Zcb` v1.0.3-1	Code Size Reduction extensions frozen v1.0.3-1
`Zcmp` v1.0.3-1	Code Size Reduction extensions frozen v1.0.3-1
Machine ISA v1.12	Privileged Architecture 20211203
Debug v0.13.2	RISC-V External Debug Support 20190322

Chapter 2. Configuration and Integration

2.1. Hazard3 Source Files

Hazard3's source is written in Verilog 2005, and is self-contained. It can be found here: github.com/Wren6991/Hazard3/blob/stable/hdl. The file hdl/hazard3.f is a list of all the source files required to instantiate Hazard3.

For more information on the Verilog 2005 language, refer to IEEE 1364-2005 (a PDF can be found online).

Files ending with `.vh` are preprocessor include files used by the Hazard3 source. The following two are particularly noteworthy:

- hazard3_config.vh: the main Hazard3 configuration header. Lists and describes Hazard3's global configuration parameters, such as ISA extension support
- hazard3_config_inst.vh: a file which propagates configuration parameters through module instantiations, all the way down from Hazard3's top-level modules through the internals

There are two ways to configure Hazard3 using these two files:

- Directly edit the parameter defaults in `hazard3_config.vh` in your local Hazard3 checkout (and then let the top-level parameters default when instantiating Hazard3)
- Set all configuration parameters in your Hazard3 instantiation, and let the parameters propagate down through the hierarchy

The latter method is recommended for mature projects because it supports multiple distinct configurations of Hazard3 in the same system (for instance, a high-performance applications core and a low-area control-plane core). You may find the former method more convenient for quick hacking on the configuration.

2.2. Top-level Modules

Hazard3 has two top-level modules:

- hazard3_cpu_1port
- hazard3_cpu_2port

These are both thin wrappers around the hazard3_core module. `hazard3_cpu_1port` has a single AHB5 bus port which is shared for instruction fetch, loads, stores and AMOs. `hazard3_cpu_2port` has two AHB5 bus ports, one for instruction fetch, and the other for loads, stores and AMOs. The 2-port wrapper has higher potential for performance, but the 1-port wrapper may be simpler to integrate, since there is no need to arbitrate multiple bus managers externally.

The core module `hazard3_core` can also be instantiated directly, which may be more efficient if support for some other bus standard is desired. However, the interface of `hazard3_core` will not be documented and is not guaranteed to be stable. By instantiating this module directly you are taking

on the risk that future Hazard3 releases may be incompatible with your integration.

2.3. FPGA Synthesis

Hazard3 supports FPGA synthesis using tools such as Yosys. You should set RESET_REGFILE to zero, as FPGA block RAMs and LUT RAMs often do not support reset, or are limited in the types of reset they support. Setting RESET_REGFILE to one is likely to result in the register file being implemented with logic fabric flops, which has a significant area and frequency impact.

You should synchronise the `rst_n` reset input externally. An example reset synchroniser is included in the example SoC file, but the details depend on your FPGA synthesis flow and your platform-level reset requirements.

It's recommended to tie `clk` and `clk_always_on` to the exact same clock net to conserve global buffer resources. Clock gating *is* supported on FPGA, but you must consult your toolchain documentation for the correct primitives or infererence techniques.

2.4. ASIC Synthesis

Hazard3 supports ASIC synthesis using common commercial tool flows. There are no particular requirements for configuration parameters, but your choice of configuration has an impact on area and frequency. Please raise an issue if you find a compatibility issue with your tools.

When applying the `clk_en` clock enable signal to the `clk` input in conjunction with the Xh3power extension, you must instantiate an external clock gate cell appropriate to your platform (such as an AND-and-latch type). Do not use a behavioural AND gate to gate the clock.

You must synchronise resets externally according to your STA constraints and your system-level reset strategy. Hazard3 uses an asynchronous active-low reset internally, but this can be adapted to other types by inserting an appropriate synchroniser in your core integration.

2.5. Interfaces (Top-level Ports)

Most ports are common to the two top-level wrappers, `hazard3_cpu_1port` and `hazard3_cpu_2port`. The only difference is the number of AHB5 manager ports used to access the bus: `hazard3_cpu_1port` has a single port used for all accesses, whereas `hazard3_cpu_2port` adds a separate, dedicated port for instruction fetch.

2.5.1. Interfaces Common to All Wrappers

Width	In/Out	Name	Description
Clock and reset inputs			

Width	In/Out	Name	Description
1	In	`clk`	Clock for all processor logic not driven by `clk_always_on`. Must be the same as the AHB5 bus clock. If the Xh3power extension is configured, you should instantiate an external clock gate on this clock, controlled by the `clk_en` output.
1	In	`clk_always_on`	Clock for logic required to wake from a low-power state. Connect to the same clock as `clk`, but do not insert an external clock gate.
1	In	`rst_n`	Active-low asynchronous reset for all processor logic. There is no internal synchroniser, so you must arrange externally for reset assertion/removal times to be met. For example, add an external reset synchroniser. When RESET_REGFILE is one, this input also resets the register file. You should avoid resetting the register file on FPGA, as this can prevent the register file being implemented with block RAM or LUT RAM primitives.
Power control signals			
These signals are used in the implementation of internal sleep states as configured by the msleep csr. They are used only when the Xh3power extension is enabled.			
1	Out	`pwrup_req`	Power-up request. Disconnect if Xh3power is not configured. Part of a four-phase (Gray code) req/ack handshake for negotiating power or clocks with your system power controller. The processor releases `pwrup_req` on entering a sufficiently deep `wfi` or `h3.block` state, as configured by the `msleep` CSR. It then waits for deassertion of `pwrup_ack`, before taking further action. The processor asserts `pwrup_req` when the processor intends to wake from the low-power state, and then waits for `pwrup_ack` before fetching the first instruction from the bus.

Width	In/Out	Name	Description
1	In	`pwrup_ack`	Power-up acknowledged. Tie back to `pwrup_req` if Xh3power is not configured, or if there is no external system power controller. The processor does not access the bus when either `pwrup_req` or `pwrup_ack` is low.
1	Out	`clk_en`	Control output for an external top-level clock gate on `clk`. Active-high enable. Hazard3 tolerates up to one cycle of delay between the assertion of `clk_en` and the resulting clock pulse on `clk`.
1	Out	`unblock_out`	Pulses high when an `h3.unblock` instruction executes. Disconnect if Xh3power is not configured.
1	In	`unblock_in`	A high input pulse will release a blocked `h3.block` instruction, or cause the next `h3.block` instruction to immediately fall through.
Debug Module controls			
All Debug Module signals should be connected to the signal with the matching name on the Hazard3 Debug Module implementation.			
1	In	`dbg_req_halt`	Debugger halt request. Tie low if debug support is not configured.
1	In	`dbg_req_halt_on_reset`	Debugger halt-on-reset request. Tie low if debug support is not configured.
1	In	`dbg_req_resume`	Debugger resume request. Tie low if debug support is not configured.
1	Out	`dbg_halted`	Debug halted status. Asserts when the processor is halted in Debug mode. Disconnect if debug support is not configured.
1	Out	`dbg_running`	Debug halted status. Asserts when the processor is not halted and not transitioning between halted/running states. Disconnect if debug support is not configured.
32	In	`dbg_data0_rdata`	Read data bus for mapping Debug Module `dmdata0` register as a CSR. Tie to zeroes if debug support is not configured.

