Hazard 3

Design Guide and Reference Manual

Updated: 2025-Nov-17

Table of Contents

1. Introduction . . . 1
  1.1. Architecture Overview . . . 2
    1.1.1. Interfaces . . . 2
    1.1.2. Pipeline Stages . . . 3
    1.1.3. Bus Interfaces . . . 4
    1.1.4. Multiply/Divide . . . 4
    1.1.5. Constant-time Execution . . . 5
  1.2. List of RISC-V Specifications . . . 5
2. Configuration and Integration . . . 7
  2.1. Hazard3 Source Files . . . 7
  2.2. Top-level Modules . . . 8
  2.3. FPGA Synthesis . . . 8
  2.4. ASIC Synthesis . . . 8
    2.4.1. Register File Macros . . . 9
  2.5. Interfaces (Top-level Ports) . . . 9
    2.5.1. Clock and Reset Inputs . . . 9
    2.5.2. Power Control Signals . . . 10
    2.5.3. Memory Ordering Signals . . . 11
    2.5.4. Debug Module Controls . . . 12
    2.5.5. Shared System Bus Access . . . 13
    2.5.6. Interrupt Requests . . . 14
    2.5.7. Identification Signals . . . 15
    2.5.8. AHB5 Signals for 1-port CPU . . . 16
    2.5.9. AHB5 Signals for 2-port CPU . . . 17
  2.6. Configuration Parameters . . . 20
    2.6.1. Reset state configuration . . . 20
    2.6.2. Standard RISC-V ISA support . . . 20
    2.6.3. Custom Hazard3 Extensions . . . 22
    2.6.4. CSR support . . . 23
    2.6.5. External interrupt support . . . 26
    2.6.6. Identification Registers . . . 27
    2.6.7. Performance/size options . . . 27
3. Bus Behaviour . . . 30
  3.1. Protocol . . . 30
  3.2. Single and Dual-port . . . 30
  3.3. Bursts . . . 30
  3.4. Alignment . . . 30
  3.5. Memory Ordering for Loads and Stores . . . 30

3.6. Memory Ordering for Instruction Fetch . . . 31
3.7. Cacheable and Bufferable Attributes . . . 31
3.8. Idempotency . . . 31
3.9. 64-bit Accesses . . . 32
4. CSRs . . . 33
4.1. Standard M-mode Identification CSRs . . . 33
4.1.1. mvendorid . . . 33
4.1.2. marchid . . . 33
4.1.3. mimpid . . . 33
4.1.4. mhartid . . . 34
4.1.5. mconfigptr . . . 34
4.1.6. misa . . . 35
4.2. Standard M-mode Trap Handling CSRs . . . 35
4.2.1. mstatus . . . 35
4.2.2. mstatush . . . 36
4.2.3. medeleg . . . 36
4.2.4. mideleg . . . 36
4.2.5. mie . . . 36
4.2.6. mip . . . 37
4.2.7. mtvec . . . 37
4.2.8. mscratch . . . 38
4.2.9. mepc . . . 38
4.2.10. mcause . . . 38
4.2.11. mtval . . . 39
4.2.12. mcounteren . . . 39
4.3. Standard Memory Protection CSRs . . . 40
4.3.1. pmpcfg0…3 . . . 40
4.3.2. pmpaddr0…15 . . . 40
4.4. Standard M-mode Performance Counters . . . 41
4.4.1. mcycle . . . 41
4.4.2. mcycleh . . . 41
4.4.3. minstret . . . 41
4.4.4. minstreth . . . 41
4.4.5. mhpmcounter3…31 . . . 41
4.4.6. mhpmcounter3…31h . . . 42
4.4.7. mcountinhibit . . . 42
4.4.8. mhpmevent3…31 . . . 42
4.5. Standard Trigger CSRs . . . 42
4.5.1. tselect . . . 42
4.5.2. tdata1 . . . 42
4.5.3. tdata2 . . . 45

4.5.4. tdata3 . . . 45
4.5.5. tinfo . . . 45
4.6. Standard Debug Mode CSRs . . . 46
4.6.1. dcsr . . . 46
4.6.2. dpc . . . 47
4.6.3. dscratch0 . . . 47
4.6.4. dscratch1 . . . 47
4.7. Custom Debug Mode CSRs . . . 48
4.7.1. h3.dmdata0 . . . 48
4.8. Custom Interrupt Handling CSRs . . . 48
4.8.1. h3.meiea . . . 48
4.8.2. h3.meipa . . . 49
4.8.3. h3.meifa . . . 49
4.8.4. h3.meipra . . . 50
4.8.5. h3.meinext . . . 50
4.8.6. h3.meicontext . . . 51
4.9. Custom Memory Protection CSRs . . . 54
4.9.1. h3.pmpcfgm0 . . . 54
4.10. Custom Power Control CSRs . . . 55
4.10.1. h3.msleep . . . 55
4.11. Custom Identification CSRs . . . 55
4.11.1. h3.misa . . . 55
5. Custom Extensions . . . 60
5.1. Xh3misa: Hazard3 ISA identification register . . . 60
5.2. Xh3irq: Hazard3 interrupt controller . . . 60
5.3. Xh3pmpm: M-mode PMP regions . . . 61
5.4. Xh3power: Hazard3 power management . . . 61
5.4.1. h3.block . . . 61
5.4.2. h3.unblock . . . 62
5.5. Xh3bextm: Hazard3 bit extract multiple . . . 62
5.5.1. h3.bextm . . . 63
5.5.2. h3.bextmi . . . 64
6. Debug . . . 66
6.1. Debug Topologies . . . 66
6.2. Trigger Module . . . 68
6.2.1. Instruction Address Triggers . . . 68
6.2.2. Instruction Count Trigger . . . 69
6.2.3. Interrupt Trigger . . . 70
6.2.4. Exception Trigger . . . 70
6.3. Implementation-defined behaviour . . . 71
6.4. Debug Module to Core Interface . . . 72

Appendix A: Instruction Cycle Counts . . . 73
A.1. Base Instruction Set (RV32I) . . . 73
A.2. Integer Multiply and Divide (M) . . . 74
A.3. Atomics (A) . . . 75
A.4. Compressed Instructions (C or Zca) . . . 75
A.5. Privileged Instructions (including Zicsr) . . . 75
A.6. Load/Store Pair (Zilsd) . . . 76
A.7. Bit Manipulation (Zba, Zbb, Zbc, Zbs) . . . 76
A.8. Additional Basic Compressed Instructions (Zcb) . . . 77
A.9. Compressed Load/Store Pair (Zclsd) . . . 77
A.10. Push, Pop and Double Move (Zcmp) . . . 78
A.11. Branch Predictor . . . 78
Appendix B: Release Notes . . . 79
Version 1.1 . . . 79
Version 1.0.2 . . . 81
Version 1.0.1 . . . 81
Version 1.0 . . . 81

Chapter 1. Introduction

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

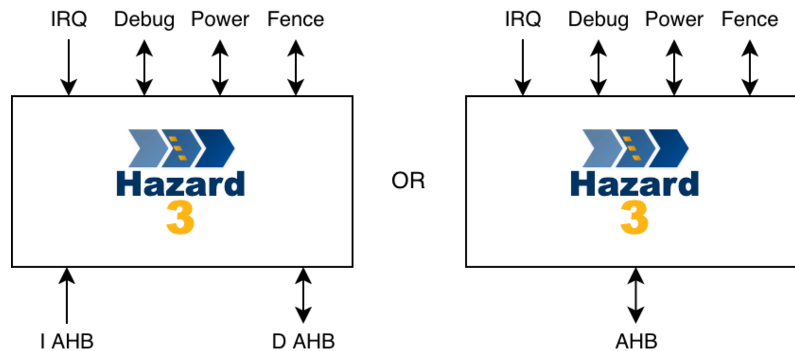
- `RV32I` or `RV32E`: 32-bit base instruction set
- `M`: integer multiply/divide/modulo
- `A`: atomic memory operations, with AHB5 global exclusives
- `C`: compressed instructions
- `Zicsr`: CSR access
- `Zilsd`: load/store pair
- `Zba`: address generation
- `Zbb`: basic bit manipulation
- `Zbc`: carry-less multiplication
- `Zbs`: single-bit manipulation
- `Zbkb`: basic bit manipulation for scalar cryptography
- `Zbkx`: crossbar permutation instructions
- `Zcb`: basic additional compressed instructions
- `Zcmp`: push/pop and double-move compressed instructions
- `Zclsd`: compressed load/store pair instructions
- Debug, Machine and User privilege/execution modes
- Privileged instructions `ecall`, `ebreak`, `mret` and `wfi`
- Physical memory protection (PMP) with up to 16 regions (configurable support for NAPOT and/or TOR matching)
- External debug support
- Instruction address trigger unit (hardware breakpoints)

This document describes Hazard3 version v1.1. See Release Notes for changes introduced in this and earlier versions.

1.1. Architecture Overview

1.1.1. Interfaces

Hazard3 communicates with system hardware using the following groups of signals:



AHB

The core accesses the bus using either 1 × or 2 × 32-bit AHB5 bus manager ports, depending on choice of top-level module (see Top-level Modules).

IRQ

The core implements standard RISC-V timer and software IRQ inputs, and a choice of a single external IRQ input or an integrated interrupt controller with support for up to 512 external IRQs.

Debug

Allows the Hazard3 Debug Module implementation to access core internals, and optionally to access the system bus directly by arbitrating with the core's load/store access.

Power

The core can negotiate power up/down of external hardware based on sleep state, and enter/exit sleep states cooperatively with other cores in the same multiprocessor system.

