Hazard3

Table of Contents

Chapter 1. Introduction

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

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

The following are planned for future implementation:

- Trigger unit for debug mode
    - Likely breakpoints only

Chapter 2. Instruction Cycle Counts

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

2.1. RV32I

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

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	

2.2. M Extension

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

Instruction	Cycles	Note
32 × 32 → 32 Multiply		
`mul rd, rs1, rs2`	1	
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	

2.3. A Extension

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

2.4. C Extension

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

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

2.5. Privileged Instructions (including Zicsr)

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

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

2.6. Bit Manipulation

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

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

2.7. 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.

Chapter 3. Instruction Pseudocode

This section is a quick reference for the operation of the instructions supported by Hazard3, in Verilog syntax. Conventions used in this section:

- `rs1`, `rs2` and `rd` are 32-bit unsigned vector variables referring to the two register operands and the destination register
- `imm` is a 32-bit unsigned vector referring to the instruction's immediate value
- `pc` is a 32-bit unsigned vector referring to the program counter
- `mem` is an array of 8-bit unsigned vectors, each corresponding to a byte address in memory.

3.1. RV32I: Register-register

With the exception of the shift instructions, all instructions in this section have an immediate range of -2048 to 2047. Negative immediates can be useful for the bitwise operations too: for example `not rd, rs1` is a pseudo-op for `xori rd, rs1, -1`.

Shift instructions have an immediate range of 0 to 31.

3.1.1. add

Add register to register.

Syntax:

```
add rd, rs1, rs2
```

Operation:

```
rd = rs1 + rs2;
```

3.1.2. sub

Subtract register from register.

Syntax:

```
sub rd, rs1, rs2
```

Operation:

```
rd = rs1 - rs2;
```

3.1.3. slt

Set if less than (signed).

Syntax:

```
slt rd, rs1, rs2
```

Operation:

```
rd = $signed(rs1) < $signed(rs2);
```

3.1.4. sltu

Set if less than (unsigned).

Syntax:

```
sltu rd, rs1, rs
```

Operation:

```
rd = rs1 < rs2;
```

3.1.5. and

Bitwise AND.

Syntax:

```
and rd, rs1, rs2
```

Operation:

```
rd = rs1 & rs2;
```

3.1.6. or

Bitwise OR.

Syntax:

```
or rd, rs1, rs2`
```

Operation:

```
rd = rs1 | rs2;
```

3.1.7. xor

Bitwise XOR.

Syntax:

```
xor rd, rs1, rs2
```

Operation:

```
rd = rs1 ^ rs2;
```

3.1.8. sll

Shift left, logical.

Syntax:

```
sll rd, rs1, rs2
```

Operation:

```
rd = rs1 << rs2;
```

3.1.9. srl

Shift right, logical.

Syntax:

```
srl rd, rs1, rs2
```

Operation:

```
rd = rs1 >> rs2;
```

3.1.10. sra

Shift right, arithmetic.

Syntax:

```
sra rd, rs1, rs2
```

Operation:

```
rd = rs1 >>> rs2;
```

3.2. RV32I: Register-immediate

3.2.1. addi

Add register to immediate.

Syntax:

```
addi rd, rs1, imm
```

Operation:

```
rd = rs1 + imm
```

3.2.2. slti

Set if less than immediate (signed).

Syntax:

```
slti rd, rs1, imm
```

Operation:

```
rd = $signed(rs1) < $signed(imm);
```

3.2.3. sltiu

Set if less than immediate (unsigned).

Syntax:

```
sltiu rd, rs1, imm
```

Operation:

```
rd = rs1 < imm;
```

3.2.4. andi

Bitwise AND with immediate.

Syntax:

```
andi rd, rs1, imm
```

Operation:

```
rd = rs1 & imm;
```

3.2.5. ori

Bitwise OR with immediate.