Width	In/Out	Name	Description
32	Out	dbg_data0_wdata	Write data bus for mapping Debug Module dmdata0 register as a CSR. Disconnect if debug support is not configured.
1	Out	dbg_data0_wen	Write data strobe for mapping Debug Module dmdata0 register as a CSR. Disconnect if debug support is not configured.
32	In	dbg_instr_data	Instruction injection interface. Tie to zeroes if debug support is not configured.
1	In	dbg_instr_data_vld	Instruction injection interface. Tie low if debug support is not configured.
1	Out	dbg_instr_data_rdy	Instruction injection interface. Disconnect if debug support is not configured.
1	Out	dbg_instr_caught_exception	Exception caught during Program Buffer excecution. Disconnect if debug support is not configured.
1	Out	dbg_instr_caught_ebreak	Breakpoint instruction caught during Program Buffer execution. Disconnect if debug support is not configured.
Shared System Bus Access			
This subordinate bus port allows the standard System Bus Access (SBA) feature of the Debug Module to share bus access with the core. Alternatively, use the standalone hazard3_sbus_to_ahb adapter to provide dedicated SBA access to the system bus.			
32	In	dbg_sbus_addr	Address for System Bus Access arbitrated with this core’s load/store access. Tie to zeroes if this feature is not used.
1	In	dbg_sbus_write	Write/not-Read flag for System Bus Access arbitrated with this core’s load/store access. Tie low if this feature is not used.
2	In	dbg_sbus_size	Transfer size (0/1/2 = byte/halfword/word) for System Bus Access arbitrated with this core’s load/store access. Tie low if this feature is not used.
1	In	dbg_sbus_vld	Transfer enable signal for System Bus Access arbitrated with this core’s load/store access. Tie low if this feature is not used.
1	Out	dbg_sbus_rdy	Transfer stall signal for System Bus Access arbitrated with this core’s load/store access. Disconnect if this feature is not used.

Width	In/Out	Name	Description
1	Out	dbg_sbus_err	Bus fault signal for System Bus Access arbitrated with this core's load/store access. Disconnect if this feature is not used.
32	In	dbg_sbus_wdata	Write data bus for System Bus Access arbitrated with this core's load/store access. Tie to zeroes if this feature is not used.
32	Out	dbg_sbus_rdata	Read data bus for System Bus Access arbitrated with this core's load/store access. Disconnect if this feature is not used.
Interrupt requests			
NUM_IRQS	In	irq	If Xh3irq is not configured, this is the RISC-V external interrupt line (mip.meip) which you should connect to an external interrupt controller such as a standard RISC-V PLIC. If Xh3irq is configured, this is a vector of level-sensitive active-high system interrupt requests, which the core's internal interrupt controller can route through the mip.meip vector. Tie low if unused.
1	In	soft_irq	This is the standard RISC-V software interrupt signal, mip.msip. It should be connected to a register accessible to M-mode software on your system bus. Tie low if unused.
1	In	timer_irq	This is the standard RISC-V timer interrupt signal, mip.mtip. It should be connected to a standard RISC-V platform timer peripheral (mtime/mtimecmp) accessible to M-mode software on your system bus. Tie low if unused.

2.5.2. Interfaces for 1-port AHB5 CPU

This wrapper (hazard3_cpu_1port) adds a single standard AHB5 manager port. See the AMBA 5 AHB specification from Arm for definitions of these signals in the context of the bus protocol.

Width	In/Out	Name	Description
32	Out	haddr	Address output. AHB is always byte-addressed. Hazard3 always issues naturally-aligned accesses.

Width	In/Out	Name	Description
1	Out	`hwrite`	Driven high for a write transfer, low for a read transfer.
2	Out	`htrans`	Driven to `0` (`IDLE`) to indicate no transfer in the current address phase, and `2` (NSEQ) to indicate there is a transfer. Other types are not used.
3	Out	`hsize`	Driven to `0`, `1` or `2` to indicate byte, halfword or word sized transfers respectively. Other sizes are not used.
3	Out	`hburst`	Tied off to `0` (`SINGLE`). Hazard3 does not issue bursts.
4	Out	`hprot`	Bits `3:2` are always `0` to indicate nonbufferable and noncacheable access. Bit `1` (privileged) is `0` for U-mode access, and `1` for M-mode and Debug-mode access. Bit `0` is `0` for instruction fetch and `1` for data access (load/store or SBA).
1	Out	`hmastlock`	Hazard3 does not use legacy bus locking, so this bit is tied to 0.
8	Out	`hmaster`	8-bit manager ID. A value of `0x00` indicates access from the core (including Debug mode access via the Program Buffer), and `0x01` indicates an SBA access. (Non-SBA Debug mode load/store access can be detected by checking the `dbg_halted` status.)
1	Out	`hexcl`	Asserts high to indicate the current transfer is an Exclusive read/write as part of a read-modify-write sequence. This can be disconnected if you have not configured the A extension, or if you do not require global exclusive monitoring (for example in a single-core deployment).
1	In	`hready`	Negative stall signal. Assert low to indicate the current data phase continues on the next cycle.
1	In	`hresp`	Bus error signal. You *must* generate the complete two-phase AHB response as per the AHB5 specification.
1	In	`hexokay`	Exclusive transfer success. Hazard3 always queries the global monitor, so tie this input **high** if you do not implement global exclusive monitoring (for example in a single-core deployment). Similarly, ensure your global monitor returns a successful status for non-shared memory regions such as tightly-coupled memories.

Width	In/Out	Name	Description
32	Out	hwdata	Write data bus. The LSB of the bus is always aligned to a 4-byte boundary. Hazard3 drives the correct byte lanes depending on the transfer size and bits 1:0 of the address. Remaining byte lanes have undefined contents.
32	In	hrdata	Read data bus. The LSB of the bus is always aligned to a 4-byte boundary, so ensure you drive the correct byte lanes for narrow transfers.

2.5.3. Interfaces for 2-port AHB5 CPU

This wrapper (hazard3_cpu_2port) adds two standard AHB5 manager ports, with signals prefixed i_ for instruction and d_ for data. See the AMBA 5 AHB specification from Arm for definitions of these signals in the context of the bus protocol.

The I port only generates word-aligned word-sized read accesses. It does not use AHB5 exclusives.

When shared System Bus Access (SBA) is used, the SBA bus accesses are routed through the D port.

Port I (Instruction)			
Width	In/Out	Name	Description
32	Out	i_haddr	Address output. AHB is always byte-addressed. This port always issues word-aligned accesses (address bits 1:0 are zero).
1	Out	i_hwrite	Always driven low for to indicate a read transfer.
2	Out	i_htrans	Driven to 0 (IDLE) to indicate no transfer in the current address phase, and 2 (NSEQ) to indicate there is a transfer. Other types are not used.
3	Out	i_hsize	Always driven to 2 to indicate a word-sized transfer. Other sizes are not used.
3	Out	i_hburst	Tied off to 0 (SINGLE). Hazard3 does not issue bursts.
4	Out	i_hprot	Bits 3:2 are always 0 to indicate nonbufferable and noncacheable access. Bit 1 (privileged) is 0 for U-mode access, and 1 for M-mode and Debug-mode access. Bit 0 is tied to 0 to indicate instruction fetch.
1	Out	i_hmastlock	Hazard3 does not use legacy bus locking, so this bit is tied to 0.
8	Out	i_hmaster	8-bit manager ID. Tied to 0x00.