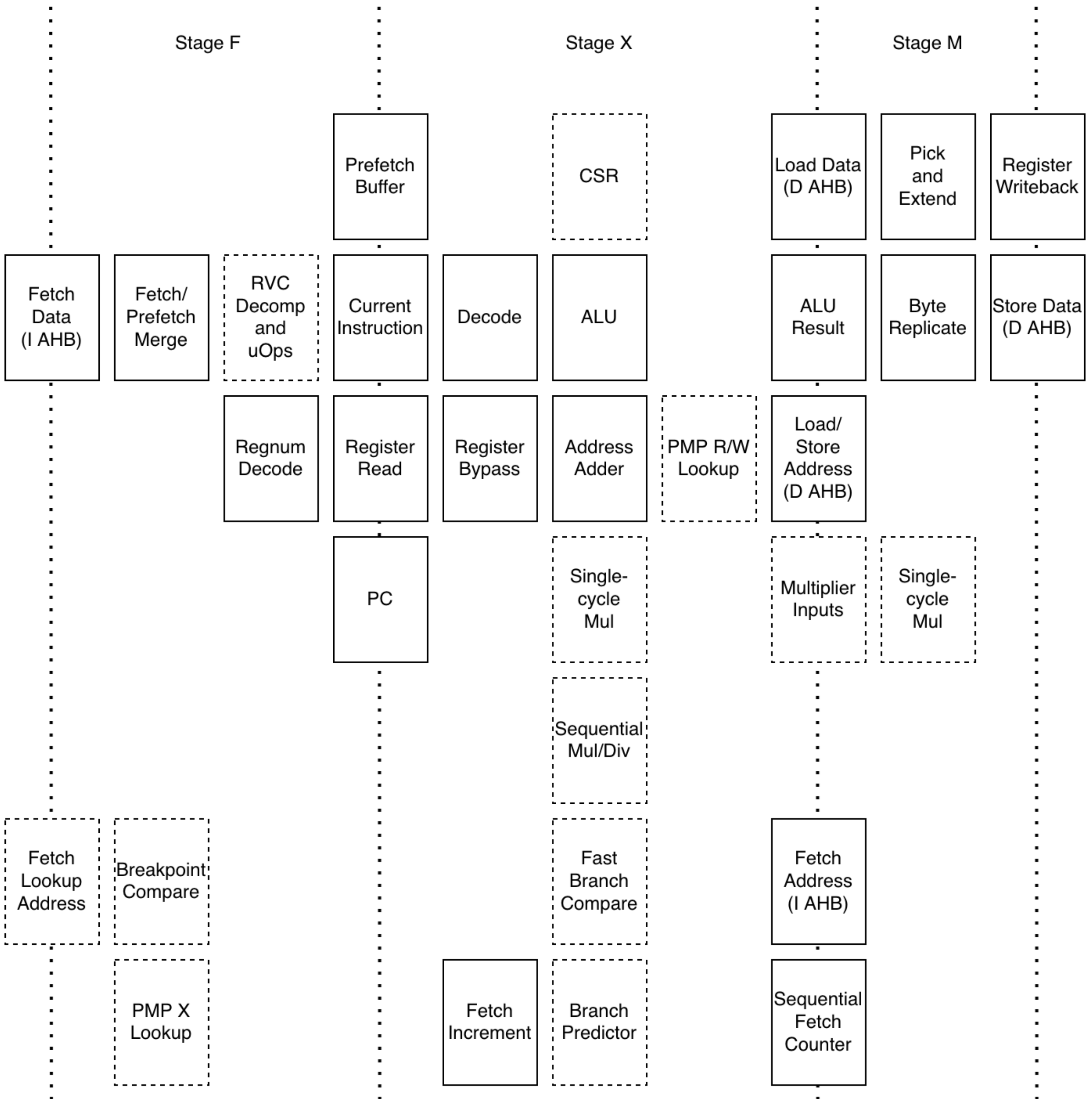
Fence

The core can request memory orderings; the system can stall the request (e.g. for a cache flush).

See Interfaces (Top-level Ports) for a full description of these interfaces.

1.1.2. Pipeline Stages

The diagram shows how hardware components are distributed through three pipeline stages. A dotted outline means an optional component. Dotted vertical lines are clock boundaries between stages; blocks that straddle these lines are synchronous elements, either registers or bus signals that are notionally registered outside the core. Dependencies (and time) flow from left to right.



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

  - Compares data-phase fetch address with breakpoint addresses
  - Looks up data-phase fetch address against PMP regions
- `X`: Execute
  - Decodes and execute instructions
  - Looks up load/store/AMO addresses against PMP regions
  - 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.

1.1.3. Bus Interfaces

Hazard3 implements either one or two AHB5 bus manager ports. The dual-port configuration adds a dedicated port for instruction fetch. Use the single-port configuration when ease of integration is a priority, since it supports simpler bus topologies. The dual-port configuration supports higher maximum frequency and greater clock-for-clock performance provided you either have a split (Harvard) bus architecture, or can handle multiple AHB5 managers on the same bus.

One of the key additions to AHB5 over older AHB versions is **exclusive accesses**, using the `HEXCL` and `HEXOKAY` signals. Exclusive accesses enable an ordered read-modify-write sequence with the guarantee that no other processor has written to the same memory location in between the read and write. Hazard3 uses AHB5 exclusives 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.

1.1.4. 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.

1.1.5. Constant-time Execution

Hazard3 supports `Zkt` constant-time execution in the following configurations:

- EXTENSION_M is `0`
- EXTENSION_M is 1, MUL_FAST is 1 and MULH_FAST is 1

The source of data-dependent timing is conditional sign correction in the `mulh` and `mulhsu` instructions when executed on the sequential multiply/divide circuit. You can avoid this by configuring these instructions to execute on the single-cycle multiplier instead. The remaining `Zkt` instructions are constant-time for all configurations.

Constant-time here refers to **data-independent timing**. Execution time can still be affected by other factors such as instruction alignment and external bus stalls.

Software can read the `Zkt` bit in h3.misa to check whether a given implementation has constant execution time for the `Zkt` instruction list.

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

Extension	Specification
`Zifencei` v2.0	Unprivileged ISA 20191213
`Zilsd` v1.0-rc3	Load/Store Pair for RV32 frozen v1.0-rc3
`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
`Zbkx` v1.0.1	Scalar Cryptography ISA extensions 20220218
`Zcb` v1.0.3-1	Code Size Reduction extensions frozen v1.0.3-1
`Zclsd` v1.0-rc3	Load/Store Pair for RV32 frozen v1.0-rc3
`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

Hazard3 is a soft processor design distributed as HDL source files. To use it in your project you must:

- Download the source files from the public git repository.
- Configure your synthesis and simulation flows to read Hazard3 source files.
- Select the appropriate top-level module for your bus architecture (`hazard3_cpu_1port` or `hazard3_cpu_2port`).
- Select the appropriate configuration parameters for your use case.
- Instantiate the top-level module, and connect or tie-off *all* ports.
- Instantiate any optional external components such as the Debug Module (`hazard3_dm`) and JTAG Debug Transport Module (`hazard3_jtag_dtm`).

The choice of configuration depends on your requirements for performance, code density, area, maximum frequency, level of memory protection, and number of system interrupts and interrupt priority levels. Configuration Parameters documents the available configuration parameters.

The minimal example SoC shows a complete example of a single-ported Hazard3 core connected to SRAM, a UART, a RISC-V platform timer and external debug components.

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. Hazard3 is distributed under the permissive Apache 2.0 licence; a full copy of the licence text is available in the git repository.

For more information on the Verilog 2005 language, refer to standard IEEE 1364-2005.

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 and Vivado. 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 inference 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 here 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.

2.4.1. Register File Macros

Generally the register file should be synthesised. Synthesised register files offer the best portability and the most flexibility during layout. However there may be an area advantage to implementing the register file using a hard layout macro.

Hazard3 supports the use of register file or SRAM macros, by modifying `hazard3_regfile_1w2r.v` to instantiate the appropriate macros. The configuration should be one of the following:

- One instance: two read ports, one write port
- Two instances: one read port, one write port, with the write ports tied together

The memory dimensions are:

- 32 bits wide
- 32 rows deep (RV32I) or 16 rows deep (RV32E) — depends on configuration of EXTENSION_E
- No requirement for bit write enable

When a write and a read simultaneously occur at the same address, the write must succeed. The read data in this case is don't-care.

The read data from address 0 is don't-care, because the special case for register `x0` is handled in the register bypass logic. This means a full 32-row or 16-row RAM can be used.

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.

NOTE Hazard3 adopts the convention that all signals are active-high, with the exception of the reset input `rst_n`.

2.5.1. 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 (`HCLK`). If the Xh3power extension is configured, you should instantiate an external clock gate on this clock, controlled by the `clk_en` output.

Width	In/Out	Name	Description
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.

2.5.2. Power Control Signals

These signals are used in the implementation of internal sleep states as configured by the h3.msleep csr. They are used only when the Xh3power extension is enabled.

Width	In/Out	Name	Description
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 `h3.msleep` CSR. It then waits for deassertion of `pwrup_ack` before taking further action. The processor asserts `pwrup_req` when it intends to wake from its low-power state. It then waits for `pwrup_ack` before fetching the first instruction from the bus.
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.

Width	In/Out	Name	Description
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. Tie low if Xh3power is not configured.

2.5.3. Memory Ordering Signals

Also see Memory Ordering for Loads and Stores for more information on Hazard3's memory model.

Width	In/Out	Name	Description
1	Out	`fence_i_vld`	Indicates the core is executing a `fence.i` instruction. Remains asserted until `fence_rdy` goes high. Never asserted at the same time as `fence_d_vld`. The core waits for all in-progress bus accesses (load/store and instruction fetch) to complete before asserting this signal, and does not issue further instruction fetches for as long as this signal is asserted. Once `fetch_rdy` is seen high, the core flushes its prefetch buffer to order the fetch of younger instructions against the fence. It then resumes normal execution.