Syntax:

```
ori rd, rs1, imm
```

Operation:

```
rd = rs1 \| imm;
```

3.2.6. xori

Bitwise XOR with immediate.

Syntax:

```
xori rd, rs1, imm
```

Operation:

```
rd = rs1 ^ imm;
```

3.2.7. slli

Shift left, logical, immediate.

Syntax:

```
slli rd, rs1, imm
```

Operation:

```
rd = rs1 << imm;
```

3.2.8. srli

Shift right, logical, immediate.

Syntax:

```
srli rd, rs1, imm
```

Operation:

```
rd = rs1 >> imm;
```

3.2.9. srai

Shift right, arithmetic, immediate.

Syntax:

```
srai rd, rs1, imm
```

Operation:

```
rd = rs1 >>> imm;
```

3.3. RV32I: Large immediate

3.3.1. lui

Load upper immediate.

Syntax:

```
lui rd, imm
```

Operation:

```
rd = imm;
```

(`imm` is a 20-bit value followed by 12 zeroes)

3.3.2. auipc

Add upper immediate to program counter.

Syntax:

```
auipc rd, imm
```

Operation:

```
rd = pc + imm;
```

(`imm` is a 20-bit value followed by 12 zeroes)

3.4. RV32I: Control transfer

3.4.1. jal

Jump and link.

Syntax:

```
jal rd, label
j label        // rd is implicitly x0
```

Operation:

```
rd = pc + 4;
pc = label;
```

NOTE	the 16-bit variant, `c.jal`, writes `pc + 2` to `rd`, rather than `pc + 4`. The `rd` value always points to the sequentially-next instruction.

3.4.2. jalr

Jump and link, target is register.

Syntax:

```
jalr rd, rs1, imm // imm is implicitly 0 if omitted.
jr rs1, imm       // rd is implicitly x0. imm is implicitly 0 if omitted.
ret               // pseudo-op for jr ra
```

Operation:

```
rd = pc + 4;
pc = rs1 + imm;
```

NOTE the 16-bit variant, `c.jalr`, writes `pc + 2` to `rd`, rather than `pc + 4`. The `rd` value always points to the sequentially-next instruction.

3.4.3. beq

Branch if equal.

Syntax:

```
beq rs1, rs2, label
```

Operation:

```
if (rs1 == rs2)
    pc = label;
```

3.4.4. bne

Branch if not equal.

Syntax:

```
bne rs1, rs2, label
```

Operation:

```
if (rs1 != rs2)
    pc = label;
```

3.4.5. blt

Branch if less than (signed).

Syntax:

```
blt rs1, rs2, label
```

Operation:

```
if ($signed(rs1) < $signed(rs2))
    pc = label;
```

3.4.6. bge

Branch if greater than or equal (signed).

Syntax:

```
bge rs1, rs2, label
```

Operation:

```
if ($signed(rs1) >= $signed(rs2))
    pc = label;
```

3.4.7. bltu

Branch if less than (unsigned).

Syntax:

```
bltu rs1, rs2, label
```

Operation:

```
if (rs1 < rs2)
    pc = label;
```

3.4.8. bgeu

Branch if less than or equal (unsigned).

Syntax:

```
bgeu rs1, rs2, label
```

Operation:

```
if (rs1 >= rs2)
    pc = label;
```

3.5. RV32I: Load and Store

3.5.1. lw

Load word.

Syntax:

```
lw rd, imm(rs1)
lw rd, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
rd = {
    mem[rs1 + imm + 3],
    mem[rs1 + imm + 2],
    mem[rs1 + imm + 1],
    mem[rs1 + imm]
};
```

3.5.2. lh

Load halfword (signed).

Syntax:

```
lh rd, imm(rs1)
lh rd, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
rd = {
    {16{mem[rs1 + imm + 1][7]}}, // Sign-extend
    mem[rs1 + imm + 1],
```

```
    mem[rs1 + imm]
};
```

3.5.3. lhu

Load halfword (unsigned).