Port I (Instruction)			
1	In	i_hready	Negative stall signal. Assert low to indicate the current data phase continues on the next cycle.
1	In	i_hresp	Bus error signal. You **must** generate the complete two-phase AHB response as per the AHB5 specification.
32	Out	i_hwdata	Write data bus. Tied to all-zeroes as this port is read-only.
32	In	i_hrdata	Read data bus. Valid on cycles where i_hready is high during non-IDLE data phases.
Port D (Data)			
32	Out	d_haddr	Address output. AHB is always byte-addressed. Hazard3 always issues naturally-aligned accesses.
1	Out	d_hwrite	Driven high for a write transfer, low for a read transfer.
2	Out	d_htrans	Driven to 0 (IDLE) to indicate no transfer in the current address phase, and 2 (NSEQ) to indicate there is a transfer. Other types are not used.
3	Out	d_hsize	Driven to 0, 1 or 2 to indicate byte, halfword or word sized transfers respectively. Other sizes are not used.
3	Out	d_hburst	Tied off to 0 (SINGLE). Hazard3 does not issue bursts.
4	Out	d_hprot	Bits 3:2 are always 0 to indicate nonbufferable and noncacheable access. Bit 1 (privileged) is 0 for U-mode access, and 1 for M-mode access. Bit 0 is tied to 1 to indicate data access (load/store or SBA).
1	Out	d_hmastlock	Hazard3 does not use legacy bus locking, so this bit is tied to 0.
8	Out	d_hmaster	8-bit manager ID. A value of 0x00 indicates access from the core (including Debug mode access via the Program Buffer), and 0x01 indicates an SBA access. (Non-SBA Debug mode load/store access can be detected by checking the dbg_halted status.)
1	Out	d_hexcl	Asserts high to indicate the current transfer is an Exclusive read/write as part of a read-modify-write sequence. This can be disconnected if you have not configured the A extension, or if you do not require global exclusive monitoring (for example in a single-core deployment).

Port I (Instruction)			
1	In	`d_hready`	Negative stall signal. Assert low to indicate the current data phase continues on the next cycle.
1	In	`d_hresp`	Bus error signal. You *must* generate the complete two-phase AHB response as per the AHB5 specification.
1	In	`d_hexokay`	Exclusive transfer success. Hazard3 always queries the global monitor, so tie this input *high* if you do not implement global exclusive monitoring (for example in a single-core deployment). Similarly, ensure your global monitor returns a successful status for non-shared memory regions such as tightly-coupled memories.
32	Out	`d_hwdata`	Write data bus. The LSB of the bus is always aligned to a 4-byte boundary. Hazard3 drives the correct byte lanes depending on the transfer size and bits `1:0` of the address. Remaining byte lanes have undefined contents.
32	In	`d_hrdata`	Read data bus. The LSB of the bus is always aligned to a 4-byte boundary, so ensure you drive the correct byte lanes for narrow transfers.

2.6. Configuration Parameters

2.6.1. Reset state configuration

RESET_VECTOR

Address of the first instruction executed after Hazard3 comes out of reset.

Default value: all-zeroes.

MTVEC_INIT

Initial value of the machine trap vector base CSR (mtvec).

Bits clear in MTVEC_WMASK will never change from this initial value. Bits set in MTVEC_WMASK can be written/set/cleared as normal.

Default value: all-zeroes.

2.6.2. Standard RISC-V ISA support

EXTENSION_A

Support for the A extension: atomic read/modify/write. 0 for disable, 1 for enable.

Default value: 1

EXTENSION_C

Support for the C extension: compressed (variable-width). 0 for disable, 1 for enable.

Default value: 1

EXTENSION_M

Support for the M extension: hardware multiply/divide/modulo. 0 for disable, 1 for enable.

Default value: 1

EXTENSION_ZBA

Support for Zba address generation instructions. 0 for disable, 1 for enable.

Default value: 0

EXTENSION_ZBB

Support for Zbb basic bit manipulation instructions. 0 for disable, 1 for enable.

Default value: 0

EXTENSION_ZBC

Support for Zbc carry-less multiplication instructions. 0 for disable, 1 for enable.

Default value: 0

EXTENSION_ZBS

Support for Zbs single-bit manipulation instructions. 0 for disable, 1 for enable.

Default value: 0

EXTENSION_ZBKB

Support for Zbkb basic bit manipulation for cryptography.

Requires: EXTENSION_ZBB. (Since Zbb and Zbkb have a large overlap, this flag enables only those instructions which are in Zbkb but aren't in Zbb. Therefore both flags must be set for full Zbkb support.)

Default value: 0

EXTENSION_ZCB:

Support for Zcb basic additional compressed instructions

Requires: EXTENSION_C. (Some Zcb instructions also require Zbb or M, as they are 16-bit aliases of 32-bit instructions present in those extensions.)

Note Zca is equivalent to C, as we do not support the F extension.

Default value: 0

EXTENSION_ZCMP

Support for Zcmp push/pop and double-move instructions.

Requires: EXTENSION_C.

Note Zca is equivalent to C, as we do not support the F extension.

Default value: 0

EXTENSION_ZIFENCEI

Support for the fence.i instruction. When the branch predictor is not present, this instruction is optional, since a plain branch/jump is sufficient to flush the instruction prefetch queue. When the branch predictor is enabled (BRANCH_PREDICTOR is 1), this instruction must be implemented.

Default value: 0

2.6.3. Custom Hazard3 Extensions

EXTENSION_XH3BEXTM

Custom bit manipulation instructions for Hazard3: `h3.bextm` and `h3.bextmi`. See Xh3bextm: Hazard3 bit extract multiple.

Default value: 0

EXTENSION_XH3IRQ

Custom preemptive, prioritised interrupt support. Can be disabled if an external interrupt controller (e.g. PLIC) is used. If disabled, and NUM_IRQS > 1, the external interrupts are simply OR'd into mip.meip. See Xh3irq: Hazard3 interrupt controller.

Default value: 0

EXTENSION_XH3PMPM

Custom PMPCFGMx CSRs to enforce PMP regions in M-mode without locking. See Xh3pmpm: M-mode PMP regions.

Default value: 0

EXTENSION_XH3POWER

Custom power management controls for Hazard3. This adds the msleep CSR, and the `h3.block` and `h3.unblock` hint instructions. See Xh3power: Hazard3 power management

Default value: 0

2.6.4. CSR support

NOTE	the Zicsr extension is implied by any of CSR_M_MANDATORY, CSR_M_TRAP, CSR_COUNTER.

CSR_M_MANDATORY

Bare minimum CSR support e.g. misa. This flag is an absolute requirement for compliance with the RISC-V privileged specification. However, the privileged specification itself is an optional extension. Hazard3 allows the mandatory CSRs to be disabled to save a small amount of area in deeply-embedded implementations.

Default value: 1

CSR_M_TRAP

Include M-mode trap-handling CSRs, and enable trap support.

Default value: 1

CSR_COUNTER

Include the basic performance counters (`cycle`/`instret`) and relevant CSRs. Note that these performance counters are now in their own separate extension (Zicntr) and are no longer mandatory.

Default value: 0

U_MODE

Support the U (user) privilege level. In U-mode, the core performs unprivileged bus accesses, and software's access to CSRs is restricted. Additionally, if the PMP is included, the core may restrict U-mode software's access to memory.

Requires: CSR_M_TRAP.

Default value: 0

PMP_REGIONS

Number of physical memory protection regions, or 0 for no PMP. PMP is more useful if U-mode is supported, but this is not a requirement.

Hazard3's PMP supports only the NAPOT and(if PMP_GRAIN is 0) NA4 region types.

Requires: CSR_M_TRAP.

Default value: 0

PMP_GRAIN

This is the *G* parameter in the privileged spec, which defines the granularity of PMP regions. Minimum PMP region size is 1 << (*G* + 2) bytes.

If *G* > 0, `pmcfg.a` can not be set to NA4 (attempting to do so will set the region to OFF instead).

If *G* > 1, the *G* - 1 LSBs of pmpaddr are read-only-0 when `pmpcfg.a` is OFF, and read-only-1 when `pmpcfg.a` is NAPOT.

Default value: 0

PMP_HARDWIRED

PMPADDR_HARDWIRED: If a bit is 1, the corresponding region's pmpaddr and pmpcfg registers are read-only, with their values fixed when the processor is instantiated. PMP_GRAIN is ignored on hardwired regions.

