RiscV Documentation

RISCV-BOOM's documentation! : https://docs.boom-core.org/en/latest/index.html

Rocket chip generator necessary to instantiate the RISC-V Rocket Core:
https://github.com/chipsalliance/rocket-chip

What is RISC-V? & Basics of RISC-V

Introduction to the RISC V and first program in RISC V:

https://www.youtube.com/watch?v=KBvAKHsHBW4

Part I: An Introduction to the RISC-V Architecture

https://www.youtube.com/watch?v=m8DqCTogb8w

ROCC vs MMIO --- ????

RISC-V, Rocket, RoCC, Chisel, Scala:

https://inst.eecs.berkeley.edu/~cs250/sp17/disc/lab2-disc.pdf

First Accelerator

https://inst.eecs.berkeley.edu/~cs250/fa13/handouts/lab3-sumaccel.pdf

Thesis

Designandprogrammingofa coprocessorforaRISC-V architecture. (n.d.). Retrieved December 15, 2020, from https://webthesis.biblio.polito.it/6589/1/tesi.pdf

What is RISC-V? & Basics of RISC-V

Introduction to the RISC V and first program in RISC V:

https://www.youtube.com/watch?v=KBvAKHsHBW4

Instruction Set Architecture (ISA)

- defines the Software interface for hardware.
- Has many hardware implementations
- Specification may be for general purpose MicroP, DSPs (digital signal processor), or specialized hardware operations
- ISA creates its own software ecosystem (x86, ARM, etc)

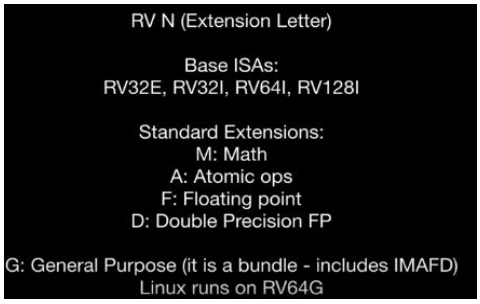
RISC V is a well organized ISA divided into categories and extensions.

Whats included in ISA

- **Defines:** Set of instructions, Data Types, Registers, Addressing nodes, I/O,..... etecetc

Why RISC V

- Simplicity
- Open Source
- New business model (since its open source)
- On par with modern CPUs (performance, code, density, power)



RV32E - RISC V 32bits (E means sub-set of 32I, were it has 16registers instead of 32)

RV32I - RISC V 32bits Integer

Devo mencionar tambem teste exemplo de código em RISC V??

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

```c
#include <stdio.h>

extern int add(int x, int y);

int main() {
    int result = 0;

    printf("Hello, World!\n");

    result = add(0xF0, 0x0F); // 0xFF

    printf("Result of add: 0x%x\n", result);
}
```

```
.section .text

.global add
.type add, @function

add:
    add     a0, a0, a1
    ret
```

Part I: An Introduction to the RISC-V Architecture

https://www.youtube.com/watch?v=m8DqCTogb8w

What is RISC-V?

- **A high-quality, license-free, royalty-free RISC ISA**
- **Standard maintained by the non-profit RISC-V Foundation**
- **Suitable for all types of computing systems**
  - From Microcontrollers to Supercomputers
- **RISC-V is available freely under a permissive license**
- **RISC-V is not…**
  - A Company
  - A CPU implementation

- **ISA Name format: RV[###][abc…..xyz]**
  - RV – Indicates a RISC-V architecture
  - [###] - {32, 64, 128} indicate the width of the integer register file and the size of the user address space
  - [abc…xyz] – Used to indicate the set of extensions supported by an implementation.

Extension	Description
I	Integer
M	Integer Multiplication and Division
A	Atomics
F	Single-Precision Floating Point
D	Double-Precision Floating Point
G	General Purpose = IMAFD
C	16-bit Compressed Instructions
Non-Standard User-Level Extensions	
Xext	Non-standard extension "ext"

Common RISC-V Standard Extensions
***Not a complete list**

Register File

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function Arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

RISC-V Modes:

Machine mode is highest privilieged mode. Flexibility allows for range of targeted implementations from simple MCUs to high-performance Application Processors

Each have Controls and Status Registers

RISC-V Modes		
Level	Name	Abbr.
0	User/Application	U
1	Supervisor	S
2	Hypervisor	HS
3	Machine	M

a

Notes:
-nothing can affect the execution of atomic command

-Same instruction set regardless of the machine (x32, x64, etc)

Compression (C extension)

Most base integer instructions compress to 16-bit equivalents. **Smaller code size** means **increased performance**, **reduced power**, resulting in **reduce cost in embedded system** (smaller Flash/ROM/RAM).