Syntax:

```
lhu rd, imm(rs1)
lhu rd, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
rd = {
    16'h0000, // Zero-extend
    mem[rs1 + imm + 1],
    mem[rs1 + imm]
};
```

3.5.4. lb

Load byte (signed).

Syntax:

```
lb rd, imm(rs1)
   lb rd, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
rd = {
    {24{mem[rs1 + imm][7]}}, // Sign-extend
    mem[rs1 + imm]
};
```

3.5.5. lbu

Load byte (unsigned).

Syntax:

```
lbu rd, imm(rs1)
   lbu rd, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
rd = {
    24'h000000, // Zero-extend
    mem[rs1 + imm]
};
```

3.5.6. sw

Store word.

Syntax:

```
sw rs2, imm(rs1)
sw rs2, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
mem[rs1 + imm]     = rs2[7:0];
mem[rs1 + imm + 1] = rs2[15:8];
mem[rs1 + imm + 2] = rs2[23:16];
mem[rs1 + imm + 3] = rs2[31:24];
```

3.5.7. sh

Store halfword.

Syntax:

```
sh rs2, imm(rs1)
sh rs2, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
mem[rs1 + imm]     = rs2[7:0];
mem[rs1 + imm + 1] = rs2[15:8];
```

3.5.8. sb

Store byte.

Syntax:

```
sb rs2, imm(rs1)
```

```
sb rs2, (rs1)    // imm is implicitly 0 if omitted.
```

Operation:

```
mem[rs1 + imm] = rs2[7:0];
```

3.6. M Extension

3.6.1. mul

Multiply 32 × 32 → 32.

Syntax:

```
mul rd, rs1, rs2
```

Operation:

```
rd = rs1 * rs2;
```

3.6.2. mulh

Multiply signed (32) by signed (32), return upper 32 bits of the 64-bit result.

Syntax:

```
mulh rd, rs1, rs2
```

Operation:

```
// Both operands are sign-extended to 64 bits:
wire [63:0] result_full = {{32{rs1[31]}}, rs1} * {{32{rs2[31]}}, rs2};
rd = result_full[63:32];
```

3.6.3. mulhsu

Multiply signed (32) by unsigned (32), return upper 32 bits of the 64-bit result.

Syntax:

```
mulhsu rd, rs1, rs2
```

Operation:

```
// rs1 is sign-extended, rs2 is zero-extended:
wire [63:0] result_full = {{32{rs1[31]}, rs1} * {32'h00000000, rs2};
rd = result_full[63:32];
```

3.6.4. mulhu

Multiply unsigned (32) by unsigned (32), return upper 32 bits of the 64-bit result.

Syntax:

```
mulhu rd, rs1, rs2
```

Operation:

```
wire [63:0] result_full = {32'h00000000, rs1} * {32'h00000000, rs2};
rd = result_full[63:32];
```

3.6.5. div

Divide (signed).

Syntax:

```
div rd, rs1, rs2
```

Operation:

```
if (rs2 == 32'h0)
    rd = 32'hffffffff;
else if (rs1 == 32'h80000000 && rs2 == 32'hffffffff) // Signed overflow
    rd = 32'h80000000;
else
    rd = $signed(rs1) / $signed(rs2);
```

3.6.6. divu

Divide (unsigned).

Syntax:

```
divu rd, rs1, rs2
```

Operation:

```
if (rs2 == 32'h0)
    rd = 32'hffffffff;
else
    rd = rs1 / rs2;
```

3.6.7. rem

Remainder (signed).

Syntax:

```
rem rd, rs1, rs2
```

Operation:

```
if (rs2 == 32'h0)
    rd = rs1;
else
    rd = $signed(rs1) % $signed(rs2);
```

3.6.8. remu

Remainder (unsigned).

