Documentation

mini-riscv processor

Professor:
Yvon Savaria
yvon.savaria@polymtl.ca

Inspired from the work of:
Mickaël Fiorentino
mickael.fiorentino@polymtl.ca

Lab instructor:
Justin Pabot
justin.pabot@polymtl.ca

Lab instructor:
Timothée TREMBLY
timothee-mateo.trembly@polymtl.ca

Automne 2023

CONTENTS

1 INTRODUCTION

The performance improvements of microprocessors—since the Intel 4004 in 1971—takes place on two main avenues:

1. Transistor technologies: The reduction of transistor sizes, the increase in integration density, and the voltages drops, have enabled a constant increase of the operating frequency of microprocessors at a near constant power density.

2. Microarchitectural design techniques: Pipelining, caches architectures, branch predictors, buses, *etc.* allow to extract the most performances from the transistor technologies.

For example, FIGURE 1 shows the layout of a RISCV PULP microprocessor, developed by ETH Zurich in 2015. It was manufactured by STMicroelectronics with a 28 nm CMOS FD-SOI technology. It contains 2.5 millions of logic gates (2500 *kGE—Gate Equivalent*), occupying a total area of 2.7 mm^2, and consuming 1.2 mW at 50 MHz with a near-threshold power supply of 0.6 V.

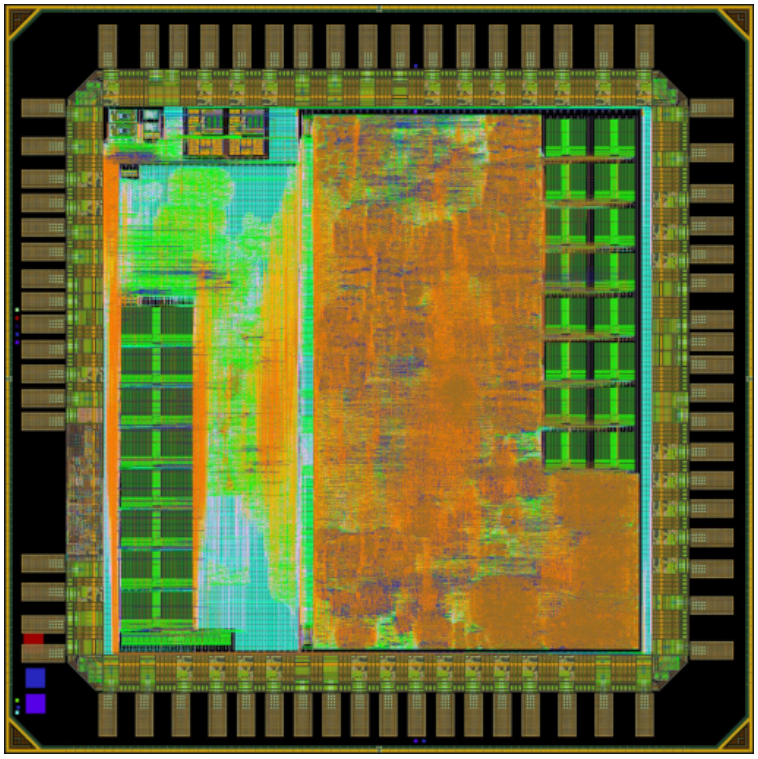

FIGURE 1: Layout of the RISCV *PULPv3* microprocessor

This laboratory consists in designing, and implementing with a 45 nm CMOS technology, a simple RISCV microprocessor called *mini-riscv*. Its instruction set architecture (ISA) is derived from the RV32I architecture, and its microarchitecture uses a 5-stages pipeline. The objective of this laboratory is to make you go through the main steps of the design of a microprocessor, from its hardware description in VHDL, up to its layout.

2 ARCHITECTURE

This part presents the architecture of the *mini-riscv* processor. That is, the specifications defining the hardware/software interface. We will first deal with the instruction set architecture, then with the instruction and data memories interfaces, and we will finish by basic concepts of assembly language, and few compilation directives.

2.1 INSTRUCTION SET ARCHITECTURE

The instruction set architecture (ISA) is the specification that a processor should implement in order to be compliant with the associated software stack. This includes the list of instructions, instruction formats, addressing modes, *etc.*. The *mini-riscv* ISA is derived from the RV32I specification. It is based on the *RISC* (*Reduced Instruction Set Computer*) design principles, which can be summarized as follow:

- Instructions are encoded in a fixed format.
- Arithmetical and logical operations are only performed on registers: *mini-riscv* contains a register file of 32 × 32 bits (`x0...x31`), with `x0=0x0`.
- The data memory is only accessible from the *load* and *store* instructions (`lw` and `sw` in the *mini-riscv*). To perform an operation on memory elements, one must first load the memory element in a register, then perform the operation, and finally store the result back in memory.