Hardwired regions are far cheaper, both in area and comparison delay, than dynamically configurable regions.

Hardwired PMP regions are a good option for setting default U-mode permissions on regions which have access controls outside of the processor, such as peripheral regions. For this case it's recommended to make hardwired regions the highest-numbered, so they can be overridden by lower-numbered dynamic regions.

Default value: all-zeroes.

PMP_HARDWIRED_ADDR

Values of pmpaddr registers whose PMP_HARDWIRED bits are set to 1. Has no effect on PMP regions which are not hardwired.

Default value: all-zeroes.

PMP_HARDWIRED_CFG

Values of pmpcfg registers whose PMP_HARDWIRED bits are set to 1. Has no effect on PMP regions which are not hardwired.

Default value: all-zeroes.

DEBUG_SUPPORT

Support for run/halt and instruction injection from an external Debug Module, support for Debug Mode, and Debug Mode CSRs.

Requires: CSR_M_MANDATORY, CSR_M_TRAP.

Default value: 0

BREAKPOINT_TRIGGERS

Number of hardware breakpoints. A breakpoint is implemented as a trigger that supports only exact execution address matches, ignoring instruction size. That is, a trigger which supports type=2 execute=1 (but not store/load=1, i.e. not a watchpoint).

Requires: DEBUG_SUPPORT

Default value: 0

2.6.5. External interrupt support

NUM_IRQS

NUM_IRQS: Number of external IRQs. Minimum 1, maximum 512. Note that if EXTENSION_XH3IRQ (Hazard3 interrupt controller) is disabled then multiple external interrupts are simply OR'd into mip.meip.

Default value: 1

IRQ_PRIORITY_BITS

IRQ_PRIORITY_BITS: Number of priority bits implemented for each interrupt in meipra, if EXTENSION_XH3IRQ is enabled. The number of distinct levels is (1 << IRQ_PRIORITY_BITS). Minimum 0, max 4. Note that multiple priority levels with a large number of IRQs will have a severe effect on timing.

Default value: 0

IRQ_INPUT_BYPASS

Disable the input registers on the external interrupts, to reduce latency by one cycle. Can be applied on an IRQ-by-IRQ basis.

Ignored if EXTENSION_XH3IRQ is disabled.

Default value: all-zeroes (not bypassed).

2.6.6. Identification Registers

MVENDORID_VAL

Value of the mvendorid CSR. JEDEC JEP106-compliant vendor ID, or all-zeroes. 31:7 is continuation code count, 6:0 is ID. Parity bit is not stored.

Default value: all-zeroes.

MIMPID_VAL

Value of the mimpid CSR. Implementation ID for this specific version of Hazard3. Should be a git hash, or all-zeroes.

Default value: all-zeroes.

MHARTID_VAL

Value of the mhartid CSR. Each Hazard3 core has a single hardware thread. Multiple cores should have unique IDs.

Default value: all-zeroes.

MCONFIGPTR_VAL

Value of the mconfigptr CSR. Pointer to configuration structure blob, or all-zeroes. Must be at least 4-byte-aligned.

Default value: all-zeroes.

2.6.7. Performance/size options

REDUCED_BYPASS

Remove all forwarding paths except X→X (so back-to-back ALU ops can still run at 1 CPI), to save area. This has a significant impact on per-clock performance, so should only be considered for extremely low-area implementations.

Default value: 0

MULDIV_UNROLL

Bits per clock for multiply/divide circuit, if present. Must be a power of 2.

Default value: 1

MUL_FAST

Use single-cycle multiply circuit for MUL instructions, retiring to stage 3. The sequential multiply/divide circuit is still used for MULH*

Default value: 0

MUL_FASTER

Retire fast multiply results to stage 2 instead of stage 3. Throughput is the same, but latency is reduced from 2 cycles to 1 cycle.

Requires: MUL_FAST.

Default value: 0

MULH_FAST

Extend the fast multiply circuit to also cover MULH*, and remove the multiply functionality from the sequential multiply/divide circuit.

Requires: MUL_FAST

Default value: 0

FAST_BRANCHCMP

Instantiate a separate comparator (eq/lt/ltu) for branch comparisons, rather than using the ALU. Improves fetch address delay, especially if `Zba` extension is enabled. Disabling may save area.

Default value: 1

RESET_REGFILE

Whether to support reset of the general purpose registers. There are around 1k bits in the register file, so the reset can be disabled e.g. to permit block-RAM inference on FPGA.

Default value: 1

BRANCH_PREDICTOR

Enable branch prediction. The branch predictor consists of a single BTB entry which is allocated on a taken backward branch, and cleared on a mispredicted nontaken branch, a fence.i or a trap. Successful prediction eliminates the 1-cyle fetch bubble on a taken branch, usually making tight loops faster.

Requires: EXTENSION_ZIFENCEI

Default value: 0

MTVEC_WMASK

MTVEC_WMASK: Mask of which bits in mtvec are writable. Full writability (except for bit 1) is recommended, because a common idiom in setup code is to set mtvec just past code that may trap, as a hardware `try {…} catch` block.

- The vectoring mode can be made fixed by clearing the LSB of MTVEC_WMASK
- In vectored mode, the vector table must be aligned to its size, rounded up to a power of two.

Default: All writable except for bit 1.

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 any custom extension is enabled (Custom Hazard3 Extensions), otherwise 0.
20	`u`	1 if User mode is supported, otherwise 0.
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
21	`tw`	Timeout wait. Only present if U-mode is supported. When 1, attempting to execute a WFI instruction in U-mode will instantly cause an illegal instruction exception.
17	`mprv`	Modify privilege. Only present if U-mode is supported. If 1, loads and stores behave as though the current privilege level were `mpp`. This includes physical memory protection checks, and the privilege level asserted on the system bus alongside the load/store address.
12:11	`mpp`	Previous privilege level. If U-mode is supported, this register can store the values 3 (M-mode) or 0 (U-mode). Otherwise, only 3 (M-mode). If another value is written, hardware rounds to the nearest supported 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 neither U-mode traps nor S-mode are supported. Access will cause an illegal instruction exception.

3.2.4. mideleg

Address: `0x303`

Unimplemented, as neither U-mode traps nor S-mode are 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 meiea.
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 meiea, 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 meipa) and enabled in meiea.
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.

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 31:2 of `mepc` are capable of holding all 30-bit values. Bit 1 is writable only if the C extension is implemented, and is otherwise hardwired to 0. 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
0	Instruction address misaligned
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

3.2.11. mtval

Address: `0x343`

Hardwired to 0.

3.2.12. mcounteren

Address: `0x306`

Counter enable. Control access to counters from U-mode. Not to be confused with mcountinhibit.

This register only exists if U-mode is supported.

Bits	Name	Description
31:3	**-**	RES0

Bits	Name	Description
2	`ir`	If 1, U-mode is permitted to access the `instret`/`instreth` instruction retire counter CSRs. Otherwise, U-mode accesses to these CSRs will trap.
1	`tm`	No hardware effect, as the `time`/`timeh` CSRs are not implemented. However, this field still exists, as M-mode software can use it to track whether it should emulate U-mode attempts to access those CSRs.
0	`cy`	If 1, U-mode is permitted to access the `cycle`/`cycleh` cycle counter CSRs. Otherwise, U-mode accesses to these CSRs will trap.

3.3. Standard Memory Protection CSRs

3.3.1. pmpcfg0…3

Address: `0x3a0` through `0x3a3`

Configuration registers for up to 16 physical memory protection regions. Only present if PMP support is configured. If so, all 4 registers are present, but some registers may be partially/completely hardwired depending on the number of PMP regions present.