Width	In/Out	Name	Description
1	Out	`fence_d_vld`	Indicates the core is executing a non-instruction fence, such as `fence rw, rw`. Remains asserted until `fence_rdy` goes high. Never asserted at the same time as `fence_i_vld`. The core waits for any in-progress load/store bus accesses to complete before asserting this signal, and does not issue further loads/stores for as long as this signal is asserted.
1	In	`fence_rdy`	Signal to the core that the memory subsystem has finished processing the fence requested by `fence_i_vld` or `fence_d_vld`. If no special handling of fences is required, tie this signal high.

2.5.4. Debug Module Controls

All Debug Module signals should be connected to the signal with the matching name on the Hazard3 Debug Module implementation (hazard3_dm).

Width	In/Out	Name	Description
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 execution. 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.

2.5.5. 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 from the Debug Module to the system bus.

Width	In/Out	Name	Description
32	In	dbg_sbus_addr	Address for SBA 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 SBA 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 SBA 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 SBA arbitrated with this core's load/store access. Tie low if this feature is not used.

Width	In/Out	Name	Description
1	Out	`dbg_sbus_rdy`	Transfer stall signal for SBA arbitrated with this core's load/store access. Disconnect if this feature is not used.
1	Out	`dbg_sbus_err`	Bus fault signal for SBA arbitrated with this core's load/store access. Disconnect if this feature is not used.
32	In	`dbg_sbus_wdata`	Write data bus for SBA 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 SBA arbitrated with this core's load/store access. Disconnect if this feature is not used.

2.5.6. Interrupt Requests

All interrupts are level-sensitive and synchronous to `clk`. Asynchronous interrupts must be synchronised externally before entering the processor. You must use the ungated version of `clk` (`clk_always_on`) for synchronisation if you use the interrupt to wake from sleep states.

Briefly asserting then deasserting an interrupt signal is not guaranteed to cause the processor to enter the relevant handler; the processor ignores interrupts it is not able to take at that time, for example when interrupts are disabled via `mstatus.mie`. There is also no guarantee that the processor will *not* take such an interrupt. Peripherals and software should follow these guidelines for reliable interrupt handling:

- When an interrupt is asserted, it should remain asserted until the condition which caused the interrupt is cleared. For example, an RX FIFO valid interrupt should remain asserted until the processor pops data from the FIFO.
- Interrupt handlers should check the peripheral status at the top of the handler to see if it still requires servicing. If an interrupt is synchronised externally, it may still briefly be observed as asserted after returning from the handler, causing re-entry.
- To avoid the previous issue, interrupt handlers should acknowledge (clear) the peripheral interrupt as soon as possible in the handler, to ensure the interrupt is seen deasserted before the processor returns from the interrupt handler.

Width	In/Out	Name	Description
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.

2.5.7. Identification Signals

Width	In/Out	Name	Description
32	In	mhartid_val	Set the value of the mhartid CSR, which software uses to determine on which hart it is running. Hazard3 implements one hart per core, so this is effectively a per-core identification value. At least one hart must have the value of all-zeroes. Generally you should tie this off to a unique constant value per core. Dynamic mhartid is useful in scenarios such as dual-core lockstep: cores must execute identically when lockstep is engaged so their mhartid_val should be driven to the same value.

Width	In/Out	Name	Description
4	In	eco_version	Set the value of the eco version number in mimpid. On ASIC this should be connected to metal-programmable tie cells so that it can be incremented with metal changes. The initial value should be all-zeroes. On FPGA this should be tied to all-zeroes.

2.5.8. AHB5 Signals for 1-port 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.
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.

Width	In/Out	Name	Description
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 `ERROR` 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.
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.

2.5.9. AHB5 Signals for 2-port 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 to indicate a read transfer.

Port I (Instruction)			
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.
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 ERROR 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.

Port I (Instruction)			
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).
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 ERROR 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.

C is equivalent here to the Zca extension, because Hazard3 does not implement F or D.

Hazard3 supports some other extensions which use 16-bit instructions: Zcmp (EXTENSION_ZCMP), Zcb (EXTENSION_ZCB) and Zclsd (EXTENSION_ZCLSD). You must set EXTENSION_C to enable basic compressed instruction support before enabling any of these other extensions.

Default value: `1`

EXTENSION_E

Implement the RV32E base extension rather than RV32I. RV32E reduces the number of integer registers from 31 to 15. Set `1` to select RV32E, `0` to select RV32I.

Default value: `0`

EXTENSION_M

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

The exact circuit used to implement these instructions, and the resulting area and performance,

depends on the following parameters: MULDIV_UNROLL, MUL_FAST, MUL_FASTER, and MULH_FAST.

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_ZBKB

Support for Zbkb basic bit manipulation for cryptography.

Requires: EXTENSION_ZBB = 1.

NOTE	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_ZBKX

Support for Zbkx crossbar permutation 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_ZCB

Support for Zcb basic additional compressed instructions.

Requires: EXTENSION_C = 1, EXTENSION_M = 1 and EXTENSION_ZBB = 1.

NOTE The RISC-V specifications state that Zcb depends on Zca; on Hazard3, Zca is synonymous with C, as the F and D extensions are not supported.

NOTE Some of the 16-bit opcodes from Zcb are compressed aliases for 32-bit opcodes from M and Zbb, which is why those extensions must also be enabled.

Default value: `0`

EXTENSION_ZCLSD

Support for Zclsd compressed load/store pair instructions.

Requires: EXTENSION_ZILSD = 1 and EXTENSION_C = 1.

Default value: `0`

EXTENSION_ZCMP

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

Requires: EXTENSION_C = 1.

NOTE The RISC-V specifications state that Zcmp depends on Zca; on Hazard3, Zca is synonymous with C, as the F and D extensions are not supported.

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 = 1), this instruction must be implemented.

Default value: `0`

EXTENSION_ZILSD

Support for Zilsd load/store pair instructions.

Hazard3 issues a 64-bit load or store instruction as two 32-bit memory accesses. Therefore this extension improves code density but does not directly improve performance.

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

Enable the custom PMPCFGMx CSRs, which can enforce PMP regions in M-mode without locking the regions. See Xh3pmpm: M-mode PMP regions.

Default value: `0`

EXTENSION_XH3POWER

Custom power management controls for Hazard3. This adds the h3.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.

Setting this parameter to `0` makes the core incapable of handling exceptions and interrupts. Instructions which would raise exceptions, such as illegal opcodes or unaligned load/store addresses, instead execute as NOPs.

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 = 1.

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 implements all PMP registers `pmpaddr0` through `pmpaddr15` and `pmpcfg0` through `pmpcfg3` if PMP_REGIONS is greater than zero. They are implemented in the sense that software can read and write them without trapping. However, configuration bits for regions PMP_REGIONS through 15 are hardwired to zero.

Requires: CSR_M_TRAP = 1.

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. For example, *G* = 3 means 32-byte protection granularity.

If *G* > 0, `pmcfg.a` cannot be set to NA4; attempting to do so will set the region to OFF instead. The *G* LSBs of each `pmpaddr` register read back as all-zeroes when `pmpcfg.a` is set to TOR or OFF.

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

Increasing PMP_GRAIN from its default value reduces the area and delay cost associated with the PMP unit.

Default value: `0`

PMP_MATCH_NAPOT

PMP_MATCH_NAPOT: Enable PMP support for the NAPOT (naturally-aligned power-of-two) and NA4 (naturally-aligned four-byte) matching modes. When disabled, attempting to select these modes will set the PMP region to OFF.

Default value: 1

PMP_MATCH_TOR

PMP_MATCH_TOR: Enable PMP support for the TOR (top-of-range) matching mode. When disabled, attempting to select this mode will set the region to OFF.

Note that NA4 and NAPOT use a separate comparator circuit from TOR comparisons. Setting both PMP_MATCH_NAPOT and PMP_MATCH_TOR incurs the area cost of both circuits. For best area efficiency, select the best single comparator type for your application, noting that TOR covers all the possibilities of NAPOT if there are no gaps between the regions. Also consider whether PMP_GRAIN can be increased from its default value.

Default value: `0`

PMP_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

Enable the following hardware functionality:

- Support for run/halt and instruction injection from an external Debug Module,
- Support for Debug Mode
- Debug Mode CSRs
- A minimal Trigger Module with an exception trigger, an interrupt trigger and an instruction count

trigger (see Trigger Module)

If the Hazard3 Debug Module is also instantiated and correctly connected to the core, this is sufficient for debugging the core from any RISC-V-compliant debug host.

Consider also enabling hardware breakpoints using BREAKPOINT_TRIGGERS if debugging code in read-only memory is an anticipated use case.

Requires: CSR_M_MANDATORY = 1 and CSR_M_TRAP = 1.

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 = 1.

Default value: `0`

2.6.5. External interrupt support

NUM_IRQS

Number of external IRQs. Minimum 1, maximum 512. Note that if EXTENSION_XH3IRQ = `0`, the internal interrupt controller is not present. In this case 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 the internal interrupt controller described in Xh3irq: Hazard3 interrupt controller. The h3.meipra array CSR configures the individual priorities.

The number of distinct levels is `(1 << IRQ_PRIORITY_BITS)`. The minimum value is `0` (1 level), and the maximum value is `4` (16 levels). Note that multiple priority levels with a large number of IRQs will have a severe effect on timing.

Requires: EXTENSION_XH3IRQ = 1.

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. Use this when the interrupt already comes straight from a register, e.g. when you have synchronised an interrupt to the processor clock from an asynchronous source; the

additional register inside the processor is redundant in this case.

Ignored if EXTENSION_XH3IRQ = `0`. In this case, when the internal interrupt controller is not present, all external IRQ inputs are OR'd together before being registered in a single flip-flop.

Default value: all-zeroes (not bypassed).

2.6.6. Identification Registers

MVENDORID_VAL

Value of the mvendorid CSR. Should be either a JEDEC JEP-106-compliant vendor ID, or all-zeroes.

Bits `31:7` store the continuation code count, and bits `6:0` are the ID. The parity bit is not stored.