Atomics (A extension)

Atomic memory operation (AMO) perform Read-Modify-Write in single Atomic instruction (Acquire aq and Release rl for release consistency)

Fences instructions: are used to enforce program order on device I/O and memory accesses (barrier) Não entendi bem mas é importante para agora?

Control and Status Registers (CSRs)

Have their own instructions (R/W, Read and set/clear bit). Used to transfer control of the execution environment and a higher privileged mode

CSRs are registers that contain the working state of the RISC-V machine

CSRS are specific and different to a Mode, and defined in detail in RISC-V specification.

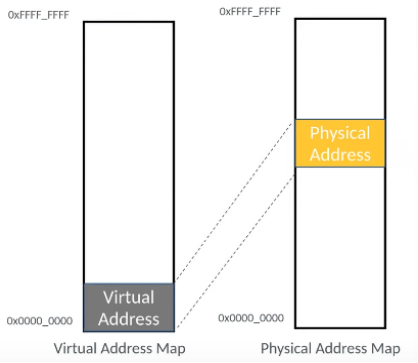
Machine Status (*mstatus*) – Most important CSR, most related to interrupts.

Timer CSRs - *mtime* and *mtimecmp* (used to generate timer interrupt). Must run in constant frequency

Supervisor CSRs – can be used to control state of Supervisor and User Modes. Machine mode CSRs have Supervisro mode equivalents, having the same mapping without the machine control bits.

Virtual Memory

Has supost for Virtual Memory allowing for sophisticated memory management and OS support (Linux). Requires an S-Mode implementation. (Page tables contain access permissions attributes, if the page in memory is accessible by less privileged modes)

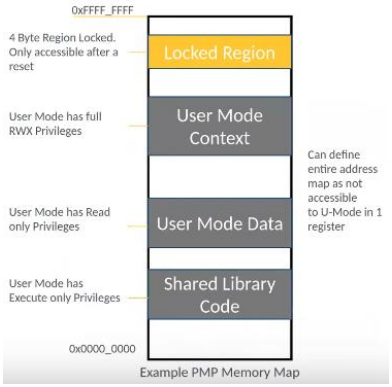


Physical Memory Protection (PMP)

Used to enforce access restrictions on less privileged modes (Prevent Supervisor and User Mode software from accessing unwanted memory).

Up to 16 regions, with minimum region size of 4 bytes.

And ability to Lock a region. (enforces permissions on all accesses, only way to unlock is through a Reset)



Interrupts

-Software -Timer -External -Local (hart specific Petipheral interrupts)

Optionally per privilege level (interrupts target for Supervisor, and other for User)

Interrupts are identified by reading the *mcause* CSR. The interrupt field determines if a trap was caused by an interrupt or an exception.

Bits	Field Name	Description
XLEN-1	Interrupt	Identifies if an interrupt was synchronous or asynchronous
[XLEN-2:0]	Exception Code	Identifies the exception

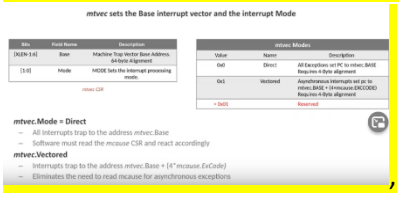
mcause CSR

Interrupt = 1 (interrupt)	
Exception Code	Description
0	User Software Interrupt
1	Supervisor Software Interrupt
2	Reserved
3	Machine Software Interrupt
...	

Interrupt = 0 (exception)	
Exception Code	Description
0	Instruction Address Misaligned
1	Instruction Access Fault
2	Illegal Instruction
3	Breakpoint
...	

mie (enable diseble a interrupt) *mip* (indicates which interrupts are currently pending)

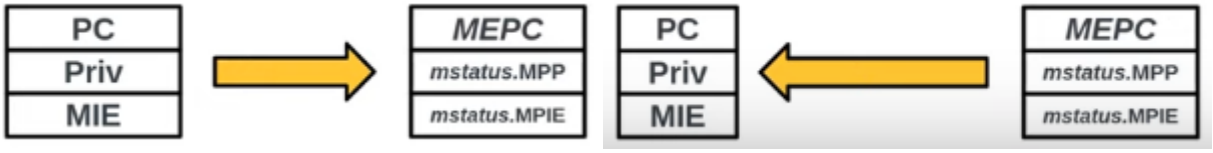
-Machine trap Vector. Pareceu-me bué complicado passei á frente. Ok ou convem entender bem?