The *mini-riscv* ISA uses a fixed instruction format of 32 bits. There are 6 different instruction formats, as shown on FIGURE 2.

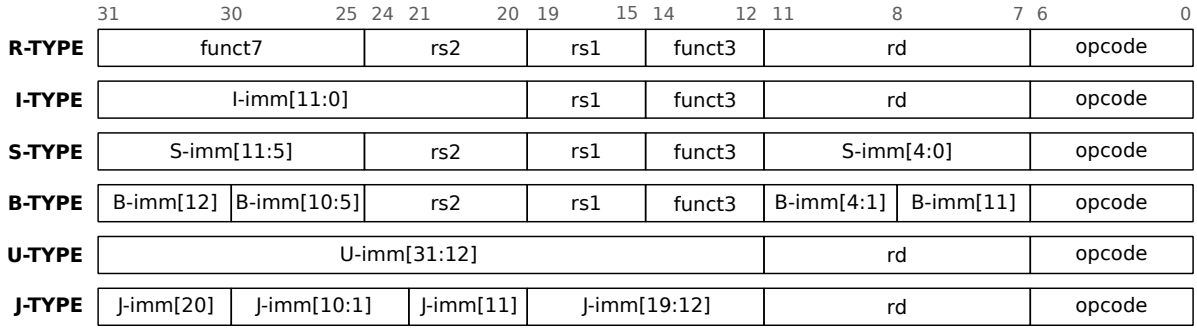

FIGURE 2: Instruction formats

The `funct7`, `funct3`, and `opcode` portions encode the type of instructions (*e.g.* `add`, `sub`, `beq`). The `rs1`, `rs2`, and `rd` portions encode the registers addresses (operands and destination respectively), and the `*-imm[]` portions encode immediate values. We can distinguish between 5 immediate formats, which values encoded in the various instruction formats are extended to 32 bits following the encoding presented on FIGURE 3 (here `inst[]` refers to the portions in the instruction format).

FIGURE 4 details the 25 instructions that are supported by the *mini-riscv*, as well as the instruction format associated with each of them.

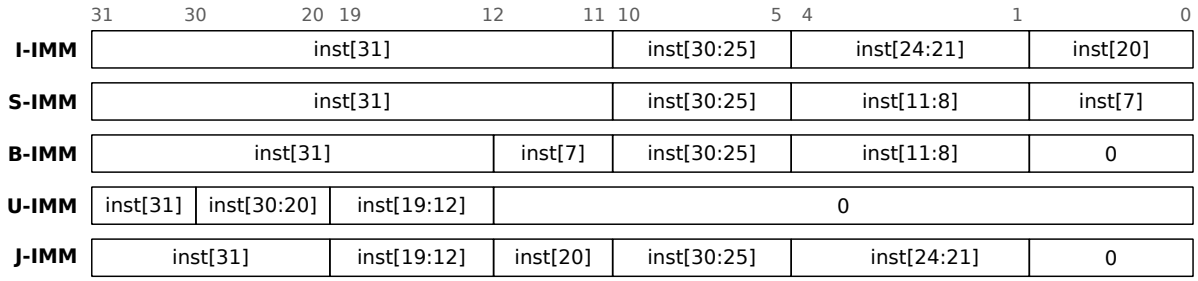

FIGURE 3: Immediate formats

U-imm[31:12]				rd	0110111	**LUI**
J-imm[20 | 10:1 | 11 | 19:12]				rd	1101111	**JAL**
I-imm[11:0]		rs1	000	rd	1100111	**JALR**
B-imm[12|10:5]	rs2	rs1	000	B-imm[4:1|11]	1100011	**BEQ**
I-imm[11:0]		rs1	010	rd	0000011	**LW**
S-imm[11:5]	rs2	rs1	010	S-imm[4:0]	0100011	**SW**
I-imm[11:0]		rs1	000	rd	0010011	**ADDI**
I-imm[11:0]		rs1	010	rd	0010011	**SLTI**
I-imm[11:0]		rs1	011	rd	0010011	**SLTIU**
I-imm[11:0]		rs1	100	rd	0010011	**XORI**
I-imm[11:0]		rs1	110	rd	0010011	**ORI**
I-imm[11:0]		rs1	111	rd	0010011	**ANDI**
0000000	shamt	rs1	001	rd	0010011	**SLLI**
0000000	shamt	rs1	101	rd	0010011	**SRLI**
0100000	shamt	rs1	101	rd	0010011	**SRAI**
0000000	rs2	rs1	000	rd	0110011	**ADD**
0100000	rs2	rs1	000	rd	0110011	**SUB**
0000000	rs2	rs1	001	rd	0110011	**SLL**
0000000	rs2	rs1	010	rd	0110011	**SLT**
0000000	rs2	rs1	011	rd	0110011	**SLTU**
0000000	rs2	rs1	100	rd	0110011	**XOR**
0000000	rs2	rs1	101	rd	0110011	**SRL**
0100000	rs2	rs1	101	rd	0110011	**SRA**
0000000	rs2	rs1	110	rd	0110011	**OR**
0000000	rs2	rs1	111	rd	0110011	**AND**