IMPORTANT	The number of continuation codes is *one less than* the JEP106 bank number. For example, Raspberry Pi has an ID of `0x13` (19 decimal) in bank 10 (decimal). This has a continuation code count of 9, yielding an `mvendorid` of `0x00000493`.

Requires: CSR_M_MANDATORY = 1.

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.

Requires: CSR_M_MANDATORY = 1.

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

Configure bits-per-clock for the sequential multiply/divide circuit, if present. Must be a power of 2.

Default value: 1

MUL_FAST

Use a single-cycle multiply circuit for `MUL` instructions, retiring to stage 3 by default (one throughput cycle, two latency cycles).

The sequential multiply/divide circuit is still used for `mulh`, `mulhu` and `mulhsu`.

Default value: `0`

MUL_FASTER

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

Requires: MUL_FAST = 1.

Default value: `0`

MULH_FAST

Extend the fast multiply circuit to also cover `mulh`, `mulhu` and `mulhsu`. Removethe multiply functionality from the sequential multiply/divide circuit.

Requires: MUL_FAST = 1.

Default value: `0`

FAST_BRANCHCMP

Instantiate a separate comparator (equal, less-than-signed, less-than-unsigned) for branch comparisons, rather than using the ALU. Enabling the separate comparator improves the address-phase timing for instruction fetch, especially if the `Zba` extension is enabled (EXTENSION_ZBA). Disabling may save area.

Default value: 1

RESET_REGFILE

If 1, the `rst_n` input resets all general purpose registers to all-zeroes. If `0`, it does not. There are around 1k bits in the register file, so the reset can be disabled to permit block RAM and LUT RAM inference on FPGA, or to reduce flop area on ASIC.

Outside of the register file, all other flops in Hazard3 are always reset by the `rst_n` input, no matter the value of this parameter.

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 non-taken branch, a `fence.i` or a trap.

Successful prediction eliminates the 1-cyle fetch bubble on a taken branch, so tight loops execute faster. See Branch Predictor for more information on this feature.

Requires: EXTENSION_ZIFENCEI = 1.

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, in bits `1:0`, 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. If all four standard M-mode vectors (exception, `msip`, `mtip` and `meip`) are in use, this means 64-byte alignment.

Default: All writable except for bit 1.

Chapter 3. Bus Behaviour

Hazard3 implements one or two AHB5 bus manager ports. AHB5 Signals for 1-port CPU describes the signal-level implementation for a single-ported processor, and AHB5 Signals for 2-port CPU for dual-ported. This section describes their behaviour at a higher level, and the requirements Hazard3 imposes on your bus subsystem.

3.1. Protocol

Hazard3 implements the AHB5 protocol, as described in the AMBA 5 AHB protocol specification.

3.2. Single and Dual-port

In a dual-ported processor, loads and stores issue to the D port and instruction fetch issues to the I port. In a single-ported processor, all accesses issue to the single available port.

When the processor experiences internal contention, loads and stores take priority over instruction fetch. The rationale for this behaviour is that stalling a load or store will always increase the number of cycles required to execute a program, but stalling an instruction fetch has no cycle cost if the prefetch buffer is able to cover the gap.

For single-ported implementations it's highly recommended to enable compressed instruction support (EXTENSION_C = 1). This reduces contention between instruction fetch and load/store, increasing performance.

3.3. Bursts

Hazard3 never generates bursts. `HTRANS` is always `IDLE` or `NSEQ`; `HBURST` is always `SINGLE`.

3.4. Alignment

All Hazard3 bus accesses are naturally aligned. That is, address modulo access size is always zero; `HADDR % (1 << HSIZE) == 0` (if `HTRANS` is not `IDLE`). Additionally, instruction fetch is always word-aligned and word-sized.

The Zilsd instructions `ld` and `sd` notionally perform 64-bit accesses, but Hazard3 issues these as pairs of 32-bit accesses over its 32-bit AHB bus. The alignment of each access is still four bytes.

3.5. Memory Ordering for Loads and Stores

Hazard3 is sequentially consistent for loads and stores when it is connected to a sequentially consistent memory subsystem. Equivalently, if a load/store A is before another load/store B in program order, then A issues before B on the core's AHB5 load/store interface. Architecturally this is a (stronger) subset of the RVTSO memory model.

When executing a `fence` instruction, the core waits for all earlier loads and stores to complete before

raising the `fence_d_vld` signal, then waits for `fence_rdy` before issuing more loads or stores. This has no effect on the order the core issues loads and stores to the bus, as they already unconditionally issue in program order. See Memory Ordering Signals for more information about the core's external fence signals.

If a core A executes a `fence` and core B also executes a `fence` at a later time, the memory subsystem must perform any necessary synchronisation to ensure A's load/store accesses issued before A's fence are observed by B's load/store accesses issued after B's fence. Hazard3 does not distinguish different types of data fences: all fences should be treated as `fence rwio, rwio`.

The core is guaranteed not to issue any load or store accesses while `fence_d_vld` is asserted. For a two-ported implementation this is a good opportunity for system hardware to manipulate any logic such as cache controllers attached to the core's load/store port. No such guarantee is made for instruction fetch, so in a single-ported implementation you cannot assume based on `fence_d_vld` that the AHB5 port is idle.

3.6. Memory Ordering for Instruction Fetch

Hazard3 does not order stores before instruction fetch except with an explicit `fence.i` instruction. This instruction waits for in-progress stores to complete, asserts `fence_i_vld`, waits for `fence_rdy`, and then flushes the prefetch buffer to ensure the next instruction data to be executed is fetched after the instruction fence was observed by the system. It is up to the system designer to ensure that the instruction fetch AHB5 port observes earlier writes from the load/store AHB5 port; this may require flushing of external caches.

In system implementations which lack core-local caches it is usually sufficient to ignore the `fence_i_vld` and `fence_d_vld` signals, and tie `fence_rdy` high.

3.7. Cacheable and Bufferable Attributes

Hazard3 marks all transfers as non-cacheable and non-bufferable using `HPROT[3:2]`. These signals are tied off to constants and are not controllable by software.

All loads are issued to the external bus, even if they alias with a prior store; there is no store-to-load forwarding. All stores are issued to the external bus, even if they alias with a later store; there is no dead store elimination or write merging. Stores are issued immediately to the bus without buffering inside the core.

It is possible to use Hazard3 in systems with caches, but an alternative mechanism must be implemented to control cacheability of accesses. For example, implement multiple mirrors of the cached memory with attributes decoded from the address.

3.8. Idempotency

Hazard3 expects that all memory used for instruction fetch is read-idempotent. This limitation comes about because PMP execute permissions are checked at the point an instruction is *executed*, not the point it is fetched. This is a deliberate design decision that reduces the logic depth of fetch address generation logic. As a consequence it is possible to trigger reads from arbitrary addresses by

jumping to them (though the data returned by that read is discarded if it lacks execute permissions).

You must not permit instruction fetch from IO regions if U_MODE is configured, because this would allow U-mode software to cause IO read side effects on privileged registers by (fatally) jumping to an IO address. This can be enforced in one of two ways:

- For a 2-port processor, do not connect instruction fetch to IO regions. This is already desirable to improve routing and logic complexity.
- For a 1-port processor, filter instruction fetch from IO regions by rejecting transfers if `HPROT[0]` is `0`.

This limitation does not apply to load/store; Hazard3 checks load/store addresses for PMP read/write permissions before issuing the transfer to the bus. However, idempotency is still a concern for instructions which generate multiple bus accesses, namely:

- Zcmp: `cm.push`, `cm.pop`, `cm.popret`, `cm.popretz`
- Zilsd: `ld`, `sd`
- Zclsd: `c.ld`, `c.ldsp`, `c.sd`, `c.sdsp`

An interrupt occurring mid-instruction terminates its execution immediately. When returning from the interrupt, the instruction restarts from the beginning, as there is no provision in RISC-V for saving and restoring the execution progress. Therefore each individual read or write may execute multiple times before the instruction eventually completes uninterrupted. This can cause lost or duplicated data when accessing IO regions. For this reason these instructions should be avoided when accessing non-idempotent memory.

3.9. 64-bit Accesses

Hazard3 implements the Zilsd and Zclsd extensions, which perform 64-bit loads and stores to and from pairs of registers. The register pair is always an even register plus the consecutively next higher-numbered register.

64-bit loads and stores are implemented as pairs of 32-bit AHB5 transfers. Since these accesses only require 4-byte alignment, the load/store address can also be 4-byte-aligned, even though it is notionally an 8-byte transfer.

The RISC-V specification does not guarantee the order in which these two writes are issued, and Hazard3 takes advantage of this to simplify fault handling. When the base address register `rs1` is an even-numbered register, the odd-numbered register in the `rd` or `rs2` pair is transferred first. Conversely, when `rs1` is an odd-numbered register, the even register in the pair is transferred first. This ensures that a fault occurring on the second half of an `ld` instruction does not modify `rs1`. It also means that a consecutively-addressed sequence of 64-bit load/stores is not necessarily consecutively addressed on the AHB5 port. When using `ld` and `sd` instructions for IO, ensure that the peripheral is not sensitive to the order of transfers.

Chapter 4. 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.

4.1. Standard M-mode Identification CSRs

4.1.1. 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. The MVENDORID_VAL parameter sets the value.

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.

4.1.2. 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

4.1.3. mimpid

Address: `0xf13`

Implementation identifier. Most of this register is hardwired to indicate the release version of the

Hazard3 RTL.

Bits	Name	Description
31	`prerelease`	Value of 1 indicates this Hazard3 instance was synthesised from RTL not yet released to the `stable` branch.
30:28	**-**	RES0
27:24	`major`	Major release version, i.e. the `1` in `v1.2.3`
23:16	`minor`	Minor release version, i.e. the `2` in `v1.2.3`
15:8	`patch`	Patch version, i.e. the `3` in `v1.2.3`
7:4	`eco`	Connected to external signals, initially zero
3:0	**-**	RES0

This should match the version of the Github release of the processor's source code. For example, version 1.0.2 would have a `major` of 1, `minor` of 0 and `patch` of 2.