Syntax:

```
remu rd, rs1, rs2
```

Operation:

```
if (rs2 == 32'h0)
    rd = rs1;
else
    rd = rs1 % rs2;
```

3.7. A Extension

(TODO)

3.8. C Extension

All C extension instructions are 16-bit aliases of 32-bit instructions from other extensions (in the case of Hazard3, entirely from the RV32I base extension). They behave identically to their 32-bit counterparts.

3.9. Zba: Bit manipulation (address generation)

3.9.1. sh1add

Add, with the second addend shifted left by 1.

Syntax:

```
sh1add rd, rs1, rs2
```

Operation:

```
rd = rs1 + (rs2 << 1);
```

3.9.2. sh2add

Add, with the second addend shifted left by 2.

Syntax:

```
sh2add rd, rs1, rs2
```

Operation:

```
rd = rs1 + (rs2 << 2);
```

3.9.3. sh3add

Add, with the second addend shifted left by 3.

Syntax:

```
sh3add rd, rs1, rs2
```

Operation:

```
rd = rs1 + (rs2 << 3);
```

3.10. Zbb: Bit manipulation (basic)

3.10.1. andn

Bitwise AND with inverted operand.

Syntax:

```
andn rd, rs1, rs2
```

Operation:

```
rd = rs1 & ~rs2;
```

3.10.2. clz

Count leading zeroes (starting from MSB, searching LSB-ward).

Syntax:

```
clz rd, rs1
```

Operation:

```
rd = 32;          // Default = 32 if no set bits
reg found = 1'b0; // Local variable

for (i = 0; i < 32; i = i + 1) begin
    if (rs1[i] && !found) begin
        found = 1'b1;
        rd = i;
    end
end
```

3.10.3. cpop

Population count.

Syntax:

```
cpop rd, rs1
```

Operation:

```
rd = 0;
for (i = 0; i < 32; i = i + 1)
    rd = rd + rs1[i];
```

3.10.4. ctz

Count trailing zeroes (starting from LSB, searching MSB-ward).

Syntax:

```
ctz rd, rs1
```

Operation:

```
rd = 32;          // Default = 32 if no set bits
reg found = 1'b0; // Local variable

for (i = 0; i < 32; i = i + 1) begin
    if (rs1[31 - i] && !found) begin
        found = 1'b1;
        rd = i;
    end
end
```

3.10.5. max

Maximum of two values (signed).

Syntax:

```
max rd, rs1, rs2
```

Operation:

```
if ($signed(rs1) < $signed(rs2))
    rd = rs2;
else
    rd = rs1;
```

3.10.6. maxu

Maximum of two values (unsigned).

Syntax:

```
maxu rd, rs1, rs2
```

Operation:

```
if (rs1 < rs2)
    rd = rs2;
else
    rd = rs1;
```

3.10.7. min

Minimum of two values (signed).

Syntax:

```
min rd, rs1, rs2
```

Operation:

```
if ($signed(rs1) < $signed(rs2))
    rd = rd1;
else
    rd = rs2;
```

3.10.8. minu

Minimum of two values (unsigned).

Syntax:

```
minu rd, rs1, rs2
```

Operation:

```
if (rs1 < rs2)
    rd = rs1;
else
    rd = rs2;
```

3.10.9. orc.b

Or-combine of bits within each byte.

Syntax:

```
orc.b rd, rs1
```

Operation:

```
rd = {
    {8{|rs1[31:24]}},
    {8{|rs1[23:16]}},
    {8{|rs1[15:8]}},
    {8{|rs1[7:0]}}
};
```

3.10.10. orn

Bitwise OR with inverted operand.

Syntax:

```
orn rd, rs1, rs2
```

Operation:

```
rd = rs1 | ~rs2;
```

3.10.11. rev8

Reverse bytes within word.

Syntax:

```
rev8 rd, rs1
```

Operation:

```
rd = {
    rs1[7:0],
    rs1[15:8],
    rs1[23:16],
    rs1[31:24]
```

```
};
```

3.10.12. rol

Rotate left.