FIGURE 4: List of instruction supported by the *mini-riscv*

2.1.1 BRANCHES

Unconditional branches instructions—also called jumps—JAL (*Jump And Link*) and JALR (*Jump And Link Register*), are presented in FIGURE 5. They modify the program counter.

- The **JAL** instruction uses the J-TYPE format, where the 20 bits immediate (`offset`) encodes the range of the branch relatively to the current value of the program counter: the `offset` is extended on 32 bits signed with the J-IMM immediate format, and is added to the program counter to form the branch destination address. The address of the instruction following the branch (`pc+4`) is saved in the destination register (`dest`).

- The **JALR** instruction uses the I-TYPE format. It performs an unconditional branch that is independent from the value of the program counter: the 12 bits immediate (`offset`) is extended on 32 bits signed with the I-IMM immediate format, and is added to the `base` register to form the

J-imm[20 | 10:1 | 11 | 19:12]			rd	opcode
OFFSET[20:1]			**DEST**	**JAL**
I-imm[11:0]	rs1	funct3	rd	opcode
OFFSET[11:0]	**BASE**	**0**	**DEST**	**JALR**

FIGURE 5: Instructions de sauts inconditionnels

B-imm[12|10:5]	rs2	rs1	funct3	B-imm[4:1|11]	opcode
OFFSET[12,10:5]	**SRC2**	**SRC1**	**BEQ**	**OFFSET[11,4:1]**	**BRANCH**

FIGURE 6: Instruction de branchement conditionnels

branch destination address. The address of the instruction following the branch (`pc+4`) is saved in the destination register (`dest`).

- The **BEQ** (*Branch On Equal*) instruction is presented in FIGURE 6. It performs a conditional branch, that is made relatively to the value of the program counter. It uses the B-TYPE instruction format, where the 12 bits immediate (`offset`) encodes the range of the branch. The `offset` is extend on 32b bits signed with the B-IMM immediate format, and is added to the program counter to form the branch destination address. If the branching condition is satisfied—*i.e.* the `src1` and `src2` registers are equal—the branch is taken and the program counter should point to the branch destination address, otherwise the branch is not taken and the program counter should point to the instruction following the branch (`pc+4`).

2.1.2 ACCESS TO THE DATA MEMORY

Instruction to access the data memory—LW (*Load Word*) and SW (*Store Word*)—are presented at FIGURE 7. The operate between the register file and the data memory.

I-imm[11:0]		rs1	funct3	rd	opcode
OFFSET[11:0]		**BASE**	**LW**	**DEST**	**LOAD**
S-imm[11:5]	rs2	rs1	funct3	S-imm[4:0]	opcode
OFFSET[11:5]	**SRC**	**BASE**	**SW**	**OFFSET[4:0]**	**STORE**

FIGURE 7: Instruction to access the data memory

- The **LW** instruction uses the I-TYPE format, where the 12-bit immediate (`offset`) encodes the read address relatively to the content of the `base` register: the `offset` is extended on 32-bits signed and added to the `base` register to form the read address. The value read from the data memory at this address is saved in the `dest` register.

- The **SW** instruction uses the S-TYPE format, where the 12-bit immediate (`offset`) encodes the write address relatively to the content of the `base` register: the `offset` is extended on 32-bit signed and added to the `base` register to form the write address. The value of the `src` register is written in memory at this address.

2.1.3 ARITHMETICAL & LOGICAL OPERATIONS

Arithmetical and logical operations are divided in three flavors: one uses the R-TYPE format, another uses the I-TYPE format, and the last one uses the U-TYPE format.