When exception happens:

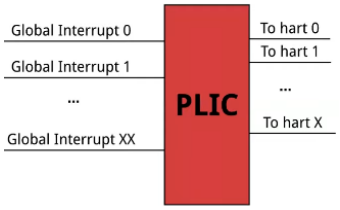
mtevc.MODE = Direct

On entry, the RISC-V hart saves the current state (MEPC, *mstatus.MPP, mstatus.MPIE)* and then set PC = *mtvec*, and *mstatus.MIE = 0.* In exit, MRET instruction restores state



Since Pushing and Popping Registers in Assembly is a pain, the *interrupt* attribute was added to GCC to facilitate interrupt handlers written entirely in C (no assembly code). The Interrupt handler is with *interrupt* attribute, and the fuction only saves/restores necessary registers onto the stack.

Global interrupts, are defined as Interrupts which can be routed to any hart in a system. Global I. are prioritized and distributed by the Platform Level Interrupt Controller (PLIC). The PLIL is connected to the External Interrupt signal for 1 or more harts in an implementation.



On interrupt:

1. the PLIC signals the Hart using Machine External Interrupts.
2. The interrupt handler branches to the defined function to handle the Machine External Interrupt
3. The Machine External Interrupt handler does:
    a. Reads the PLIC's register to determine the highest priority pending interrupt
    b. Uses another vector table to branch to the interrupt's specific handler
    c. Completes the interrupt by writing the interrupt number back to the PLIC's

```
void machine_external_interrupt()
{
  //get the highest priority pending PLIC interrupt
  uint32_t int_num = plic.claim_comlete;
  //branch to handler
  plic_handler[int_num]();
  //complete interrupt by writing interrupt number
  back to PLIC
  plic.claim_complete = int_num;
}
```

Resources

//riscv.org/

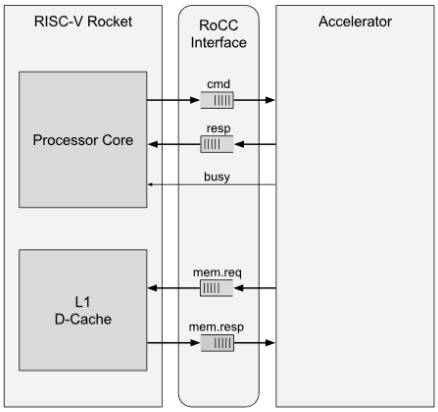
Github.com/sifive/

Github.com/riscv/

Sifive.com/

RoCC vs MMIO

What's RoCC?



Simplified View of RoCC

Rocket

- Rocket is one implementation of the RISC–V ISA
- Rocket is a 64 bit implementation that has an integrated L1 and L2 data cache
- A special interface, known as the RoCC interface, was defined to help attach accelerators to Rocket

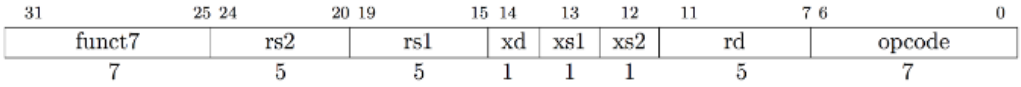
Cmd – carries the instructions

Resp – carries the valueto be written into the destination reg

Mem.req – carries memory requests

Mem.resp – carries a response to a mem request

Rocket core processes custom Rocket Custom Coprocessor (RoCC) instructions and passes requests to the accelerator. RoCC instructions format:



"*The accelerator will interact with the Rocket Processor and the shared memory system via the standard RocCC interface. Each portion of the interface is decoupled by standard queues and ready-valid signals.*

The Rocket core initiates an accelerator command by passing most of the RoCC instructions directly to the accelerator, as well as the relevant register values."

What's MMIO?

"Memory-mapped I/O (MMIO) is a method of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer. An alternative approach is using dedicated I/O processors, commonly known as channels on mainframe computers, which execute their own instructions. (…)

Memory-mapped I/O uses the same address space to address both memory and I/O devices. The memory and registers of the I/O devices are mapped to (associated with) address values. (…)

To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory. The reservation may be permanent, or temporary. (…)

Software writes data to an address and then writes data to another address, the cache write buffer does not guarantee that the data will reach the peripherals in that order."

RoCC vs MMIO

https://chipyard.readthedocs.io/en/latest/Customization/RoCC-or-MMIO.html

Accelerators or custom I/O devices can be added to your SoC (System on Chip) in several ways:

- MMIO Peripheral (Memory Maped I/O) (TileLink-Attached Accelerator)
    - With the TileLink-Attached approach, the processor communicates with MMIO peripherals through memory-mapped registers.