By default, M-mode has full permissions (RWX) on all of memory, and U-mode has no permissions. A PMP region can be configured to alter this default within some range of addresses. For every memory location executed, loaded or stored, the processor looks up the *lowest active region* that overlaps that memory location, and applies its permissions to determine whether this access is allowed. The full description can be found in the RISC-V privileged ISA manual.

Each `pmpcfg` register divides into four identical 8-bit chunks, each corresponding to one region, and laid out as below:

Bits	Name	Description
7	`L`	Lock region, and additionally enforce its permissions on M-mode as well as U-mode.
6:5	-	RES0
4:3	`A`	Address-matching mode. Values supported are 0 (OFF), 2 (NA4, naturally aligned 4-byte) and 3 (NAPOT, naturally aligned power-of-two). Attempting to write an unsupported value will set the region to OFF.
2	`X`	Execute permission
1	`W`	Write permission
0	`R`	Read permission

3.3.2. pmpaddr0…15

Address: `0x3b0` through `0x3bf`

Address registers for up to 16 physical memory protection regions. Only present if PMP support is configured. If so, all 16 registers are present, but some may fully/partially hardwired.

`pmpaddr` registers express addresses in units of 4 bytes, so on Hazard3 (a 32-bit processor with no virtual address support) only the lower 30 bits of each address register are implemented.

The interpretation of the `pmpaddr` bits depends on the `A` mode configured in the corresponding `pmpcfg` register field:

- For NA4, the entire 30-bit PMP address is matched against the 30 MSBs of the checked address.
- For NAPOT, `pmpaddr` bits up to and including the least-significant zero bit are ignored, and the remaining bits are matched against the MSBs of the checked address.

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`. The RISC-V spec says this about `mcycle`: "Any CSR write takes effect after the writing instruction has otherwise completed."

3.4.2. mcycleh

Address: `0xb80`

Upper half of the 64-bit cycle counter. Readable and writable by software. Increments on cycles where `mcycle` has the value `0xffffffff`, unless `mcountinhibit.cy` is 1, or the processor is in Debug Mode.

This includes when `mcycle` is written on that same cycle, since RISC-V specifies the CSR write takes place *after* the increment for that cycle.

3.4.3. minstret

Address: `0xb02`

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

If some value `n` is written to `minstret`, and it is read back by the very next instruction, the value read will be exactly `n`. This is because the CSR write logically takes place after the instruction has otherwise completed.

3.4.4. minstreth

Address: `0xb82`

Upper half of the 64-bit instruction retire counter. Readable and writable by software. Increments when the core retires an instruction and the value of `minstret` is `0xffffffff`, unless `mcountinhibit.ir` is 1, or the processor is in Debug Mode.

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`. Resets to 1.
0	`cy`	When 1, inhibit counting of `mcycle`/`mcycleh`. Resets to 1.

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 executed in M-mode will break to Debug Mode instead of trapping
12	`ebreaku`	When 1, `ebreak` instructions executed in U-mode will break to Debug Mode instead of trapping. Hardwired to 0 if U-mode is not supported.
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`	Read the privilege state the core was in when it entered Debug Mode, and set the privilege state it will be in when it exits Debug Mode. If U-mode is implemented, the values 3 and 0 are supported. Otherwise hardwired to 3.

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.

Cause	Description
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 Debug Mode CSRs

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.

3.8. Custom Interrupt Handling CSRs

3.8.1. meiea

Address: `0xbe0`

External interrupt enable array. Contains a read-write bit for each external interrupt request: a `1` bit indicates that interrupt is currently enabled. At reset, all external interrupts are disabled.

If enabled, an external interrupt can cause assertion of the standard RISC-V machine external interrupt pending flag (`mip.meip`), and therefore cause the processor to enter the external interrupt vector. See meipa.

There are up to 512 external interrupts. The upper half of this register contains a 16-bit window into the full 512-bit vector. The window is indexed by the 5 LSBs of the write data. For example:

```
csrrs a0, meiea, a0 // Read IRQ enables from the window selected by a0
csrw meiea, a0      // Write a0[31:16] to the window selected by a0[4:0]
csrr a0, meiea      // Read from window 0 (edge case)
```

The purpose of this scheme is to allow software to *index* an array of interrupt enables (something not usually possible in the CSR space) without introducing a stateful CSR index register which may have to be saved/restored around IRQs.

Bits	Name	Description
31:16	`window`	16-bit read/write window into the external interrupt enable array
15:5	-	RES0
4:0	`index`	Write-only self-clearing field (no value is stored) used to control which window of the array appears in `window`.

3.8.2. meipa

Address: `0xbe1`

External interrupt pending array. Contains a read-only bit for each external interrupt request. Similarly to `meiea`, this register is a window into an array of up to 512 external interrupt flags. The status appears in the upper 16 bits of the value read from `meipa`, and the lower 5 bits of the value *written* by the same CSR instruction (or 0 if no write takes place) select a 16-bit window of the full interrupt pending array.

A `1` bit indicates that interrupt is currently asserted. IRQs are assumed to be level-sensitive, and the relevant `meipa` bit is cleared by servicing the requestor so that it deasserts its interrupt request.

When any interrupt of sufficient priority is both set in `meipa` and enabled in `meiea`, the standard RISC-V external interrupt pending bit `mip.meip` is asserted. In other words, `meipa` is filtered by `meiea` 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 `meipa`
- The matching interrupt enable bit is set in `meiea`
- The interrupt priority is greater than or equal to the preemption priority in `meicontext`
- 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)

Bits	Name	Description
31:16	`window`	16-bit read-only window into the external interrupt pending array
15:5	-	RES0
4:0	`index`	Write-only, self-clearing field (no value is stored) used to control which window of the array appears in `window`.

3.8.3. meifa

Address: `0xbe2`

External interrupt force array. Contains a read-write bit for every interrupt request. Writing a 1 to a bit in the interrupt force array causes the corresponding bit to become pending in `meipa`. Software can use this feature to manually trigger a particular interrupt.

There are no restrictions on using `meifa` inside of an interrupt. The more useful case here is to schedule some lower-priority handler from within a high-priority interrupt, so that it will execute before the core returns to the foreground code. Implementers may wish to reserve some external IRQs with their external inputs tied to 0 for this purpose.

Bits can be cleared by software, and are cleared automatically by hardware upon a read of `meinext` which returns the corresponding IRQ number in `meinext.irq` (no matter whether `meinext.update` is written).

`meifa` implements the same array window indexing scheme as `meiea` and `meipa`.

Bits	Name	Description
31:16	`window`	16-bit read/write window into the external interrupt force array
15:5	-	RES0

Bits	Name	Description
4:0	`index`	Write-only, self-clearing field (no value is stored) used to control which window of the array appears in `window`.

3.8.4. meipra

Address: `0xbe3`

External interrupt priority array. Each interrupt has an (up to) 4-bit priority value associated with it, and each access to this register reads and/or writes a 16-bit window containing four such priority values. When less than 16 priority levels are available, the LSBs of the priority fields are hardwired to 0.

When an interrupt's priority is lower than the current preemption priority `meicontext.preempt`, it is treated as not being pending. The pending bit in `meipa` will still assert, but the machine external interrupt pending bit `mip.meip` will not, so the processor will ignore this interrupt. See meicontext.

Bits	Name	Description
31:16	`window`	16-bit read/write window into the external interrupt priority array, containing four 4-bit priority values.
15:7	-	RES0
6:0	`index`	Write-only, self-clearing field (no value is stored) used to control which window of the array appears in `window`.