U-imm[31:12]	rd	opcode
U-imm[31:12]	**DEST**	**LUI**

FIGURE 8: LUI instruction

I-imm[11:0]	rs1	funct3	rd	opcode
I-imm[11:0]	**SRC**	**ADDI/SLTI[U]/ ANDI/ORI/XORI**	**DEST**	**OP-IMM**

FIGURE 9: Arithmetical and logical instructions operating on immediates

funct7	rs2	rs1	funct3	rd	opcode
0000000	**SRC2**	**SRC1**	**ADD/SLT[U] AND/OR/XOR**	**DEST**	**OP**
0100000	**SRC2**	**SRC1**	**SUB**	**DEST**	**OP**

FIGURE 10: Arithmetical and logical instructions operating on registers

funct7	rs2	rs1	funct3	rd	opcode
0000000 0100000	**SHAMT[4:0]**	**SRC**	**SLL/SRL SRA**	**DEST**	**OP**
I-imm[11:5]	I-imm[4:0]	rs1	funct3	rd	opcode
0000000 0100000	**SHAMT[4:0]**	**SRC**	**SLLI/SRLI SRAI**	**DEST**	**OP-IMM**

FIGURE 11: Shift instructions

- The **LUI** (*Load Upper Immediate*) instruction is presented at FIGURE 8. It uses the U-TYPE instruction format. The instruction puts the first 20 bits (`U-imm[31:12]`) of its immediate in the 20 most significant bits of the destination register `rd`, and fill the rest with zeros.

- Arithmetical and logical instructions that operate on *immediates* are presented at FIGURE 9. They use the I-TYPE instruction format, and share the same opcode (`OP-IMM`). **ADDI**, **ANDI**, **ORI**, and **XORI** instructions respectively operate an *addition*, a logical *and*, a logical *or* and a logical *xor* between the content of the `src` register and the immediate value `I-imm` extended in 32-bit signed. **SLTI** and **SLTIU** (*Set Less Than Immediate* and *Set Less Than Immediate Unsigned*) compare the value of the `src` register with the immediate value `I-imm` extended on 32-bit signed or unsigned. If `src < I-imm` then `dest` equals 1, else `dest` equals 0.

- Arithmetical and logical instructions that operate on *registers* are presented at FIGURE 10. They use the R-TYPE instruction format, and share the same opcode (`OP`). **ADD**, **SUB**, **AND**, **OR**, and

XOR instructions operate respectively an *addition*, a *subtraction*, a logical *and*, a logical *or*, and a logical *xor* between the content of the `src1` and of the `src2` registers. Note that the SUB instruction only differs from the ADD instruction by the value of the `funct7` field. **SLT** and **SLTU** instructions (*Set Less Than* and *Set Less Than Unsigned*) compare the value of the `src1` register with the value of the `src2` register (signed and non-signed respectively). If `src1 < src2` then `dest` equals 1, else `dest` equals 0.

- Shifts instructions are presented at FIGURE 11. there are three types of shifts: Left shift (**SLL**: *Shift Left Logical*), logical right shift (**SRL**: *Shift Right Logical*), and arithmetical right shift (**SRA**: *Shift Right Arithmetic*). There are also equivalent instructions operating on immediates (**SLLI**, **SRLI**, **SRAI**). The instruction consists in shifting the content of the `src` register by the value of `shamt` (5 first bits of the `rs2` register, or of the `I-imm` immediate), and saving the result ins the `dest` register. Note that the SRA[I] instruction differs from the SRL[I] instruction only by the value of the `funct7` field.

2.2 PROGRAMMING

2.2.1 MEMORY INTERFACES

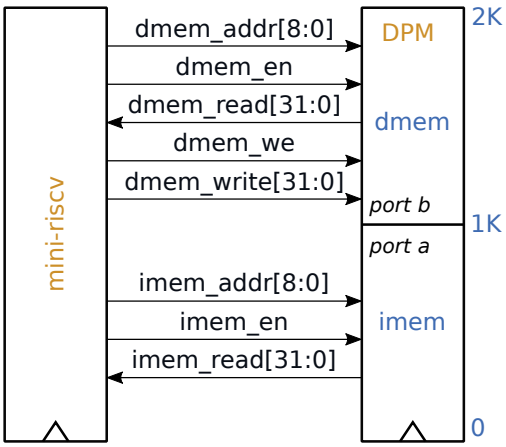

FIGURE 12: Memory interfaces

The memory subsystem in the *mini-riscv* (see FIGURE 12) is composed of a byte-addressable dual-port memory. This memory is separated in two address spaces of `1kB`: the address space of the instruction memory (`imem`) is between `0` and `1kB` on *port a*, and the address space of the data memory (`dmem`) is between `1kB` and `2kB` on *port b*. The memory is an instance of the `dpm` entity, described in the `dpm.vhd` VHDL file (this file is part of the work repository). It is initialized at the beginning of the simulation from a `.hex` file containing the list of instructions and data if the program in hexadecimal format. The memory has a 1 cycle read and write latency.