- Tightly-Coupled RoCC Accelerator
    - The processor communicates with a RoCC accelerators through a custom protocol and custom non-standard ISA instructions reserved in the RISC-V ISA encoding space.
    - Each core can have up to four accelerators that are controlled by custom instructions and share resources with the CPU.

These approaches differ in the method of the communication between the processor and the custom block.

The communication through a RoCC interface requires a custom software toolchain, whereas MMIO peripherals can use that standard toolchain with appropriate driver support.

Escrever diferenças

THESIS

Introduction

Coprocessor

Accelerator, used to improve the performance of a computing system. In the past some of this devices, like arithmetic cores or even FPUs, used to be external chips used to expand the capabilities of a system. Nowadays many of these functions are integrated in the instruction sets and are often embedded in the pipeline of the processors.

Thesis objective

studying and understanding the Rocket core and the RoCC interface and exploring the possibility of enhancing the core with the use of accelerators. Presents a new approach for the integration of coprocessors with this interface.

An analysis of the performance of the RoCC interface is presented, taking into account the overhead and the latency introduced by the communication between the core and the coprocessor.

• **Chapter 1** contains the basic background information regarding the RISCV ISA, the Rocket chip generator and the Rocket core which are needed for understanding the platform on top of which this work is developed.

• **Chapter 2** explains the RoCC interface and its main signals, providing an analysis of the latency and some possible improvement.

RISC-V and Rocket-core overview

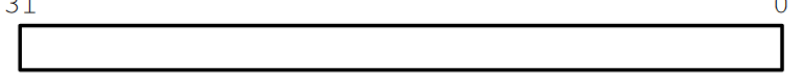
1.1 RISC-V an open ISA

RISC-V is now increasingly drawing the attention of both industry and academia, because it offers the possibility to ease the design of processors from the high costs of compiler support, without requiring to resort to expensive commercial ISAs.

The base integer instruction set, in both in its 32 and 64 bits variant, was devised in order to include a small number of instructions and reduce the complexity of the hardware needed for a minimal implementation. Even if the base ISA is small, it was also designed to be a reasonable target for compilers and software development.

1.1.1 Instruction formats

- By convention, RISCV instructions are each 1 word = 4 bytes = 32 bits



RISC-V base integer ISA defines four basic instruction formats called *R-type*, *I-type*, *S-type* and *U-type.*

- R-Format: instructions using 3 register inputs –*add, xor, mul* —arithmetic/logical ops
- I-Format: instructions with immediates, loads–*addi, lw, jalr, slli*
- S-Format: store instructions: *sw, sb*
- SB-Format: branch instructions*: beq,bge*
- U-Format: instructions with upper immediates–*lui, auipc* —upper immediate is 20-bits
- UJ-Format: jump instructions: *jal*

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

These instruction format were defined so that the registers fields for both the sources (*rs1* and *rs2*) and the destination (*rd*) are kept int the same position for all the formats, in order to simplify the decoding hardware. For the same reason the immediates were placed towards the leftmost significant bits and the sign bit position is always in the bit 31 of the instruction.

R-Format Instruction

- Define “fields” of the following number of bits each: 7 + 5 + 5 + 3 + 5 + 7 = 32.
- Each field is viewed as its own unsigned int
- 5-bit fields can represent any number 0-31, while 7-bit fields can represent any number 0-128

1. *funct7+funct3 (10):* combined with opcode, these two fields describe what operation to perform.
   1.1. With opcode fixed at 0b0110011, number of instructions able to we encoded: $2^7 * 2^3 = 2^{10} = 1024$
2. rs1 (5): 1st operand (“source register 1”)
3. rs2 (5): 2nd operand (second source register)
4. rd (5): “destination register” — receives the result of computation
   4.1. RISCV has 32 registers. A 5 bit field can represent exactly $2^5 = 32$ things

1.2 Rocket Chip SoC generator and the Rocket Core

Rocket Chip generator is used to configure and generate the Rocket Core. They are part of the same framework that allows the designers of SoCs to rapidly come up with design architectures.

The SoC generator allow the composition of modular designs by using the parametrization features of the *Chisel* language.

The basic design flow allows the integration of new hardware component described in Chisel, the creation of new custom configurations, the compilation of the Chisel/Scala sources to generate C++ models for cycle accurate simulations and the generation of the synthesizable Verilog RTL.

Rocket core

Rocket core is both a processor generator and a library of processor components.
As a generator it is able to produce a family of processor core designs, with different configuration parameters.