3.8.5. meinext

Address: `0xbe4`

Get next interrupt. Contains the index of the highest-priority external interrupt which is both asserted in `meipa` and enabled in `meiea`, left-shifted by 2 so that it can be used to index an array of 32-bit function pointers. If there is no such interrupt, the MSB is set.

When multiple interrupts of the same priority are both pending and enabled, the lowest-numbered wins. Interrupts with priority less than `meicontext.ppreempt` — the *previous* preemption priority — are treated as though they are not pending. This is to ensure that a preempting interrupt frame does not service interrupts which may be in progress in the frame that was preempted.

Bits	Name	Description
31	`noirq`	Set when there is no external interrupt which is enabled, pending, and has sufficient priority. Can be efficiently tested with a `bltz` or `bgez` instruction.
30:11	-	RES0
10:2	`irq`	Index of the highest-priority active external interrupt. Zero when no external interrupts with sufficient priority are both pending and enabled.

Bits	Name	Description
1	-	RES0
0	`update`	Writing 1 (self-clearing) causes hardware to update `meicontext` according to the IRQ number and preemption priority of the interrupt indicated in `noirq`/`irq`. This should be done in a single atomic operation, i.e. `csrrsi a0, meinext, 0x1`.

3.8.6. meicontext

Address: `0xbe5`

External interrupt context register. Configures the priority level for interrupt preemption, and helps software track which interrupt it is currently in. The latter is useful when a common interrupt service routine handles interrupt requests from multiple instances of the same peripheral.

A three-level stack of preemption priorities is maintained in the `preempt`, `ppreempt` and `pppreempt` fields. The priority stack is saved when hardware enters the external interrupt vector, and restored by an `mret` instruction if `meicontext.mreteirq` is set.

The top entry of the priority stack, `preempt`, is used by hardware to ensure that only higher-priority interrupts can preempt the current interrupt. The next entry, `ppreempt`, is used to avoid servicing interrupts which may already be in progress in a frame that was preempted. The third entry, `pppreempt`, has no hardware effect, but ensures that `preempt` and `ppreempt` can be correctly saved/restored across arbitary levels of preemption.

Bits	Name	Description
31:28	`pppreempt`	Previous `ppreempt`. Set to `ppreempt` on priority save, set to zero on priority restore. Has no hardware effect, but ensures that when `meicontext` is saved/restored correctly, `preempt` and `ppreempt` stack correctly through arbitrarily many preemption frames.
27:24	`ppreempt`	Previous `preempt`. Set to `preempt` on priority save, restored to to `pppreempt` on priority restore. IRQs of lower priority than `ppreempt` are not visible in `meinext`, so that a preemptee is not re-taken in the preempting frame.
23:21	-	RES0

Bits	Name	Description
20:16	preempt	Minimum interrupt priority to preempt the current interrupt. Interrupts with lower priority than preempt do not cause the core to transfer to an interrupt handler. Updated by hardware when when meinext.update is written, or when hardware enters the external interrupt vector. If an interrupt is present in meinext, then preempt is set to one level greater than that interrupt's priority. Otherwise, ppreempt is set to one level greater than the maximum interrupt priority, disabling preemption.
15	noirq	Not in interrupt (read/write). Set to 1 at reset. Set to meinext.noirq when meinext.update is written. No hardware effect.
14:13	-	RES0
12:4	irq	Current IRQ number (read/write). Set to meinext.irq when meinext.update is written.
3	mtiesave	Reads as the current value of mie.mtie, if clearts is set. Otherwise reads as 0. Writes are ORed into mie.mtie.
2	msiesave	Reads as the current value of mie.msie, if clearts is set. Otherwise reads as 0. Writes are ORed into mie.msie.
1	clearts	Write-1 self-clearing field. Writing 1 will clear mie.mtie and mie.msie, and present their prior values in the mtiesave and msiesave of this register. This makes it safe to re-enable IRQs (via mstatus.mie) without the possibility of being preempted by the standard timer and soft interrupt handlers, which may not be aware of Hazard3's interrupt hardware. The clear due to clearts takes precedence over the set due to mtiesave/msiesave, although it would be unusual for software to write both on the same cycle.
0	mreteirq	Enable restore of the preemption priority stack on mret. This bit is set on entering the external interrupt vector, cleared by mret, and cleared upon taking any trap other than an external interrupt. Provided meicontext is saved on entry to the external interrupt vector (before enabling preemption), is restored before exiting, and the standard software/timer IRQs are prevented from preempting (e.g. by using clearts), this flag allows the hardware to safely manage the preemption priority stack even when an external interrupt handler may take exceptions.

The following is an example of an external interrupt vector (mip.meip) which implements nested, prioritised interrupt dispatch using meicontext and meinext:

```
isr_external_irq:
    // Save caller saves and exception return state whilst IRQs are disabled.
    // We can't be pre-empted during this time, but if a higher-priority IRQ
    // arrives ("late arrival"), that will be the one displayed in meinext.
    addi sp, sp, -80
    sw ra,  0(sp)
    ... snip
    sw t6, 60(sp)

    csrr a0, mepc
    sw a0, 64(sp)
    // Set bit 1 when reading to clear+save mie.mtie and mie.msie
    csrrsi a0, meicontext, 0x2
    sw a0, 68(sp)
    csrr a0, mstatus
    sw a0, 72(sp)

    j get_next_irq

dispatch_irq:
    // Preemption priority was configured by meinext update, so enable preemption:
    csrsi mstatus, 0x8
    // meinext is pre-shifted by 2, so only an add is required to index table
    la a1, _external_irq_table
    add a1, a1, a0
    jalr ra, a1

    // Disable IRQs on returning so we can sample the next IRQ
    csrci mstatus, 0x8

get_next_irq:
    // Sample the current highest-priority active IRQ (left-shifted by 2) from
    // meinext, and write 1 to the LSB to tell hardware to tell hw to update
    // meicontext with the preemption priority (and IRQ number) of this IRQ
    csrrsi a0, meinext, 0x1
    // MSB will be set if there is no active IRQ at the current priority level
    bgez a0, dispatch_irq

no_more_irqs:
    // Restore saved context and return from handler
    lw a0, 64(sp)
    csrw mepc, a0
    lw a0, 68(sp)
    csrw meicontext, a0
    lw a0, 72(sp)
    csrw mstatus, a0

    lw ra,  0(sp)
    ... snip
    lw t6, 60(sp)
```

```
    addi sp, sp, 80
    mret
```

3.9. Custom Memory Protection CSRs

3.9.1. pmpcfgm0

Address: 0xbd0

PMP M-mode configuration. One bit per PMP region. Setting a bit makes the corresponding region apply to M-mode (like the `pmpcfg.L` bit) but does not lock the region.

PMP is useful for non-security-related purposes, such as stack guarding and peripheral emulation. This extension allows M-mode to freely use any currently unlocked regions for its own purposes, without the inconvenience of having to lock them.

Note that this does not grant any new capabilities to M-mode, since in the base standard it is already possible to apply unlocked regions to M-mode by locking them. In general, PMP regions should be locked in ascending region number order so they can't be subsequently overridden by currently unlocked regions.

Note also that this is not the same as the "rule locking bypass" bit in the ePMP extension, which does not permit locked and unlocked M-mode regions to coexist.

Bits	Name	Description
31:16	-	RES0
15:0	`m`	Regions apply to M-mode if this bit *or* the corresponding `pmpcfg.L` bit is set. Regions are locked if and only if the corresponding `pmpcfg.L` bit is set.

3.10. Custom Power Control CSRs

3.10.1. msleep

Address: `0xbf0`

M-mode sleep control register. Resets to all-zeroes.

Bits	Name	Description
31:3	-	RES0
2	`sleeponblock`	Enter the deep sleep state on a `h3.block` instruction as well as a standard `wfi`. If this bit is clear, a `h3.block` is always implemented as a simple pipeline stall.

