# Hornet RISC-V Core
# User Guide

Yavuz Selim Tozlu
0yavuz0@gmail.com

July 1, 2021

# Contents

# Chapter 1

# Supported Instructions

Hornet supports the following RISC-V instructions,

- RV32I Base Integer Instruction Set, Version 2.1 - Excluding FENCE instruction
- "M" Standard Extension for Integer Multiplication and Division, Version 2.0
- "Zicsr", Control and Status Register (CSR) Instructions, Version 2.0
- MRET trap return instruction

Attempting to execute instructions that are not supported will cause an illegal instruction exception.

# Chapter 2

# Machine-Level ISA

## 2.1 CSRs

Hornet supports the following CSRs, as defined in the Machine-Level ISA, Version 1.11

- `mstatus` — mie, mpie and mpp bits only, rest are hardwired to zero.
- `mtvec`
- `mip` and `mie` — msi*, mti*, mei* and platform-specific interrupt bits only, rest are hardwired to zero.
- `mscratch`
- `mepc`
- `mcause`

Writes to non-existent CSRs are ignored.

## 2.2 Interrupts and Exceptions

Hornet supports all machine-level interrupts, i.e. software, timer and external interrupts. It also supports platform-specific interrupts.

Hornet supports the following exception sources,

- Instruction access fault
- Instruction address misaligned
- Illegal instruction
- ECALL and EBREAK
- Store/Load access fault

An instruction access fault occurs whenever the `instr_access_fault_i` input is asserted.

An instruction address misaligned exception occurs when the branch/jump address calculation at EX stage produces a misaligned address.

An illegal instruction exception occurs when an unknown instruction is encountered. This exception will be raised when the illegal instruction reaches the ID stage.

An ECALL or an EBREAK exception will raised when the associated instructions are encountered. These exceptions are raised when the instructions reach the ID stage.

A store/load access fault occurs whenever the `data_err_i` input is asserted.

When an interrupt occurs, the PC of the oldest instruction in the pipeline is saved, and the pipeline is flushed. The next instruction is fetched from the interrupt handler routine. An MRET instruction will return the core from the handler, and the execution will continue from the interrupted instruction.

When an exception occurs, the PC of the faulting instruction is saved, and the pipeline is flushed. The next instruction is fetched from the exception handler routine. An MRET instruction will return the core from the handler. However, it is the handler's responsibility to update the contents of `mepc` register, as the core will return to the address that was stored there.

# Chapter 3

# Memory Interface

Hornet employs a simple data memory interface. The signals and their functions are listed below,

- `data_i`: 32-bit data input.
- `data_wmask_o`: 4-bit data write-mask output.
- `data_wen_o`: 1-bit write-enable output, active-low.
- `data_addr_o`: 32-bit data address output.
- `data_o`: 32-bit data output.
- `data_req_o`: 1-bit data request output. Driven high if a memory transaction is requested.
- `data_err_i`: 1-bit data access error input. If this signal is driven high, a load/store access fault occurs.

Address of the data is calculated and outputted at the EX stage, along with data, mask, write-enable and request signals. If the instruction is a load, then at the MEM stage, the data is read from the `data_i` input.

Instruction memory interface is identical to data memory interface, except it has even fewer signals.

Hornet can handle misaligned memory accesses. It will split the misaligned access into two aligned accesses. Hence, a misaligned access will take one additional clock cycle to complete.

# Chapter 4

# Integrating Hornet into Your Design

In order to integrate Hornet core into your design, you need to instantiate it in your source file as follows,

```
core    //Program counter will be set to reset_vector when a reset occurs.
        //By default, it is 0.
        #(.reset_vector())
        core0(
        //Clock and reset signals. Resets are both active-low
        .clk_i(),
        .hreset_i(), //Hard reset, sets mcause to 0
        .sreset_i(), //Soft reset, sets mcause to 1. Tie this to 1 if
                     //you don't need it

        //Data memory interface
        .data_addr_o(),
        .data_i(),
        .data_o(),
        .data_wmask_o(),
        .data_wen_o(), //active-low
        .data_req_o(),
        .data_err_i(),

        //Instruction memory interface
        .instr_addr_o(),
        .instr_i(),
        .instr_access_fault_i(),

        //Interrupts
        .meip_i(),
        .mtip_i(),
        .msip_i(),
        .fast_irq_i(),
        .irq_ack_o());
```

`hreset_i` and `sreset_i` signals reset all registers in the core. In addition to that, they also set the `mcause` register to the specified values. This allows the user to distinguish the source of the reset.

`meip_i` signal is the machine-level external interrupt input. This input should be kept high until the core responds by asserting the `irq_ack_o` signal for one clock cycle. The IRQ acknowledge signal is only asserted from this interrupt, as a way to notify the interrupt controller.

`msip_i` and `mtip_i` signals are machine-level software and timer interrupt inputs, respectively. From the core's perspective, the only difference is the value that `mcause` is set to.

`fast_irq_i` signal is a 16-bit long input. It corresponds to the platform-specific interrupts. The core utilizes a priority scheme among these fast interrupts. The interrupt with the smallest index, i.e. `fast_irq_i[0]`, has the highest priority; and the interrupt with the greatest index, i.e. `fast_irq_i[15]`, has the lowest priority. Driving the inputs high for one clock cycle is sufficient to cause an interrupt, as the core will register them internally.

Platform-specific interrupts have higher priority than machine-level interrupts, hence the name "fast interrupts".

If you don't plan to use interrupts, you can tie the inputs to zero.

The only mandatory peripheral for the core to work is memory. An example memory module is provided in the peripherals directory.