`eco` is connected to the `eco_version` input port at Hazard3 top level; the implementer may connect this to metal tie cells in their integration so that they can increment the ECO (engineering change order) number with metal revisions that affect the processor, or may simply tie it to 0. See Identification Signals.

NOTE Versions of Hazard3 older than v1.1 allowed the implementer to set the `mimpid` contents arbitrarily, with a recommendation to set it to the git hash. The hashes of released versions are: `86fc4e3f` (RP2350, v1.0-rc1), `918aaee10` (v1.0), `82729101` (v1.0.1) and `787da131a` (v1.0.2).

4.1.4. mhartid

Address: `0xf14`

Hart identification register. Read-only.

The value of this register is configured by the `mhartid_val` input port (see Identification Signals). In a correctly configured system, the value of `mhartid` is unique on each hart, and one hart has an ID of all-zeroes. There is no strict requirement to assign hart IDs consecutively, but there may be performance or memory usage advantages to doing so on certain operating systems.

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

4.1.5. 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.

4.1.6. 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`	Always 1 to indicate presence of Xh3misa: Hazard3 ISA identification register, and possibly other custom extensions.
20	`u`	1 if User mode is supported, otherwise 0.
12	`m`	1 if the M extension is present (EXTENSION_M is 1)
8	`i`	1 if the base ISA is RV32I (EXTENSION_E is `0`)
4	`e`	1 if the base ISA is RV32E (EXTENSION_E is 1)
2	`c`	1 if the C extension is present (EXTENSION_C is 1)
1	`b`	1 if Zba, Zbb and Zbs are all present (EXTENSION_ZBA, EXTENSION_ZBB and EXTENSION_ZBS are 1)
0	`a`	1 if the A extension is present (EXTENSION_A is 1)

4.2. Standard M-mode Trap Handling CSRs

4.2.1. 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.

Bits	Name	Description
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.

4.2.2. mstatush

Address: `0x310`

Hardwired to `0`.

4.2.3. medeleg

Address: `0x302`

Unimplemented, as neither U-mode traps nor S-mode are supported. Access will cause an illegal instruction exception.

4.2.4. mideleg

Address: `0x303`

Unimplemented, as neither U-mode traps nor S-mode are supported. Access will cause an illegal instruction exception.

4.2.5. mie

Address: `0x304`

Interrupt enable register. This is not to be confused with `mstatus.mie`, which is a global enable, and has 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 h3.meiea.
7	`mtie`	Timer interrupt enable. A timer interrupt is requested when `mie.mtie`, `mip.mtip` and `mstatus.mie` are all 1.

Bits	Name	Description
3	msie	Software interrupt enable. A software interrupt 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 h3.meiea, and keeps the upper half of mie reserved.

4.2.6. 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 h3.meipa) and enabled in h3.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.

4.2.7. 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.

Bits	Name	Description
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.

4.2.8. 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.

4.2.9. 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.

4.2.10. 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`)

Cause	Description
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
8	Environment call from U-mode
11	Environment call from M-mode

4.2.11. mtval

Address: `0x343`

Hardwired to 0.

4.2.12. 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
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.

4.3. Standard Memory Protection CSRs

4.3.1. 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). 1 (TOR, top of range) is not supported. Attempting to write an unsupported value will set the region to OFF.
2	`X`	Execute permission
1	`W`	Write permission
0	`R`	Read permission

4.3.2. 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.

4.4. Standard M-mode Performance Counters

4.4.1. 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."

4.4.2. 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.

4.4.3. 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.

4.4.4. 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.

4.4.5. mhpmcounter3…31

Address: `0xb03` through `0xb1f`

Hardwired to 0.

4.4.6. mhpmcounter3...31h

Address: `0xb83` through `0xb9f`

Hardwired to 0.

4.4.7. 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.

4.4.8. mhpmevent3...31

Address: `0x323` through `0x33f`

Hardwired to 0.

4.5. Standard Trigger CSRs

Trigger CSRs control Hazard3's trigger module, described in Trigger Module. They are implemented if DEBUG_SUPPORT = 1, and otherwise raise illegal instruction exceptions when accessed.

4.5.1. tselect

Address: `0x7a0`

Select a trigger for configuration. The selected trigger is described in tinfo and can be configured through tdata1 and tdata2. See Trigger Module for an overview of Hazard3's trigger capabilities.

This register implements exactly enough bits to index the configured number of triggers. Remaining bits are hardwired to zero.

4.5.2. tdata1

Address: `0x7a1`

Configure the trigger selected by tselect.

Bits	Name	Description
31:28	type	On Hazard3 this field is hardwired for a given value of tselect: • 0: no trigger • 2: address/data match • 3: instruction count • 4: interrupt • 5: exception
27	dmode	Claim this trigger for Debug Mode use. Only writable by Debug Mode. When set to 1, other modes cannot write to tdata* registers for this trigger. This bit must be set to enable Debug Mode entry from a trigger (action = 1). Hazard3 hardwires this field to zero for the instruction count trigger, which only supports breaking to M-mode. It has normal read/write behaviour for other trigger types.

The layout of bits 26:0 of this register depends on the value of type.

mcontrol

mcontrol is the alias for tdata1 when type is 2 (address/data match trigger). Hazard3 implements only exact instruction address match (breakpoints).

Bits	Name	Description
26:21	maskmax	Hardwired to zero; only exact matches are supported
20	hit	Hardwired to zero; this feature is not implemented
19	select	Hardwired to zero; only address matching is supported (not data matching)
18	timing	Hardwired to zero; the breakpoint takes place before the matching instruction executes (but after all earlier instructions in program order are observed to have completed)
17:16	sizelo	Hardwired to zero; all instruction sizes are matched
15:12	action	Supports the values 0 (break to M-mode ebreak exception handler) and 1 (break to Debug Mode). Bits 15:13 are hardwired to zero. An action of 0 is ignored when tcontrol.mte is 0. An action of 1 is ignored when tdata1.dmode is 0.
11	chain	Hardwired to zero; chaining is not supported
10:7	match	Hardwired to zero; match is always on the exact address
6	m	Write 1 to enable matching during M-mode execution

Bits	Name	Description
3	u	Write 1 to enable matching during U-mode execution. Hardwired to zero if U_MODE = 0.
2	execute	Write 1 to enable matching on instruction addresses
1	store	Hardwired to zero; store address/data matching is not supported
0	load	Hardwired to zero; load address/data matching is not supported

icount

icount is the alias for tdata1 when type is 3 (instruction count trigger).

Bits	Name	Description
24	hit	Hardwired to zero; this feature is not implemented
23:10	count	Hardwired to 1; only single-stepping is supported
9	m	Write 1 to enable matching during M-mode execution, if tcontrol.mte is also set. Self-clears when this trigger fires (in any privilege mode).
6	u	Write 1 to enable matching during U-mode execution. Hardwired to zero if U_MODE = 0. Self-clears when this trigger fires (in any privilege mode).
5:0	action	Hardwired to 0; this trigger only supports breaking to M-mode with an exception cause of 3 (ebreak). The corresponding Debug Mode functionality is already provided by dcsr.step.

See Instruction Count Trigger for more information about this trigger's behaviour on Hazard3.

itrigger

itrigger is the alias for tdata1 when type is 4 (interrupt trigger).

Bits	Name	Description
24	hit	Hardwired to zero; this feature is not implemented
9	m	Write 1 to enable matching during M-mode execution, if tcontrol.mte is also set.
6	u	Write 1 to enable matching during U-mode execution. Hardwired to zero if U_MODE = 0.
5:0	action	Hardwired to 1; this trigger only supports breaking to Debug Mode (as an M-mode breakpoint exception triggered immediately after M-mode interrupt entry would be unrecoverable).

See Interrupt Trigger for more information about this trigger's behaviour on Hazard3.

etrigger

etrigger is the alias for tdata1 when type is 5 (exception trigger).

Bits	Name	Description
24	hit	Hardwired to zero; this feature is not implemented
9	m	Write 1 to enable matching during M-mode execution, if tcontrol.mte is also set.
6	u	Write 1 to enable matching during U-mode execution. Hardwired to zero if U_MODE = 0.
5:0	action	Hardwired to 1; this trigger only supports breaking to Debug Mode (as an M-mode breakpoint exception triggered immediately after M-mode interrupt entry would be unrecoverable).

See Exception Trigger for more information about this trigger's behaviour on Hazard3.

4.5.3. tdata2

Address: 0x7a2

The interpretation of this CSR depends on the type field of tdata1:

type = 2
: Address/data trigger: tdata2 contains the address/data to be matched. On Hazard3 this is always an instruction address, and tdata2 only implemenents enough bits to store any valid program counter (multiple of IALIGN).

type = 3
: Instruction count trigger: tdata2 is hardwired to zero.

type = 4
: Interrupt trigger: tdata2 contains a bitmap of the interrupt mcause values upon which this trigger will match. For example, setting bit 11 (value 0x800) enables triggering on the machine external IRQ vector (mip.meip).

type = 5
: Exception trigger: tdata2 contains a bitmap of the non-interrupt mcause values upon which this trigger will match. For example, setting bit 2 (value 0x4) enables triggering on illegal instruction exceptions.

4.5.4. tdata3

Address: 0x7a3

Hardwired to zero.

4.5.5. tinfo

Address: 0x7a4

Lists the capabilities of the trigger currently selected by tselect. This takes the form of a bitmap of values supported by tdata1.type. For example, bit 2 being set (a value of 0x4) indicates the trigger supports type = 2, "address/data match".

Bits	Description
5	Supports `type` = `5`, exception trigger
4	Supports `type` = `4`, interrupt trigger
3	Supports `type` = `3`, instruction count trigger
2	Supports `type` = `2`, address/data match trigger
0	Supports `type` = `0`, disabled or no trigger

Hazard3 triggers support only a single `type` each. Therefore `tinfo` always has exactly one bit set, and the `tdata1.type` field is read-only.