Bits	Name	Description
1	`powerdown`	Release the external power request when going to sleep. The function of this is platform-defined — it may do nothing, it may do something simple like clock-gating the fabric, or it may be tied to some complex system-level power controller. When waking, the processor reasserts its external power-up request, and will not fetch any instructions until the request is acknowledged. This may add considerable latency to the wakeup.
0	`deepsleep`	Deassert the processor clock enable when entering the sleep state. If a clock gate is instantiated, this allows most of the processor (everything except the power state machine and the interrupt and halt input registers) to be clock gated whilst asleep, which may reduce the sleep current. This adds one cycle to the wakeup latency.

Chapter 4. Custom Extensions

Hazard3 implements a small number of custom extensions. All are optional: custom extensions are only included if the relevant feature flags are set to 1 when instantiating the processor (Configuration Parameters). Hazard3 is always a *conforming* RISC-V implementation, and when these extensions are disabled it is also a *standard* RISC-V implementation.

If any one of these extensions is enabled, the `x` bit in misa is set to indicate the presence of a nonstandard extension.

4.1. Xh3irq: Hazard3 interrupt controller

This is a lightweight extension to control up to 512 external interrupts, with up to 16 levels of preemption.

This extension does not add any instructions, but does add several CSRs:

- meiea
- meipa
- meifa
- meipra
- meinext
- meicontext

If this extension is disabled then Hazard3 supports a single external interrupt input (or multiple inputs that it simply ORs together in an uncontrolled fashion), so an external PLIC can be used for standard interrupt support.

Note that, besides the additional CSRs, this extension is effectively a slightly more complicated way of driving the standard `mip.meip` flag (mip). The RISC-V trap handling CSRs themselves are always completely standard.

4.2. Xh3pmpm: M-mode PMP regions

This extension adds a new M-mode CSR, pmpcfgm0, which allows a PMP region to be enforced in M-mode without locking the region.

This is useful when the PMP is used for non-security-related purposes such as stack guarding, or trapping and emulation of peripheral accesses.

4.3. Xh3power: Hazard3 power management

This extension adds a new M-mode CSR (msleep), and two new hint instructions, `h3.block` and `h3.unblock`, in the `slt` nop-compatible custom hint space.

The `msleep` CSR controls how deeply the processor sleeps in the WFI sleep state. By default, a WFI is

implemented as a normal pipeline stall. By configuring `msleep` appropriately, the processor can gate its own clock when asleep or, with a simple 4-phase req/ack handshake, negotiate power up/down of external hardware with an external power controller. These options can improve the sleep current at the cost of greater wakeup latency.

The hints allow processors to sleep until woken by other processors in a multiprocessor environment. They are implemented on top of the standard WFI state, which means they interact in the same way with external debug, and benefit from the same deep sleep states in `msleep`.

4.3.1. h3.block

Enter a WFI sleep state until either an unblock signal is received, or an interrupt is asserted that would cause a WFI to exit.

If `mstatus.tw` is set, attempting to execute this instruction in privilege modes lower than M-mode will generate an illegal instruction exception.

If an unblock signal has been received in the time since the last `h3.block`, this instruction executes as a `nop`, and the processor does not enter the sleep state. Conceptually, the sleep state falls through immediately because the corresponding unblock signal has already been received.

An unblock signal is received when a neighbouring processor (the exact definition of "neighbouring" being left to the implementor) executes an `h3.unblock` instruction, or for some other platform-defined reason.

This instruction is encoded as `slt x0, x0, x0`, which is part of the custom nop-compatible hint encoding space.

Example C macro:

```
#define __h3_block() asm ("slt x0, x0, x0")
```

Example assembly macro:

```
.macro h3.block
    slt x0, x0, x0
.endm
```

4.3.2. h3.unblock

Post an unblock signal to other processors in the system. For example, to notify another processor that a work queue is now nonempty.

If `mstatus.tw` is set, attempting to execute this instruction in privilege modes lower than M-mode will generate an illegal instruction exception.

This instruction is encoded as `slt x0, x0, x1`, which is part of the custom nop-compatible hint encoding space.

Example C macro:

```
#define __h3_unblock() asm ("slt x0, x0, x1")
```

Example assembly macro:

```
.macro h3.unblock
    slt x0, x0, x1
.endm
```

4.4. Xh3bextm: Hazard3 bit extract multiple

This is a small extension with multi-bit versions of the "bit extract" instructions from Zbs, used for extracting small, contiguous bit fields.

4.4.1. h3.bextm

"Bit extract multiple", a multi-bit version of the `bext` instruction from Zbs. Perform a right-shift followed by a mask of 1-8 LSBs.

Encoding (R-type):

Bits	Name	Value	Description
31:29	`funct7[6:4]`	`0b000`	RES0
28:26	`size`	-	Number of ones in mask, values 0→7 encode 1→8 bits.
25	`funct7[0]`	`0b0`	RES0, because aligns with `shamt[5]` of potential RV64 version of `h3.bextmi`
24:20	`rs2`	-	Source register 2 (shift amount)
19:15	`rs1`	-	Source register 1
14:12	`funct3`	`0b000`	`h3.bextm`
11:7	`rd`	-	Destination register
6:2	`opc`	`0b01011`	custom0 opcode
1:0	`size`	`0b11`	32-bit instruction

Example C macro (using GCC statement expressions):

```
// nbits must be a constant expression
#define __h3_bextm(nbits, rs1, rs2) ({\
    uint32_t __h3_bextm_rd; \
    asm (".insn r 0x0b, 0, %3, %0, %1, %2"\
        : "=r" (__h3_bextm_rd) \
```

```
        : "r" (rs1), "r" (rs2), "i" ((((nbits) - 1) & 0x7) << 1)\
    ); \
    __h3_bextm_rd; \
})
```

Example assembly macro:

```
// rd = (rs1 >> rs2[4:0]) & ~(-1 << nbits)
.macro h3.bextm rd rs1 rs2 nbits
.if (\nbits < 1) || (\nbits > 8)
.err
.endif
#if NO_HAZARD3_CUSTOM
    srl  \rd, \rs1, \rs2
    andi \rd, \rd, ((1 << \nbits) - 1)
#else
.insn r 0x0b, 0x0, (((\nbits - 1) & 0x7 ) << 1), \rd, \rs1, \rs2
#endif
.endm
```

4.4.2. h3.bextmi

Immediate variant of `h3.bextm`.

Encoding (I-type):

Bits	Name	Value	Description
31:29	`imm[11:9]`	`0b000`	RES0
28:26	`size`	-	Number of ones in mask, values 0→7 encode 1→8 bits.
25	`imm[5]`	`0b0`	RES0, for potential future RV64 version
24:20	`shamt`	-	Shift amount, 0 through 31
19:15	`rs1`	-	Source register 1
14:12	`funct3`	`0b100`	`h3.bextmi`
11:7	`rd`	-	Destination register
6:2	`opc`	`0b01011`	custom0 opcode
1:0	`size`	`0b11`	32-bit instruction

Example C macro (using GCC statement expressions):

```
// nbits and shamt must be constant expressions
#define __h3_bextmi(nbits, rs1, shamt) ({\
    uint32_t __h3_bextmi_rd; \
    asm (".insn i 0x0b, 0x4, %0, %1, %2"\
```

```
        : "=r" (__h3_bextmi_rd) \
        : "r" (rs1), "i" ((((nbits) - 1) & 0x7) << 6 | ((shamt) & 0x1f)) \
    ); \
    __h3_bextmi_rd; \
})
```