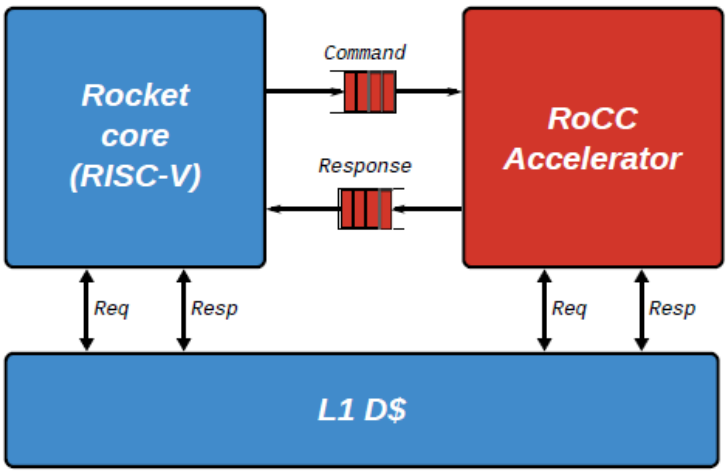
The RoCC coprocessor interface

Details of the RoCC coprocessor interface and presents an analysis of the latency using the model of *read/write* operations for the data exchange between the core and the accelerator. *load/store* instruction model is introduced for the data transfers between the coprocessor and the memory.

2.1 RoCC interface overview

RoCC is an interface designed in order to extend the Rocket Core and allow easy decoupled communications between the core and the attached coprocessors.
The RoCC interface is divided in sub-interfaces each creating directional links to connect the accelerators with other parts of the SoC.

The ***cmd*** sub-interface connects the core with the accelerator and is used to send commands. With these commands the core can also request data to the coprocessor, this data can be sent by the coprocessor to the core through the ***resp*** (response) interface.



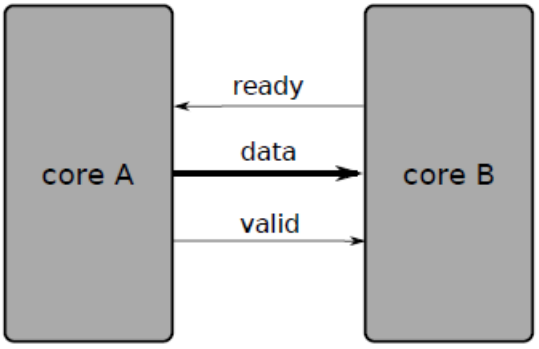
In order to allow the coprocessor to access the memory the RoCC interface also provides the ***mem_req*** (request) and ***mem_resp*** (response) direct links to the data cache.

RoCC interface provides some more sub-interfaces (*extended RoCC interface*), p.e., it is possible to connect an accelerator with the FPU (Floating Point Unit).

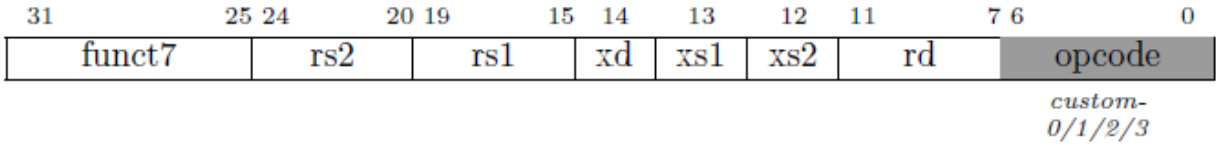
The interface also provides some more status signals and an interrupt line that can be used for synchronizing with the core or for signaling errors.

2.1.1 RoCC command and response interfaces

The command interface is used to send the instructions and the corresponding data to the coprocessor, while the response interface is used by the coprocessor to send the results to the integer register file.



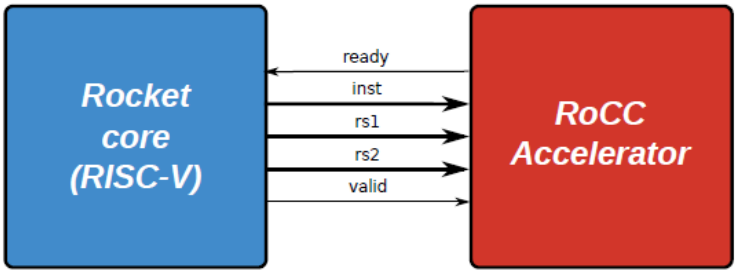
Both the command and the response ports are based on the *Decoupled* interface available in Chisel. This type of connection is based on a FIFO like *ready/valid* protocol in which the sender drives the *valid* signal and the data and waits for the receiver to raise the *ready* signal, the transfer is considered accepted if both valid and ready are *high* on the same clock cycle.



