Hazard3

Table of Contents

1. Introduction 1
2. Instruction Cycle Counts
2.1. RV32I
2.2. M Extension
2.3. C Extension
2.4. Privileged Instructions (including Zicsr).
3. CSRs
3.1. Standard CSRs
3.1.1. mvendorid
3.1.2. marchid
3.1.3. mimpid
3.1.4. mstatus
3.1.5. misa
3.2. Custom CSRs
3.2.1. midcr
3.2.2. meie0
3.2.3. meip0
3.2.4. mlei
3.2.5. Maybe-adds
4. Debug
4.1. Implementation-defined behaviour.
4.2. UART DTM

Chapter 1. Introduction

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

- RV32I: 32-bit base instruction set
- M extension: integer multiply/divide/modulo
- C extension: compressed instructions
- Zicsr extension: CSR access
- M-mode privileged instructions ECALL, EBREAK, MRET
- 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:

- Support for WFI instruction
- A extension: atomic memory access
 - LR/SC fully supported
 - AMONone PMA on all of memory (AMOs are decoded but unconditionally trigger access fault without attempting memory access)

Chapter 2. Instruction Cycle Counts

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

2.1. RV32I

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

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

2.2. M Extension

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

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

2.3. C Extension

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

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

2.4. Privileged Instructions (including Zicsr)

Instruction	Cycles	Note		
CSR Access	CSR Access			
csrrw rd, csr, rs1	1			
csrrc rd, csr, rs1	1			
csrrs rd, csr, rs1	1			
csrrwi rd, csr, imm	1			
csrrci rd, csr, imm	1			
csrrsi rd, csr, imm	1			
Trap Request				
ecall	3	Time given is for jumping to mtvec		
ebreak	3	Time given is for jumping to mtvec		

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

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

Chapter 3. CSRs

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

3.1. Standard CSRs

3.1.1. mvendorid

Address: 0xf11

Read-only, constant. Value is configured when the processor is instantiated. Should contain either all-zeroes, or a valid JEDEC JEP106 vendor ID.

3.1.2. marchid

Address: 0xf12

Read-only, constant. Architecture identifier for Hazard3, value can be altered when the processor is instantiated. Default is currently all zeroes as unregistered.

3.1.3. mimpid

Address: 0xf12

Read-only, constant. Value is configured when the processor is instantiated. Should contain either all-zeroes, or some number specifiying a version of Hazard3 (e.g. git hash).

3.1.4. mstatus

blah blah

3.1.5. misa

Read-only, constant. Value depends on which ISA extensions Hazard5 is configured with.

3.2. Custom CSRs

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

3.2.1. midcr

Address: 0xbc0

Implementation-defined control register. Miscellaneous nonstandard controls.

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

3.2.2. meie0

Address: 0xbe0

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

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

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

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

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

3.2.3. meip0

Address: 0xfe0

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

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

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

• An interrupt is currently asserted in meip0

- The matching interrupt enable bit is set in meie0
- The standard M-mode interrupt enable mstatus.mie is set
- The standard M-mode global external interrupt enable mie.meie is set

In this case, the processor jumps to either:

- mtvec directly, if vectoring is disabled (mtvec[0] is 0)
- mtvec + 0x2c, if vectoring is enabled (mtvec[0] is 1) and modified external IRQ vectoring is disabled (midcr.eivect is 0)
- mtvect + (mlei + 16) * 4, if vectoring is enabled (mtvec[0] is 1) and modified external IRQ
 vectoring is enabled (midcr.eivect is 1). `
 - mlei is a read-only CSR containing the lowest-numbered pending-and-enabled external interrupt.

3.2.4. mlei

Address: 0xfe4

Lowest external interrupt. Contains the index of the lowest-numbered external interrupt which is both asserted in meip0 and enabled in meie0. Can be used for faster software vectoring when modified external interrupt vectoring (midcr.eivect = 1) is not in use.

Bits	Name	Description
31:5	-	RESO
4:0	-	Index of the lowest-numbered active external interrupt. A LSB-first priority encode of meip0 & meie0. Zero when no external interrupts are both pending and enabled.

3.2.5. Maybe-adds

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

Chapter 4. Debug

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

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

Hazard3's debug support implements the following:

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

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

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

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

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

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

4.1. Implementation-defined behaviour

Features implemented by DM (beyond the mandatory):

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

Not implemented:

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

Core behaviour:

- Branch, jal, jalr and auipc are illegal in debug mode, because they observe PC: attempting to execute will halt Program Buffer execution and report an exception in abstractcs.cmderr
- The dret instruction is not implemented (a special purpose DM-to-core signal is used to signal resume)
- The dscratch CSRs are not implemented
- External data0 register is exposed as a dummy CSR mapped at 0x7b2 (the location of dscratch0), readable and writable by the DM.
 - $\circ\,$ This is a debug mode CSR, so raises an illegal instruction exception when accessed in machine mode
 - The DM ignores writes unless it is currently executing an abstract command on this core (hartsel = this core, abstractcs.busy = 1)
- dcsr.stepie is hardwired to 0 (no interrupts during single stepping)
- dcsr.stopcount and dcsr.stoptime are hardwired to 1 (no counter or internal timer increment in debug mode)
- dcsr.mprven is hardwired to 0
- dcsr.prv is hardwired to 3 (M-mode)

4.2. UART DTM

Hazard3 defines a minimal UART Debug Transport Module, which allows the Debug Module to be accessed via a standard 8n1 asynchronous serial port. The UART DTM is always accessed by the host using a two-wire serial interface (TXD RXD) running at 1 Mbaud. The interface between the

DTM and DM is an AMBA 3 APB port with a 32-bit data bus and 8-bit address bus.

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

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

The host may send the following commands:

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

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

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

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

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