If bit `0` is set, and no other bit is set, there is no trigger corresponding to the current value of `tselect`. There are also no triggers at higher values of `tselect`; trigger enumeration terminates at this point.

4.6. 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.

4.6.1. 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.

Bits	Name	Description
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 on entry to Debug Mode:

Cause	Description
`1`	An `ebreak` instruction executed in M-mode when `dcsr.ebreakm` was set, or U-mode when `dcsr.ebreaku` was set.
`2`	A trigger fired.
`3`	The Debug Module requested a processor halt, or a reset-halt request was present when the core reset was released.
`4`	The processor executed one instruction with single-stepping enabled via `dcsr.step`.

Cause `5` (`resethaltreq`) is never set by hardware. This event is reported as a normal halt, cause `3`.

4.6.2. 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.

4.6.3. 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 [reg-dmdata0].

4.6.4. dscratch1

Address: `0x7b3`

Not implemented. Access will cause an illegal instruction exception.

4.7. Custom Debug Mode CSRs

4.7.1. h3.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.

4.8. Custom Interrupt Handling CSRs

4.8.1. h3.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 h3.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

Bits	Name	Description
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`.

4.8.2. h3.meipa

Address: `0xbe1`

External interrupt pending array. Contains a read-only bit for each external interrupt request. Similarly to `h3.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 `h3.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 `h3.meipa` bit is cleared by servicing the requester so that it deasserts its interrupt request.

When any interrupt of sufficient priority is both set in `h3.meipa` and enabled in `h3.meiea`, the standard RISC-V external interrupt pending bit `mip.meip` is asserted. In other words, `h3.meipa` is filtered by `h3.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 `h3.meipa`
- The matching interrupt enable bit is set in `h3.meiea`
- The interrupt priority is greater than or equal to the preemption priority in `h3.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`.

4.8.3. h3.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 `h3.meipa`. Software can use this feature to manually trigger a particular interrupt.

There are no restrictions on using `h3.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 `h3.meinext` which returns the corresponding IRQ number in `h3.meinext.irq` (no matter whether `h3.meinext.update` is written).

`h3.meifa` implements the same array window indexing scheme as `h3.meiea` and `h3.meipa`.

Bits	Name	Description
31:16	`window`	16-bit read/write window into the external interrupt force 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`.

4.8.4. h3.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 `h3.meicontext.preempt`, it is treated as not being pending. The pending bit in `h3.meipa` will still assert, but the machine external interrupt pending bit `mip.meip` will not, so the processor will ignore this interrupt. See h3.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`.

4.8.5. h3.meinext

Address: `0xbe4`

Get next interrupt. Contains the index of the highest-priority external interrupt which is both asserted in `h3.meipa` and enabled in `h3.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 `h3.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.
1	-	RES0
0	`update`	Writing 1 (self-clearing) causes hardware to update `h3.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`.

4.8.6. h3.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 `h3.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 arbitrary 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 `h3.meicontext` is saved/restored correctly, `preempt` and `ppreempt` stack correctly through arbitrarily many preemption frames.

Bits	Name	Description
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 `h3.meinext`, so that a preemptee is not re-taken in the preempting frame.
23:21	**-**	RES0
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 `h3.meinext.update` is written, or when hardware enters the external interrupt vector. If an interrupt is present in `h3.meinext`, then `preempt` is set to one level greater than that interrupt's priority. Otherwise, `preempt` 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 `h3.meinext.noirq` when `h3.meinext.update` is written. No hardware effect.
14:13	**-**	RES0
12:4	`irq`	Current IRQ number (read/write). Set to `h3.meinext.irq` when `h3.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.

Bits	Name	Description
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 `h3.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 `h3.meicontext` and `h3.meinext`:

```
isr_external_irq:
    // Save caller saves and exception return state whilst IRQs are disabled.
    // We can't be preempted during this time, but if a higher-priority IRQ
    // arrives ("late arrival"), that will be the one displayed in h3.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, h3.meicontext, 0x2
    sw a0, 68(sp)
    csrr a0, mstatus
    sw a0, 72(sp)

    j get_next_irq

dispatch_irq:
    // Preemption priority was configured by h3.meinext update, so enable preemption:
    csrsi mstatus, 0x8
    // h3.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
    // h3.meinext, and write 1 to the LSB to tell hardware to tell hw to update
    // h3.meicontext with the preemption priority (and IRQ number) of this IRQ
```

```
    csrrsi a0, h3.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 h3.meicontext, a0
    lw a0, 72(sp)
    csrw mstatus, a0

    lw ra,  0(sp)
    ... snip
    lw t6, 60(sp)
    addi sp, sp, 80
    mret
```

4.9. Custom Memory Protection CSRs

4.9.1. h3.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.

4.10. Custom Power Control CSRs

4.10.1. h3.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.
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.

4.11. Custom Identification CSRs

4.11.1. h3.misa

Address: `0xbf1`

M-mode ISA identification register.

The `h3.misa` register provides a simple list of which standard RISC-V ISA extensions are supported. It is conceptually similar to the standard misa register, but stores a larger bit array, sufficient to enumerate all standard extensions. This register comprises the entirety of the Xh3misa extension, as described in Xh3misa: Hazard3 ISA identification register.

The bit assignment is provided by the RISC-V C API documentation here: https://github.com/riscv-non-isa/riscv-c-api-doc/blob/main/src/c-api.adoc#extension-bitmask-definitions. The latest assignment can always be found in the upstream documentation, but at time of writing the following bits are assigned:

Extension	groupid	bit position
A	0	0
B	0	1
C	0	2
D	0	3
E	0	4
F	0	5
H	0	7
I	0	8
M	0	12
Q	0	16
V	0	21
Zacas	0	26
Zba	0	27
Zbb	0	28
Zbc	0	29
Zbkb	0	30
Zbkc	0	31
Zbkx	0	32
Zbs	0	33
Zfa	0	34
Zfh	0	35
Zfhmin	0	36
Zicboz	0	37
Zicond	0	38
Zihintntl	0	39
Zihintpause	0	40
Zknd	0	41
Zkne	0	42
Zknh	0	43
Zksed	0	44
Zksh	0	45
Zkt	0	46
Ztso	0	47

Extension	groupid	bit position
Zvbb	0	48
Zvbc	0	49
Zvfh	0	50
Zvfhmin	0	51
Zvkb	0	52
Zvkg	0	53
Zvkned	0	54
Zvknha	0	55
Zvknhb	0	56
Zvksed	0	57
Zvksh	0	58
Zvkt	0	59
Zve32x	0	60
Zve32f	0	61
Zve64x	0	62
Zve64f	0	63
Zve64d	1	0
Zimop	1	1
Zca	1	2
Zcb	1	3
Zcd	1	4
Zcf	1	5
Zcmop	1	6
Zawrs	1	7
Zilsd	1	8
Zclsd	1	9
Zcmp	1	10
Zifencei	1	11
Zmmul	1	12

The bit index within the `h3.misa` bitmap can be calculated as `64 * groupid + bit position`. A read from h3.misa returns a 32-bit slice of the bitmap, indexed by the write data of the same CSR instruction. For example, Zcmp is at bit index 74, which is bit 10 of word 2.

```
// Get bits 95:64 of the ISA bitmap:
```

```
csrrwi a0, h3.misa, 2
// Get bit 10 of the result:
bexti a0, a0, 10
// a0 is 1 if Zcmp is supported.
```

The implementation is not required to decode all bits of the index. The length of the bitmap can be checked by reading from the special index `0x400`. Software should ignore data from bit indices higher than the indicated bitmap length.

The following is an example of testing for an extension based on the `groupid` and `bit_position` from the RISC-V C API documentation:

```
uint32_t h3_misa_read(uint32_t index) {
    uint32_t ret;
    asm (
        "csrrw %0, 0xbf1, %1\n"
        : "=r" (ret)
        : "r" (index)
    );
    return ret;
}

bool h3_misa_extension_supported(unsigned int groupid, unsigned int bit_position) {
    unsigned int index = groupid * 64u + bit_position;
    if (index >= h3_misa_read(0x400u)) {
        return false;
    }
    return h3_misa_read(index >> 5) & (1u << (index & 0x1f));
}
```

Index `0x401` contains the length in *words* of the custom extension listing, which starts from index `0x500`.

Indices `0x500` onward contain versions of custom Hazard3 extensions:

Word index	Extension
`0x500`	Xh3misa
`0x501`	Xh3irq
`0x502`	Xh3pmpm
`0x503`	Xh3power
`0x504`	Xh3bextm

The custom extension entries have the same `major.minor.patch` format as mimpid:

Bits	Name	Description
31:28	**-**	RES0

Bits	Name	Description
27:24	`major`	Major release version, i.e. the `1` in `v1.2.3`
23:16	`minor`	Minor release version, i.e. the `2` in `v1.2.3`
15:8	`patch`	Patch version, i.e. the `3` in `v1.2.3`
7:0	**-**	RES0

A value of all-zeroes indicates the custom extension is not implemented.

Chapter 5. Custom Extensions

Hazard3 implements a small number of custom extensions. Xh3misa is always included; the remaining custom extensions are only included if the relevant feature flags are set to 1 when instantiating the processor (Configuration Parameters).

The x bit in misa is always set, to indicate (at least) the presence of the Xh3misa extension.

5.1. Xh3misa: Hazard3 ISA identification register

This document describes Xh3misa version: 1.0

This extension is enabled by: CSR_M_MANDATORY.

This extension adds the h3.misa CSR. This is an extended version of the standard misa register, capable of enumerating all standard RISC-V extensions (not just the single-letter ones).

It describes presence and absence of extensions using the bitmap format described in the RISC-V C API documentation:

https://github.com/riscv-non-isa/riscv-c-api-doc/blob/main/src/c-api.adoc#extension-bitmask-definitions

This register also lists the versions of any custom Hazard3 extensions which may be present, including Xh3misa itself. For more details see the description of the h3.misa register.