Example assembly macro:

```
// rd = (rs1 >> shamt) & ~(-1 << nbits)
.macro h3.bextmi rd rs1 shamt nbits
.if (\nbits < 1) || (\nbits > 8)
.err
.endif
.if (\shamt < 0) || (\shamt > 31)
.err
.endif
#if NO_HAZARD3_CUSTOM
    srli \rd, \rs1, \shamt
    andi \rd, \rd, ((1 << \nbits) - 1)
#else
.insn i 0x0b, 0x4, \rd, \rs1, (\shamt & 0x1f) | (((\nbits - 1) & 0x7 ) << 6)
#endif
.endm
```

Chapter 5. Debug

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

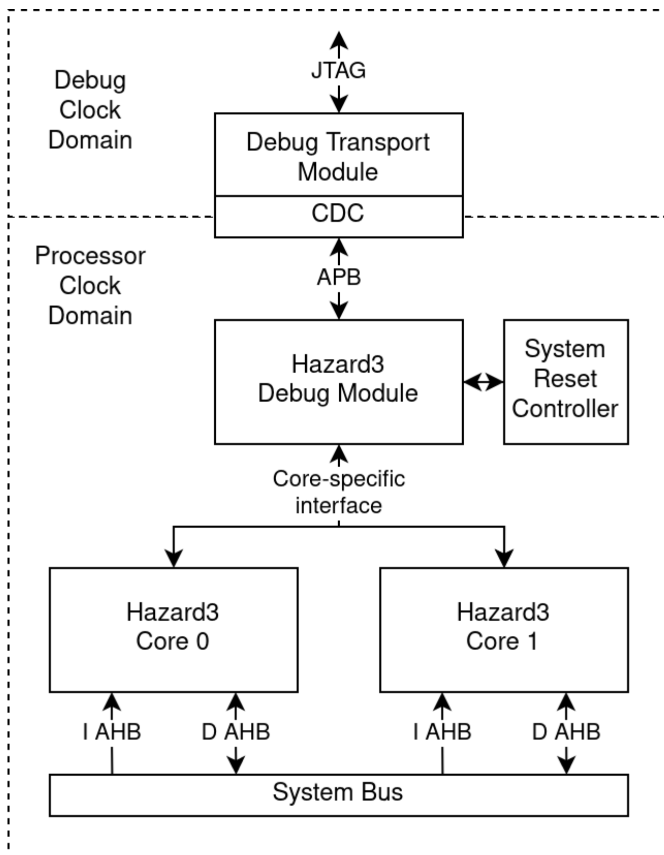
- Run/halt/reset control as required
- Abstract GPR access as required
- Program Buffer, 2 words plus `impebreak`
- Automatic trigger of abstract command (`abstractauto`) on `data0` or Program Buffer access for efficient memory block transfers from the host
- Support for multiple harts (multiple Hazard3 cores) connected to a single Debug Module (DM)
- The hart array mask registers, for applying run/halt/reset controls to multiple cores simultaneously
- (Optional) System Bus Access, either through a dedicated AHB5 manager interface, or multiplexed with a processor load/store port
- (Optional) An instruction address trigger unit (hardware breakpoints)

5.1. Debug Topologies

Hazard3's Debug Module (DM) 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
- Some reset request/acknowledge signals which require careful handshaking with system-level reset logic

This is shown in the example topology below.



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

Upstream of the DM is at least one Debug Transport Module, which bridges some host-facing interface such as JTAG to the APB DM Interface. Hazard3 provides an implementation of a standard RISC-V JTAG-DTM, but any APB master could be used. The DM 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 DM 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 DMs, 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. In this case, the first DM must be located at address `0x0` in the DMI address space, and you must set the `NEXT_DM_ADDR` parameter on each DM so that the debugger can walk the (null-terminated) linked list and discover all the DMs.

5.2. 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 DM 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 DM 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.

5.3. 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.

Appendix A: Instruction Cycle Counts

All timings are given assuming perfect bus behaviour (no downstream bus stalls), and that the core is configured with `MULDIV_UNROLL = 2` and all other configuration options set for maximum performance.

A.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]	

Instruction	Cycles	Note
`beq rs1, rs2, label`	1 or 2[1]	1 if correctly predicted, 2 if mispredicted.
`bne rs1, rs2, label`	1 or 2[1]	1 if correctly predicted, 2 if mispredicted.
`blt rs1, rs2, label`	1 or 2[1]	1 if correctly predicted, 2 if mispredicted.
`bge rs1, rs2, label`	1 or 2[1]	1 if correctly predicted, 2 if mispredicted.
`bltu rs1, rs2, label`	1 or 2[1]	1 if correctly predicted, 2 if mispredicted.
`bgeu rs1, rs2, label`	1 or 2[1]	1 if correctly predicted, 2 if mispredicted.
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	

A.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	
32 × 32 → 64 Multiply, Upper Half		
`mulh rd, rs1, rs2`	1	
`mulhsu rd, rs1, rs2`	1	
`mulhu rd, rs1, rs2`	1	
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	

A.3. A Extension

Instruction	Cycles	Note
Load-Reserved/Store-Conditional		
`lr.w rd, (rs1)`	1 or 2	2 if next instruction is dependent[2], an `lr.w`, `sc.w` or `amo*.w`.[3]
`sc.w rd, rs2, (rs1)`	1 or 2	2 if next instruction is dependent[2], 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]

A.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.

A.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	

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

A.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	

Instruction	Cycles	Note
`clmulr rd, rs1, rs2`	1	
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	
Zbkb (basic bit manipulation for cryptography)		
`pack rd, rs1, rs2`	1	
`packh rd, rs1, rs2`	1	
`brev8 rd, rs1`	1	
`zip rd, rs1`	1	
`unzip rd, rs1`	1	

A.7. Zcb Extension

Similarly to the C extension, this extension contains 16-bit variants of common 32-bit instructions:

- RV32I base ISA: `lbu`, `lh`, `lhu`, `sb`, `sh`, `zext.b` (alias of `andi`), `not` (alias of `xori`)
- Zbb extension: `sext.b`, `zext.h`, `sext.h`
- M extension: `mul`

They perform identically to their 32-bit counterparts.

A.8. Zcmp Extension

Instruction	Cycles	Note
`cm.push {rlist}, -imm`	$1 + n$	n is number of registers in rlist
`cm.pop {rlist}, imm`	$1 + n$	n is number of registers in rlist
`cm.popret {rlist}, imm`	4 (n = 1)[5] or 2 + n (n >= 2)[1]	n is number of registers in rlist
`cm.popretz {rlist}, imm`	5 (n = 1)[5] or 3 + n (n >= 2)[1]	n is number of registers in rlist
`cm.mva01s r1s', r2s'`	2	
`cm.mvsa01 r1s', r2s'`	2	

A.9. Branch Predictor

Hazard3 includes a minimal branch predictor, to accelerate tight loops:

- The instruction frontend remembers the last taken, backward branch
- If the same branch is seen again, it is predicted taken
- All other branches are predicted nontaken
- If a predicted-taken branch is not taken, the predictor state is cleared, and it will be predicted nontaken on its next execution.

Correctly predicted branches execute in one cycle: the frontend is able to stitch together the two nonsequential fetch paths so that they appear sequential. Mispredicted branches incur a penalty cycle, since a nonsequential fetch address must be issued when the branch is executed.

[1] A jump or 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 in stage 2 (e.g. an `add`) uses data from stage 3 (e.g. a `lw` result), a 1-cycle bubble is inserted between the pair. A load data → store data dependency is *not* an example of this, because data is produced and consumed in stage 3. However, load data → load address *would* qualify, as would e.g. `sc.w` → `beqz`.

[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.

[5] The single-register variants of `cm.popret` and `cm.popretz` take the same number of cycles as the two-register variants, because of an internal load-use dependency on the loaded return address.