The data bus of the command interface is composed of the following signals:

- **rs1** a 32-bit (or 64 depending on the *XLen* parameter) data bus for the content of the integer register addressed by the *rs1* field of the instruction
- **rs2** a 32-bit data bus holding the value of the integer register addressed by the *rs2* field of the instruction
- **xd** bit is set when *inst_rd* is a valid destination register: the core wants to receive data in the destination register pointed by *inst_rd*.
- **xs1** bit is set when *inst_rs1* is a valid source register: the core is sending the content of the first source register (*inst_rs1*) in the *rs1* data bus.
- **xs2** bit is set when *inst_rs2* is a valid source register: the core is sending the content of the second source register (*inst_rs2*) in the *rs2* data bus.

The values assumed by these bits heavily influences the behaviour of the pipeline



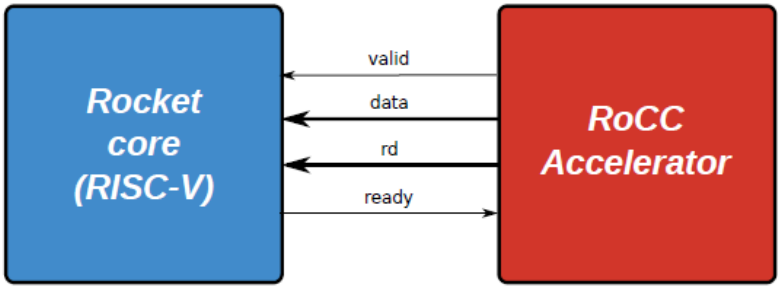
The *cmd* interface signals

When a custom instruction with the *inst_xd* bit set arrives, the core will expect to receive, at some point, a result to be stored in the register pointed by *inst_rd*. This means that if a successive instruction uses that register before the value is produced and stored, the core will stall the pipeline to wait the data from the coprocessor. This means that if a successive instruction uses that register before the value is produced and stored, the core will stall the pipeline to wait the data from the coprocessor.

The *resp* interface signals

Then, the accelerator have to use the *resp* sub-interface, which is also a *Decoupled* (ready/valid) interface. In this case the coprocessor drives the valid signal and the "data" bus, while the core only drives the ready signal. The data information includes the following buses:

- **rd** the five bits field specifying the destination register of the response, must be the same received with the command.
- **data** a 32 or 64-bit bus with the data content to be written in the *rd* register.



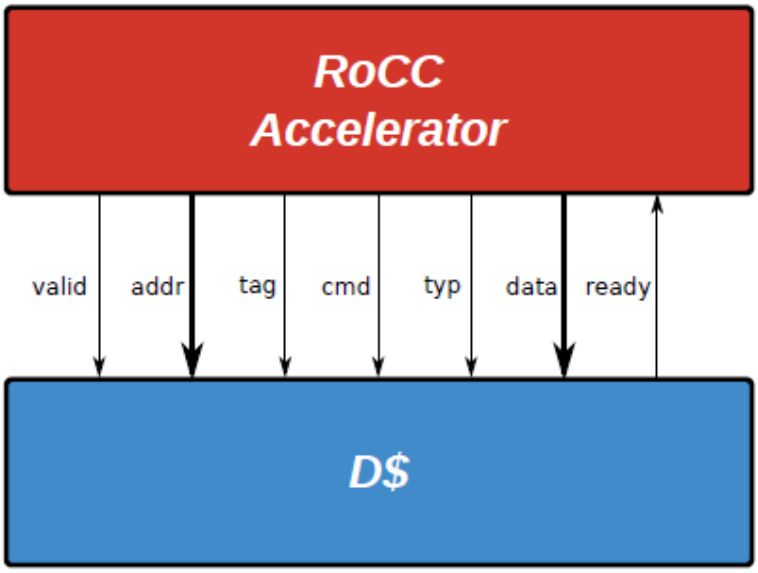
2.1.2 Memory request and response interfaces

In order to allow the accelerator to have direct access to the first level data cache, the RoCC interface specifies two channels for the memory requests ***mem_req*** and for the response's ***mem_resp*.**

mem_req sub-interface

When an accelerator wants to issue a load or store operations to the memory it can use the *mem_req* sub-interface (*Decoupled* interface. The most important data signals of this sub-interface are:

- **addr** a 32 (for RV32) or 40-bit (for RV64) bus carrying the address for the memory access.
- **tag** 8-bit bus used to uniquely identify each request.
- **cmd** 5-bit bus carrying the memory operation code (0000_2 = load, 0001_2 = store)
- **typ** 3-bit bus that specifies the width of the of the transfer operation (000_2 = 8-bit, 001_2 = 16-bit, 010_2 = 32-bit and 011 = 64 bit).
- **phys** 1 bit signal asserted if a physical address is used or zeroed if the address is virtual and needs a translation.
- **data** 32 or 64-bit bus used for sending the data in case of store operations.