5.2. Xh3irq: Hazard3 interrupt controller

This document describes Xh3irq version: 1.0

This extension is enabled by: EXTENSION_XH3IRQ

This lightweight extension controls up to 512 external interrupts, with up to 16 levels of preemption. It adds no new instructions, but several CSRs:

- h3.meiea
- h3.meipa
- h3.meifa
- h3.meipra
- h3.meinext
- h3.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.

5.3. Xh3pmpm: M-mode PMP regions

This document describes Xh3pmpm version: 1.0

This extension is enabled by: EXTENSION_XH3PMPM

This extension adds a new M-mode CSR, h3.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.

5.4. Xh3power: Hazard3 power management

This document describes Xh3power version: 1.0

This extension is enabled by: EXTENSION_XH3POWER

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

The `h3.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 `h3.msleep` appropriately, the processor can gate its own clock when asleep or, with a simple 4-phase req/ack handshake, negotiate with an external power controller for power up/down of external hardware. 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 `h3.msleep`.

5.4.1. 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 implementer) 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
```

5.4.2. 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
```

5.5. Xh3bextm: Hazard3 bit extract multiple

This document describes Xh3bextm version: 1.0

This extension is enabled by: EXTENSION_XH3BEXTM

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

5.5.1. 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`	`0b00010`	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
```

5.5.2. 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`	`0b00010`	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 6. Debug

Hazard3 implements version 0.13.2 of the RISC-V debug specification, including:

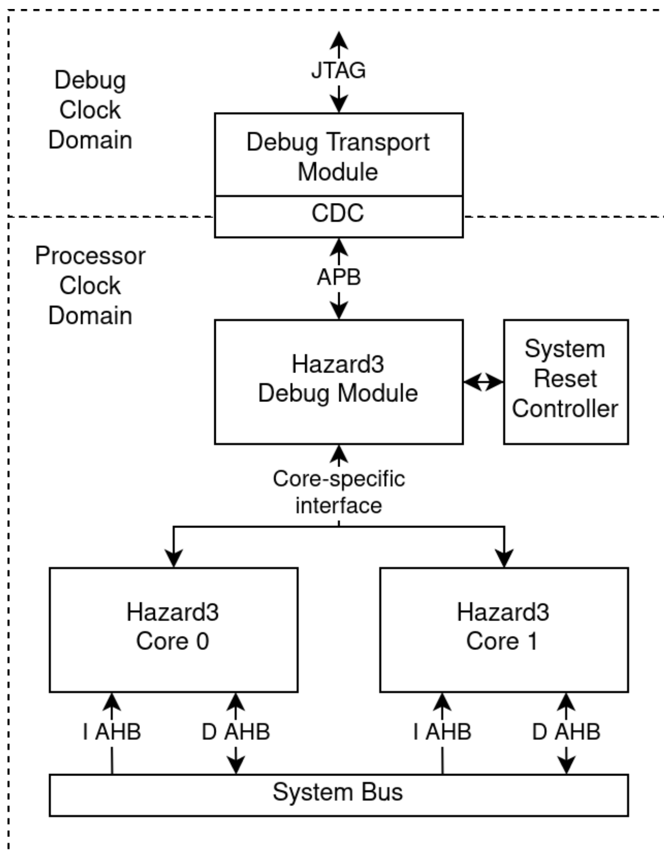
- Run/halt/reset control (including halt-on-reset for each hart)
- Abstract GPR access
- Program Buffer: 2 words plus `impebreak`
- Automatic trigger of abstract command (`abstractauto`) on `data0` or Program Buffer access for efficient memory block transfers from the host
- Support for multiple harts (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
- A trigger unit with exception, interrupt and instruction count triggers
  - Exception and interrupt triggers are hardwired with `action` = `1` (break to Debug mode only)
  - Instruction count trigger is hardwired with `action` = `0` and `count` = `1` (provides single-stepping of U-mode or M-mode from an M-mode exception handler)
  - The `tcontrol.mte` and `mpte` bits prevent trigger loops under M-mode trigger usage
- (Optional) Instruction address match triggers (hardware breakpoints)
  - Usable from both Debug mode and M-mode

6.1. 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 is implemented by the `hazard3_dm` module, found in hdl/debug/dm/hazard3_dm.v. 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 manager could be used. The DM requires at least 7 bits of word addressing, i.e. 9 bits of byte address space. The top-level module for Hazard3's JTAG-DTM implementation is `hazard3_jtag_dtm`, found in hdl/debug/dtm/hazard3_jtag_dtm.v.

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.

6.2. Trigger Module

Hazard3's Trigger Module generates synchronous traps to either Debug Mode or M-mode when certain pre-programmed conditions are met. It support four types of trigger: instruction address (breakpoint), interrupt, exception, and instruction count. The number of instruction address match triggers is configurable, and the remaining trigger types have one instance each.

Use the following CSRs to configure individual triggers:

- tselect: Choose a trigger to configure. Implements exactly enough bits to index all implemented triggers; remaining bits are hardwired to 0
- tinfo: List the capabilities of the currently selected trigger (the supported values for `tdata1.type`)
- tdata1: Configure the trigger. Layout changes based on the value of `tdata1.type`
- tdata2: Further configuration, or address/data to match on
- tdata3: Supports filtering of triggers based on context — not implemented on Hazard3, so hardwired to zero

A compliant debugger can enumerate the triggers on a particular Hazard3 implementation using the standard method: incrementing `tselect` from 0, checking `tinfo` for each `tselect` index, and terminating when `tinfo` is 0 or `tselect` wraps.

Hazard3 implements only a single trigger type for each `tselect` index. Therefore `tinfo` has only one bit set, and the `tdata1.type` field for each trigger is read-only, matching the bit set in `tinfo`.

Most of Hazard3's triggers support a programmable `action` field, which determines whether the trigger targets M-mode (`action` = `0`) or Debug Mode (`action` = `1`).

M-mode triggers generate an `ebreak` exception (`mcause` = `3`) with `mepc` pointing to the instruction which generated the trigger condition. This provides hardware support for self-hosted debuggers, such as an M-mode operating system supporting debug of U-mode applications on the same core.

Debug Mode triggers enter Debug Mode with `dpc` pointing to the instruction which generated the trigger condition. The core ceases to execute instructions autonomously, and its internals become accessible to the external debug host.

An `action` of `1` is ignored when that trigger's `tdata1.dmode` bit is clear. Only Debug Mode can modify this bit, and setting this bit prevents other modes from modifying the trigger's configuration. This effectively claims the trigger for external debugger use, and makes it unavailable to self-hosted debug software.

6.2.1. Instruction Address Triggers

The BREAKPOINT_TRIGGERS parameter configures the trigger unit with 0 or more hardware breakpoint comparators, assigned tselect indices `0` through `BREAKPOINT_TRIGGERS - 1`. Each comparator continuously monitors the program counter for an exact match for its programmed address. When the comparator matches, the core takes a trap with `dpc` or `mepc` equal to the programmed breakpoint address.

In standard RISC-V terms, these are instruction address match triggers, with the following characteristics as described by hardwired fields in `mcontrol` (aka tdata1):

- `tdata1.type` = `2`: address/data match
- `mcontrol.timing` = `0`: trigger fires before the instruction executes
- `mcontrol.select` = `0`: match on address (not data)
- `mcontrol.match` = `0`: match on exact address
- `mcontrol.store` = `load` = `0`: load/store address comparison not supported

The `mcontrol.u` and `mcontrol.m` bits are fully implemented, and determine in which privilege modes the trigger will fire.

The `mcontrol.action` field can be programmed with the value `0` or `1`, indicating a break to M-mode or D-mode respectively. D-mode breaks are taken only when `mcontrol.dmode` is set.

`tdata2` contains the address to be matched on. The LSB of this register is hardwired to zero, as the RISC-V program counter never contains odd addresses. When compressed instruction support is disabled (EXTENSION_C is `0`), bit `1` is also hardwired to zero.

The `mcontrol.execute` flag enables and disables the trigger (1/0).

6.2.2. Instruction Count Trigger

Hazard3 implements a single instruction count trigger, assigned tselect index `BREAKPOINT_TRIGGERS`. For example, if no breakpoint triggers are configured, it is accessed at `tselect` = `0`.

This trigger is hardwired with a `count` of `1`, and is intended to be a lightweight feature to support self-hosted M-mode single-stepping. Debug Mode already has this capability in the form of the `dcsr.step` flag, so the instruction count trigger supports `action` = `0` only (break to M-mode).

The following configuration bits in tdata1 are *not* hardwired:

- `icount.m`: if `1`, enable the trigger in M-mode
- `icount.u`: if `1`, enable the trigger in U-mode (only available when the U_MODE parameter is `1`)

The trigger fires after one instruction successfully executes. `mepc` points to the next instruction in program order. Since `icount.count` is hardwired to `1`, hardware clears both the `m` and `u` flags after the trigger fires, to avoid repeat fires.

IMPORTANT	As a deliberate departure from the 0.13.2 specification, the trigger does not fire on exceptions, because the resulting breakpoint exception would make the original exception unrecoverable and impossible to diagnose. To handle an exception occurring during single-stepping, catch the exception directly.

To single-step a U-mode context from M-mode, set `icount.u` before returning to U-mode via `mret`. The U-mode context will trap back out after executing one instruction.

To single-step an M-mode target, clear `tcontrol.mte`, then set `tcontrol.mpte` and `icount.m` before

returning to an M-mode context via `mret`. The trigger is enabled after the return completes (via `mpte` shifting to `mte`), and the core traps back to the M-mode handler after executing one instruction in the M-mode returnee. The handler executes normally, because `mte` is cleared by the trap, and `icount.m` is cleared by the trigger match.

To support the above behaviour, hardware does not clear `icount.m` or `icount.u` while `tcontrol.mte` is clear. See discussion here.

6.2.3. Interrupt Trigger

Hazard3 implements a single interrupt trigger, assigned tselect index `BREAKPOINT_TRIGGERS + 1`. For example, if no breakpoint triggers are configured, it is accessed at `tselect` = `1`.

This trigger fires on a configurable subset of the interrupt cause values listed in the mcause register description. `tdata2` contains a bitmap with one bit per cause value. The break is taken after the hardware interrupt entry sequence has taken place (e.g. the setting of `mcause`), but before the first instruction in the interrupt handler executes. `dpc` points to the first instruction in the interrupt handler.