Syntax:

```
rol rd, rs1, rs2
```

Operation:

```
if (rs2[4:0] == 0)
    rd = rs1;
else
    rd = (rs1 << rs2[4:0]) | (rs1 >> (32 - rs2[4:0]));
```

3.10.13. ror

Rotate right.

Syntax:

```
ror rd, rs1, rs2
```

Operation:

```
if (rs2[4:0] == 0)
    rd = rs1;
else
    rd = (rs1 >> rs2[4:0]) | (rs1 << (32 - rs2[4:0]));
```

3.10.14. rori

Rotate right, immediate.

Syntax:

```
ror rd, rs1, imm
```

Operation:

```
if (imm[4:0] == 0)
```

```
    rd = rs1;
else
    rd = (rs1 >> imm[4:0]) | (rs1 << (32 - imm[4:0]));
```

3.10.15. sext.b

Sign-extend from byte.

Syntax:

```
sext.b rd, rs1
```

Operation:

```
rd = {
    {24{rs1[7]}},
    rs1[7:0]
};
```

3.10.16. sext.h

Sign-extend from halfword.

Syntax:

```
sext.h rd, rs1
```

Operation:

```
rd = {
    {16{rs1[15]}},
    rs1[15:0]
};
```

3.10.17. xnor

Bitwise XOR with inverted operand.

Syntax:

```
xnor rd, rs1, rs2
```

Operation:

```
rd = rs1 ^ ~rs2;
```

3.10.18. zext.h

Zero-extend from halfword.

Syntax:

```
zext.h rd, rs1
```

Operation:

```
rd = {
    16'h0000,
    rs1[15:0]
};
```

3.10.19. zext.b

Zero-extend from byte.

Syntax:

```
zext.b rd, rs1
```

Operation:

```
// Pseudo-op for RV32I instruction
andi rd, rs1, 0xff
```

3.11. Zbc: Bit manipulation (carry-less multiply)

Each of these three instructions returns a 32-bit slice of the following 64-bit result:

```
reg [63:0] clmul_result;

always @ (*) begin
    clmul_result = 0;
    for (i = 0; i < 32; i = i + 1) begin
        if (rs2[i])) begin
            clmul_result = clmul_result ^ ({32'h0, rs1} << i);
        end
    end
```

```
end
```

3.11.1. clmul

Carry-less multiply, low half.

Syntax:

```
clmul rd, rs1, rs2
```

Operation:

```
rd = cmul_result[31:0];
```

3.11.2. clmulh

Carry-les multiply, high half.

Syntax:

```
clmulh rd, rs1, rs2
```

Operation:

```
rd = clmul_result[63:32];
```

3.11.3. clmulr

Bit-reverse of carry-less multiply of bit-reverse.

Syntax:

```
clmulr rd, rs1, rs2
```

Operation:

```
rd = clmul_result[32:1];
```

3.12. Zbs: Bit manipulation (single-bit)

3.12.1. bclr

Clear single bit.

Syntax:

```
bclr rd, rs1, rs2
```

Operation:

```
rd = rs1 & ~(32'h1 << rs2[4:0]);
```

3.12.2. bclri

Clear single bit (immediate).

Syntax:

```
bclri rd, rs1, imm
```

Operation:

```
rd = rs1 & ~(32'h1 << imm[4:0]);
```

3.12.3. bext

Extract single bit.

Syntax:

```
bext rd, rs1, rs2
```

Operation:

```
rd = (rs1 >> rs2[4:0]) & 32'h1;
```

3.12.4. bexti

Extract single bit (immediate).

Syntax:

```
bexti rd, rs1, imm
```

Operation:

```
rd = (rs1 >> imm[4:0]) & 32'h1;
```

3.12.5. binv

Invert single bit.

Syntax:

```
binv rd, rs1, rs2
```

Operation:

```
rd = rs1 ^ (32'h1 << rs2[4:0]);
```

3.12.6. binvi

Invert single bit (immediate).