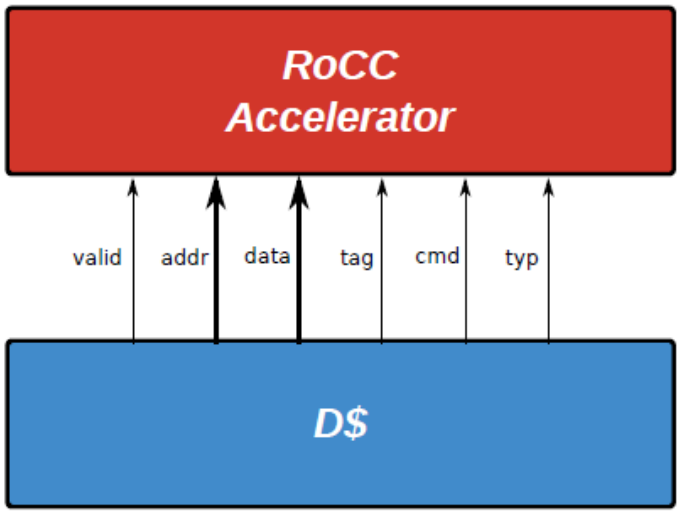


mem_resp sub-interface

When the response from the cache is ready it is sent back to the accelerator through the *mem_resp* sub-interface. It is not based on the *Decoupled* interface. This is because the coprocessor cannot keep the cache waiting, so it must accept the response without the possibility of postponing the transaction.

In fact, the *mem_resp* sub-interface does not present a *ready* signal but is instead based on a simpler *"valid"* interface. This means that the coprocessor must accept every memory response as soon as the valid line is high. With this interface the sender (the cache) drives the *data* and the *valid* signals and the receiver (the accelerator) just need to check that the valid line is high and accept the incoming data.
The most important data signals of this sub-interface are:

- **addr** a 32 (for RV32) or 40-bit (for RV64) bus carrying the load/store address.
- **tag** 8-bit bus used to distinguish between responses to multiple requests.
- **cmd** 5-bit bus carrying the memory operation code (0000_2 = load, 0001_2 = store)
- **typ** 3-bit bus that specifies the width of the of the response data (000_2 = 8-bit, 001_2 = 16-bit, 010_2 = 32-bit and 011 = 64 bit).
- **data** 32 or 64-bit bus carries the data response for a load operation.



2.2 Custom instructions

This interface make use of the four custom opcodes. To simplify the coprocessor only four operations are defined: read/write and load/store.

2.2.1 The addressing mode

xd, *xs1* and *xs2* are used to validate the source or destination registers on the core side. As a convention, that when one of these bits is zero the corresponding register field refers to an internal register of the accelerator.
Using the data arriving from the core for the internal registers addressing grants two main advantages:

- the possibility of using register banks wider than 32 registers, since in principles it is possible to address up to 2^{32} registers.
- is possible at the code level to address the internal registers of the accelerator by using normal integer variables.

xd	xs1	xs2	inst_rd	inst_rs1	inst_rs2
0	0	0	RoCC	RoCC	RoCC
0	0	1	not allowed		
0	1	0	RoCC	Core	RoCC
0	1	1	RoCC	Core	Core
1	0	0	Core	RoCC	RoCC
1	0	1	not allowed		
1	1	0	Core	Core	RoCC
1	1	1	Core	Core	Core

2.3 Read/write operations between the core and RoCC

Pseudo-instructions are defined:

- *rocc_read rd, cps*: the core reads the data arriving from the coprocessor register *cpx* and write it into the destination register *rd*.
- *rocc_write cpd, rs*: the data in the core register *rs* is written in the accelerator register *cpd*.

funct7	xd	xs1	xs2	inst_rd	inst_rs1	inst_rs2	rs1	rs2	operation
read	1	0	0	Core	Acc	-	data1	-	C[inst_rd] ← A[inst_rs1]
read	1	1	0	Core	Core	-	data1	-	C[inst_rd] ← A[data1]
write	0	1	0	Acc	Core	-	data1	-	A[inst_rd] ← data1
write	0	1	1	Acc	Core	Core	data1	data2	A[data2] ← data1
load	0	1	0	Acc	Core	-	data1	-	A[inst_rd] ← M[data1]
load	0	1	1	-	Core	Core	data1	data2	A[data2] ← M[data1]
store	0	1	0	-	Core	Acc	data1	-	M[data1] ← A[inst_rs2]
store	0	1	1	-	Core	Core	data1	data2	M[data1] ← A[data2]

Instructions defined for the RoCC coprocessors. "C[x]" indicates an access to the x-th integer register, "A[i]" indicates an access to the i-th register of the accelerator, "M[x]" is used for a memory access at the address x.