`action` is hardwired to `1` (Debug Mode entry), because all interrupts on Hazard3 target M-mode, so a breakpoint exception triggered by interrupt entry would make it impossible to return from the interrupt.

The implemented bits are `11`, `7` and `3`, corresponding to `mip.meip`, `mip.mtip` and `mip.msip` respectively.

6.2.4. Exception Trigger

Hazard3 implements a single exception trigger, assigned tselect index `BREAKPOINT_TRIGGERS + 2`. For example, if no breakpoint triggers are configured, it is accessed at `tselect` = `2`.

This trigger fires on a configurable subset of the exception cause values listed in the mcause register description. `tdata2` contains a bitmap with one bit per cause value. The break is taken after the hardware exception entry sequence has taken place (e.g. the setting of `mcause`) but before the first instruction in the exception handler executes. `dpc` points to the first instruction in the exception handler.

`action` is hardwired to `1` (Debug Mode entry) because all exceptions on Hazard3 target M-mode, so a breakpoint exception triggered by another exception would always make the original exception unrecoverable.

Bits corresponding to causes impossible on a given Hazard3 implementation are hardwired to zero. For example, bit `0` (`mcause` = `0`, fetch alignment exception) is not implemented when EXTENSION_C is `1`, because there are no fetch alignment exceptions if compressed instructions are implemented.

Bit `3` (`ebreak`) is also unimplemented. To catch `ebreak` execution from an external debugger, use the `dcsr.ebreakm` and `dcsr.ebreaku` flags.

6.3. 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`, `progbuf0` and `progbuf1` registers
- Up to 32 harts selectable via `hartsel`
- Hart array mask selection
- System bus access (optional)

Not implemented:

- Abstract access memory
- Abstract access CSR
- Post-incrementing abstract access GPR

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, h3.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` supports the values `3` (M-mode) and `0` (U-mode). If U-mode support is not configured, it is hardwired to `3`.

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 transfer 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 supports 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` CSR is configured using the `mhartid_val` input port on that core. In a correctly configured system, each core must have a unique value, and one core has the value of all-zeroes.

6.4. 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. Base Instruction Set (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. Integer Multiply and Divide (M)

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. Atomics (A)

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. Compressed Instructions (C or Zca)

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[5]	
`csrrc rd, csr, rs1`	1[5]	
`csrrs rd, csr, rs1`	1[5]	
`csrrwi rd, csr, imm`	1[5]	
`csrrci rd, csr, imm`	1[5]	
`csrrsi rd, csr, imm`	1[5]	
Trap Request		

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

A.6. Load/Store Pair (Zilsd)

Instruction	Cycles	Note
Load/Store Pair		
`ld rd, imm(rs1)`	2	Issues as two `lw`
`sd rs2, imm(rs1)`	2	Issues as two `sw`

A.7. Bit Manipulation (Zba, Zbb, Zbc, Zbs)

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	

Instruction	Cycles	Note
`zext.h rd, rs1`	1	
`zext.b rd, rs1`	1	`zext.b` is a pseudo-op for `andi rd, rs1, 0xff`
Zbc (carry-less multiply)		
`clmul rd, rs1, rs2`	1	
`clmulh rd, rs1, rs2`	1	
`clmulr rd, rs1, rs2`	1	
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.8. Additional Basic Compressed Instructions (Zcb)

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.9. Compressed Load/Store Pair (Zclsd)

These 16-bit instructions behave identically to their 32-bit counterparts, namely:

- `c.ld` expands to Zilsd `ld`

* `c.ldsp` expands to Zilsd `ld`
* `c.sd` expands to Zilsd `sd`
* `c.sdsp` expands to Zilsd `sd`

A.10. Push, Pop and Double Move (Zcmp)

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)[6] or 2 + *n* (*n* >= 2)[1]	*n* is number of registers in rlist
`cm.popretz {rlist}, imm`	3 + *n*[1]	*n* is number of registers in rlist
`cm.mva01s r1s', r2s'`	2	
`cm.mvsa01 r1s', r2s'`	2	

A.11. 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] Writes to the following CSRs take 3 cycles, plus an additional 1 cycle if the following instruction is misaligned 32-bit: `pmpaddr*`, `pmpcfg*`, `tcontrol`, `tdata1`, `tdata2`. These CSRs require an instruction fetch flush to enforce CSR-write-to-fetch ordering.

[6] The single-register variant of `cm.popret` takes the same number of cycles as the two-register variant, because of an internal load-use dependency on the loaded return address.

Appendix B: Release Notes

Version 1.1

Incompatible changes:

- The `MHARTID_VAL` parameter has been removed, and replaced with the `mhartid_val` port.
- The `MIMPID_VAL` parameter has been removed.
- The mimpid CSR has been redefined to contain the hardcoded Hazard3 release version.

New signals:

- `fence_i_vld`, `fence_d_vld` and `fence_rdy` allow external hardware to enforce memory orderings or trigger cache flushes (see Memory Ordering Signals).
- `mhartid_val` allows a dynamic `mhartid` CSR value.
- `eco_version` allows modifying part of the processor's reported version number (see mimpid).

New features:

- Implement the Zilsd extension (load/store pair), enabled by EXTENSION_ZILSD.
- Implement the Zclsd extension (compressed load/store pair), enabled by EXTENSION_ZCLSD.
- Implement the Zbkx extension (crossbar permutation), enabled by EXTENSION_ZBKX.
- Implement TOR region support for PMP, enabled by PMP_MATCH_TOR.
  - NAPOT/NA4 matching is now optional, enabled by PMP_MATCH_NAPOT.
- Implement interrupt, exception and instruction count trigger in the Trigger Module.
  - These triggers are present whenever debug support is enabled by DEBUG_SUPPORT.
- Implement the RV32E base extension, enabled by EXTENSION_E.
- Implement new custom extension: Xh3misa: Hazard3 ISA identification register, for detecting processor ISA support at finer granularity than the standard `misa` register. Enabled by `CSR_M_MANDATORY`, so it is reliably present.
- Add `hazard3_xilinx7_jtag_dtm` for tunneling RISC-V debug through Xilinx 7-series FPGA TAPs. * Fixed errata:
- Fix `fence.i` only ordering against the address phase of a preceding store (a multi-manager AHB fabric may reorder address phases, so order against the data phase instead).
- Fix local monitor flag being cleared by trap entry as well as trap exit (only `xRET` should clear it).
  - This behaviour continues to not affect entry/exit to Debug Mode, so LR/SC sequences can be single-stepped.
- Fix the Debug Module PROGBUF1 register being writable when an abstract command is executing, and not setting `abstracts.cmderr` to `busy` when written.
- Fix spurious dependency of CSR instructions on `rs2`, causing read-after-write stall.

- Fix processor lockup on AMOs that fail PMP checks.
- Fix Debug Mode loads and stores during Program Buffer execution being subject to PMP checks on M-mode-enforced (locked) regions.
- Fix a data-phase fault on the final load of a `cm.popret` which loads at least two registers reporting the return address as the exception PC.
    - The following were not affected: `cm.pop`, `cm.popretz`, 1-register `cm.popret`, PMP traps and alignment traps.

PPA optimisations:

- Move breakpoint PC comparator from stage 2 to stage 1.
- Move PMP X permission lookup from stage 2 to stage 1.
- Simplify PMP X permission lookup by looking up naturally-aligned fetches instead of both halfwords of a potentially unaligned instruction.
- Move compressed instruction expansion from stage 2 to stage 1.
- Pre-decode `rs1`/`rs2` bypass controls in stage 1.
- Restore register on `op_b` input of `hazard3_muldiv_seq` for better routing locality.

General improvements:

- Relax PMP X permission checking to allow instructions to straddle two PMP regions which both have the necessary permissions.
- RTL is now lint-clean with Verilator v5.038.
- Add Verilator testbench, command-line-compatible with existing CXXRTL testbench.
- Update to latest versions of `riscv-arch-test`, `riscv-test` and `riscv-formal`.
- Enable `riscv-formal` checks for B extension and other bit manipulation extensions.
- Update `hazard3_rvfi_monitor` to support running existing `riscv-formal` checks with the A, Zcmp, Zilsd and Zclsd extensions enabled.
- Allow synthesising the processor with the RVFI trace monitor included by defining the `HAZARD3_RVFI_STANDALONE` macro.
    - This is an intermediate step towards E-trace support, and useful for tracing processor execution on FPGA or in simulation.

Other changes:

- Writes to CSRs which affect instruction fetch now cost three cycles plus an additional cycle if the following instruction is unaligned 32-bit.
    - Specifically this affects `pmpaddr*`, `pmpcfg*`, `tcontrol`, `tdata1` and `tdata2`.
    - These CSR writes now require an instruction fetch flush to enforce the necessary orderings.
    - All other CSRs remain single-cycle write.
- Writes to `minstret`/`minstreth` or `mcycle`/`mcycleh` now inhibit the increment of the entire underlying 64-bit counter.

    ◦ The spec wording was recently changed to clarify this issue: see here.

See the Github release notes here.

Version 1.0.2

- Coding style change in `hazard3_frontend` for Verilator compatibility

See the Github release notes here.

Version 1.0.1

- Fix: abstract access commands initiated via `abstractauto` access the wrong core GPR

See the Github release notes here.

Version 1.0

This is the first stable version of Hazard3. See the Github release notes here.