Syntax:

```
binvi rd, rs1, imm
```

Operation:

```
rd = rs1 ^ (32'h1 << imm[4:0]);
```

3.12.7. bset

Set single bit.

Syntax:

```
bset rd, rs1, rs2
```

Operation:

```
rd = rs1 | (32'h1 << rs2[4:0])
```

3.12.8. bseti

Set single bit (immediate).

Syntax:

```
bseti rd, rs1, imm
```

Operation:

```
rd = rs1 | (32'h1 << imm[4:0]);
```

3.13. Zbkb: Basic bit manipulation for cryptography

NOTE	Zbkb has a large overlap with Zbb (basic bit manipulation). This section covers only those instruction in Zbkb but not in Zbb.

3.13.1. brev8

Bit-reverse within each byte.

Syntax:

```
brev8 rd, rs1
```

Operation:

```
for (i = 0; i < 32; i = i + 8) begin
    for (j = 0; j < 8; j = j + 1) begin
        rd[i + j] = rs1[i + (7 - j)];
    end
end
```

3.13.2. pack

Pack halfwords into word.

Syntax:

```
pack rd, rs1, rs2
```

Operation:

```
rd = {
    rs2[15:0],
    rs1[15:0]
};
```

3.13.3. packh

Pack bytes into halfword.

Syntax:

```
packh rd, rs1, rs2
```

Operation:

```
rd = {
    16'h0000,
    rs2[7:0],
    rs1[7:0]
};
```

3.13.4. zip

Interleave upper/lower half of register into odd/even bits of result.

Syntax:

```
zip rd, rs1
```

Operation:

```
for (i = 0; i < 32; i = i + 2) begin
    rd[i]     = rs1[i / 2];
    rd[i + 1] = rs1[i / 2 + 16];
end
```

3.13.5. unzip

Deinterleave odd/even bits of register into upper/lower half of result.

Syntax:

```
unzip rd, rs1
```

Operation:

```
for (i = 0; i < 32; i = i + 2) begin
    rd[i / 2]      = rs1[i];
    rd[i / 2 + 16] = rs1[i + 1];
end
```

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.

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. Read-only, configurable constant.

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

4.1.4. mhartid

Address: `0xf14`

Hart identification register. Read-only, configurable constant.

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

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`	1 if the core is configured to support trap-handling, otherwise 0. Hazard3 has nonstandard CSRs to enable/disable external interrupts on a per-interrupt basis, see meiea and meipa. The `misa.x` bit must be set to indicate their presence. Hazard3 does not implement any custom instructions.
12	`m`	1 if the M extension is present, otherwise 0.
2	`c`	1 if the C extension is present, otherwise 0.
0	`a`	1 if the A extension is present, otherwise 0.

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

4.2.2. mstatush

Address: `0x310`

Hardwired to 0.

4.2.3. medeleg

Address: `0x302`

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

4.2.4. mideleg

Address: `0x303`

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

4.2.5. mie

Address: `0x304`

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

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

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

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

NOTE RISC-V reserves bits 16+ of mie/mip for platform use, which Hazard3 could use for external interrupt control. On RV32I this could only control 16 external interrupts, so Hazard3 instead adds nonstandard interrupt enable registers starting at 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 meipa) and enabled in meiea.
7	mtip	Timer interrupt pending. Level-sensitive interrupt signal from outside the core. Connected to a standard, external RISC-V 64-bit timer.
3	msip	Software interrupt pending. In spite of the name, this is not triggered by an instruction on this core, rather it is wired to an external memory-mapped register to provide a cross-hart level-sensitive doorbell interrupt.

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

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

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

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`)
7	Timer interrupt (`mip.mtip`)
11	External interrupt (`mip.meip`)

The following exception causes may be set by Hazard3 hardware:

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

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

4.2.11. mtval

Address: `0x343`

Hardwired to 0.