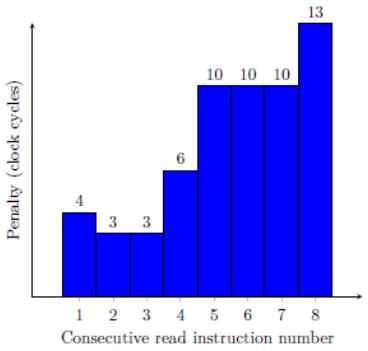
Latency study

The analysis is focused mainly on the performance of read operations, this is because the coprocessor and the core are decoupled.

read and use operations

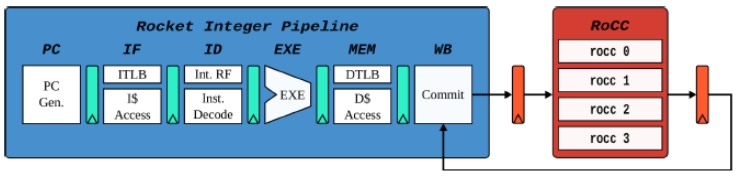
```
rocc_read   rd, cpx   ; rd ← RoCC[cpx]   5 cycle
addi        rx, rd, 1 ; rx ← rd + 1       1 cycle

rocc_read   rd, cpx   ; rd ← RoCC[cpx]   1 cycle
nop                   ;                  4 cycle
addi        rx, rd, 1 ; rx ← rd + 1       1 cycle

rocc_read   rd, cpx   ; rd ← RoCC[cpx]   1 cycle
nop                   ;                  1 cycle
...
nop                   ;                  4 cycle
addi        rx, rd, 1 ; rx ← rd + 1       1 cycle
```

burst of read operations



Improving latency

The main reasons behind the relatively high latency overhead introduced by the interface are to be researched in the position of the interface in the pipeline. The Rocket core forwards the custom instructions through all the stages of the pipeline.
Solution: With an interface placed in the Execute (EXE) stage, a couple of clock cycles could have been gained, but the coprocessors would have been more complex in order to deal with the exceptions.



The other major cause that limits the throughput of *rocc_read* operations, can be found in the way the core handles the write-back of data coming from the interface.
This means that the data arriving from the *RoCC* interface is written in the RF only when the WB stage is occupied by a "non writing" instruction.
In the WB stage a check on the *wxd* bit is performed and the *ready* signal of the RoCC *resp* interface depends on that bit being at zero. For this reason as long as there is a valid instruction with the *wxd* bit at one, in the WB stage the writing form the *resp* interface is preempted.

Solution: A possible way to avoid paying a stall for every *rocc_read* consist in masking the latency of the interface with some useful instructions with an additional "useless" instruction used as a kind of free write-back slot. In order to apply this strategy the useless instruction must be one with the *wxd* bit at zero and must not alter the internal state of the core. Basically a *"nop"* instruction that does not occupy the WB slot is needed.

- **Modification of the NOP operation**
  wb_wxd = *wb_reg_valid* && *wb_ctrl.wxd* && (*wb_waddr* != 0)
- **Using useless branch to mask the latency~**
  bnez zero, addr ; RoCC write-back slot 1 cycle

2.4 Loads and stores between RoCC and cache memory

- *rocc_load cpd, rs*: the coprocessor performs a load operation using the content of the integer register *rs* as memory address and puts the result in its internal register cpd.
- *rocc_store cps, rs*: the coprocessor performs a store operation of the value contained in *cps* using the content of the register *rs* as memory address.

The use of single transfer instructions is also limited because, before launching this kind of instructions, the integer register rs should be initialized with the proper address.
This basically results in a overhead of at least one additional instruction for each *rocc_load*, for preparing *rs*, this usually means moving the address in the register.

In general the use of burst instructions is heavily application dependent and not general enough for implementing them in every coprocessor.

Latency study

As it can be seen, the response arrives in the second clock cycle after the request is fired (*ready* and *valid* both high in the same clock cycle).
the behaviour of the *req* and *resp* channel is basically the same for single or multiple access.