4.2.12. mcounteren

Address: `0x306`

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

Not to be confused with mcountinhibit.

4.3. Standard Memory Protection

4.3.1. pmpcfg0…3

Address: `0x3a0` through `0x3a3`

Unimplemented. Access will cause an illegal instruction exception.

4.3.2. pmpaddr0...15

Address: `0x3b0` through `0x3bf`

Unimplemented. Access will cause an illegal instruction exception.

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

4.5.1. tselect

Address: 0x7a0

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

4.5.2. tdata1...3

Address: 0x7a1 through 0x7a3

Unimplemented. Access will cause an illegal instruction exception.

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

Fields not mentioned above are hardwired to 0.

Hazard3 may set the following `dcsr.cause` values:

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

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

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 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. 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. meiea

Address: `0xbe0`

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

If enabled, an external interrupt can cause assertion of the standard RISC-V machine external

interrupt pending flag (mip.meip), and therefore cause the processor to enter the external interrupt vector. See meipa.

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

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

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

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

4.8.2. meipa

Address: 0xbe1

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

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

When any interrupt of sufficient priority is both set in meipa and enabled in meiea, the standard RISC-V external interrupt pending bit mip.meip is asserted. In other words, meipa is filtered by meiea to generate the standard mip.meip flag. So, an external interrupt is taken when *all* of the following are true:

- An interrupt is currently asserted in meipa
- The matching interrupt enable bit is set in meiea
- The interrupt priority is greater than or equal to the preemption priority in meicontext
- The standard M-mode interrupt enable mstatus.mie is set
- The standard M-mode global external interrupt enable mie.meie is set

In this case, the processor jumps to either:

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

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

4.8.3. meifa

Address: `0xbe2`

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

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

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

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

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

Address: `0xbe3`

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

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

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

4.8.5. meinext

Address: `0xbe4`

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

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

Bits	Name	Description
31	**noirq**	Set when there is no external interrupt which is enabled, pending, and has sufficient priority. Can be efficiently tested with a `bltz` or `bgez` instruction.
30:11	**-**	RES0
10:2	**irq**	Index of the highest-priority active external interrupt. Zero when no external interrupts with sufficient priority are both pending and enabled.
1	**-**	RES0
0	**update**	Writing 1 (self-clearing) causes hardware to update `meicontext` with 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. 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 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` correctly track the current interrupt frame across arbitrary levels of nested interrupts, so long as software saves/restores the `meicontext` register appropriately.

Bits	Name	Description
31:28	**pppreempt**	Previous `ppreempt`. Set to `ppreempt` when taking an interrupt, set to zero by `mret`. Has no hardware effect, but ensures that when `meicontext` is saved/restored correctly, `preempt` and `ppreempt` stack correctly through arbitrarily many preemption frames. TODO: how to track whether the `mret` is a return from external IRQ?
27:24	**ppreempt**	Previous `preempt`. Set to `preempt` when taking an interrupt, restored to to `pppreempt` by `mret`. IRQs of lower priority than `preemptp` are not visible in `meinext`, so that the preempted interrupt is not re-taken in the preempting frame. One bit smaller than `preempt`, because when `preempt` has its maximum value of 16, no further preemption is possible.
23:21	**-**	RES0
20:16	**preempt**	Minimum interrupt priority to preempt the current interrupt. Set to the priority of `meinext.irq`, plus one, when `meinext.update` is written. Interrupts with lower priority than `preempt` do not cause the core to transfer to an interrupt handler.
15:12	**-**	RES0
11	**noirq**	Not in interrupt (read/write). Set to 1 at reset or when taking any trap other than an external interrupt. Set to `meinext.noirq` when `meinext.update` is written.
10:9	**-**	RES0
8:0	**irq**	Current IRQ number (read/write). Set to `meinext.irq` when `meinext.update` is written..

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

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

```
    csrr a0, mepc
    sw a0, 64(sp)
    csrr a0, meicontext
    sw a0, 68(sp)
    csrr a0, mstatus
    sw a0, 72(sp)

    j get_next_irq

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

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

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

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

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

4.9. Custom Power Control CSRs

4.9.1. msleep

Address: `0xbf0`

M-mode sleep control register.

Bits	Name	Description
31:3	-	RES0
2	`deepsleep`	Deassert the clock request signal when entering the block or WFI state, and wait for clock acknowledge to reassert before leaving the block or WFI state.
1	`block`	Write 1 to enter a WFI sleep state until either an unblock signal is received, or an interrupt is asserted that would cause a WFI to exit. Clears automatically when leaving the blocked state. If an unblock signal has been received since the last time `block` was written to 1, the write is ignored. In other words, the blocked state falls through immediately in this case. An unblock signal is received when another hart writes 1 to its `unblock` register, or for some other platform-specified reason.
0	`unblock`	Write 1 to post an unblock signal to other harts in the system. Always reads back as 0.

4.9.2. sleep

Address: `0x8f0`

U-mode sleep control register. A subset of the fields in `msleep`. If `mstatus.tw` is 1, then attempting to access this register from U-mode causes an illegal opcode trap.

Bits	Name	Description
31:2	-	RES0
1	`block`	U-mode access to the `msleep.block` bit
0	`unblock`	U-mode access to the `msleep.unblock` bit

Chapter 5. Debug

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

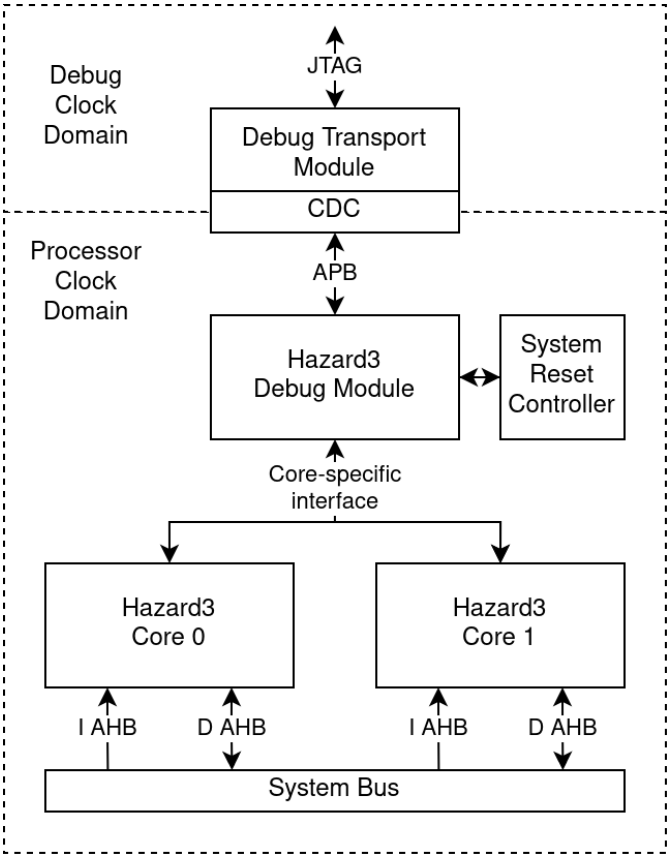
- Run/halt/reset control as required
- Abstract GPR access as required
- Program Buffer, 2 words plus `impebreak`
- Automatic trigger of abstract command (`abstractauto`) on `data0` or Program Buffer access for efficient memory block transfers from the host
- (TODO) Some minimum useful trigger unit —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

5.1. Debug Topologies

Hazard3's Debug Module has the following interfaces:

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

This is shown in the example topology below.



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

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

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

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

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

5.2. Implementation-defined behaviour

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

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

Not implemented:

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

The core behaves as follows:

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

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

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

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

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

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

5.3. Debug Module to Core Interface

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

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

TODO reset interface description