2.2.2 COMPILING

Programming the *mini-riscv* consists in filling its instruction memory with 32-bit words representing the program instructions, which should comply with the instruction format presented previously. Programs can be written in RISC-V assembly and compiled with **gcc** by using the Makefile provided in the `asm/` folder. However, note that the use of *gcc* is limited by the truncated instruction set of the *mini-riscv*.

```
% make help # Display the help
% make riscv BENCHMARK=<f> # Compile the program <f> for mini-riscv
```

2.2.3 ASSEMBLY

You will find assembly code examples in the `riscv_basic.S` file, which test the basic features of the processor, and in the `riscv_fibo.S` file, which computes the first 20 iterations of the Fibonacci series. Use these programs in your test-bench to validate the behavior of your processor. Notice the use of *pseudo-instructions*: `nop`, `li`, and `beqz`, which are converted by the compiler:

```
li rd, imm[31:0]
```

```
lui tmp, imm[31:12]
ori rd, tmp, imm[11:0]
```

```
beqz rs, offset
```

```
beq rs, x0, offset
```

```
nop
```

```
addi x0, x0, 0
```

2.2.4 REGISTERS

TABLE 1: Registers naming convention

Name	Number	Use
zero	x0	The value 0
ra	x1	Functions return address
sp	x2	Stack Pointer
gp	x3	Global Pointer
tp	x4	Thread Pointer
t0-t2	x5-x7	Temporaries
s0-s1	x8-x9	Save
a0-a7	x10-x17	Functions arguments
s2-s11	x18-x27	Save
t3-t6	x28-x31	Temporaries

3 MICROARCHITECTURE

This chapter presents the microarchitecture of the *mini-riscv* that you must design as part of this laboratory. It implements the *mini-riscv* architecture presented in the previous chapter with a 5-stages pipeline. First, the description of the modules will allow you to design the *mini-riscv* components. Then, the description of each pipeline stage will allow you to design the *core*. Finally, details regarding the management of conflicts in the pipeline will allow you to make your final system work. Note that the constants used in a non-generic fashion in the modules are defined in a package (`riscv_pkg.vhd`) that is provided in the work repository. To include of these constant in a module, use the following code snippets (SOURCE 1):

SOURCE 1: *Package* contenant les constantes

```
library work;
use work.riscv_pkg.all;
```

The core interface is presented at SOURCE 2. The `*_imem_*` signals constitute the interface with the instruction memory, and the `*_dmem_*` signals constitute the interface with the data memory.

SOURCE 2: VHDL entity of the *mini-riscv* (*core*)

```
entity riscv_core is
  port (
    i_rstn       : in  std_logic;
    i_clk        : in  std_logic;
    o_imem_en    : out std_logic;
    o_imem_addr  : out std_logic_vector(8 downto 0);
    i_imem_read  : in  std_logic_vector(31 downto 0);
    o_dmem_en    : out std_logic;
    o_dmem_we    : out std_logic;
    o_dmem_addr  : out std_logic_vector(8 downto 0);
    i_dmem_read  : in  std_logic_vector(31 downto 0);
    o_dmem_write : out std_logic_vector(31 downto 0);
    -- DFT
    i_scan_en    : in  std_logic;
    i_test_mode  : in  std_logic;
    i_tdi        : in  std_logic;
    o_tdo        : out std_logic);
end entity riscv_core;
```

3.1 MODULES

This section deals with the modules of the *mini-riscv*. That is, the ALU (composed of a generic adder, shifter, and logical operations), a program counter (PC), and a register file (RF).

3.1.1 ADDER

The *adder* module is used in the ALU. It is based on the ripple-carry design technique, which relies on half-adder in series propagating the carry at each stage to perform an multi-bits add operation. FIGURE 13 shows the diagram of the *half-adder*, and FIGURE 14 shows the diagram of the *adder*. SOURCE 3 shows the VHDL entity description of the *adder*.

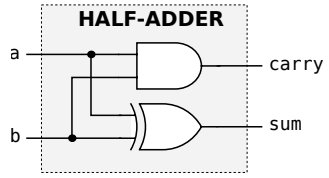

FIGURE 13: *half-adder*

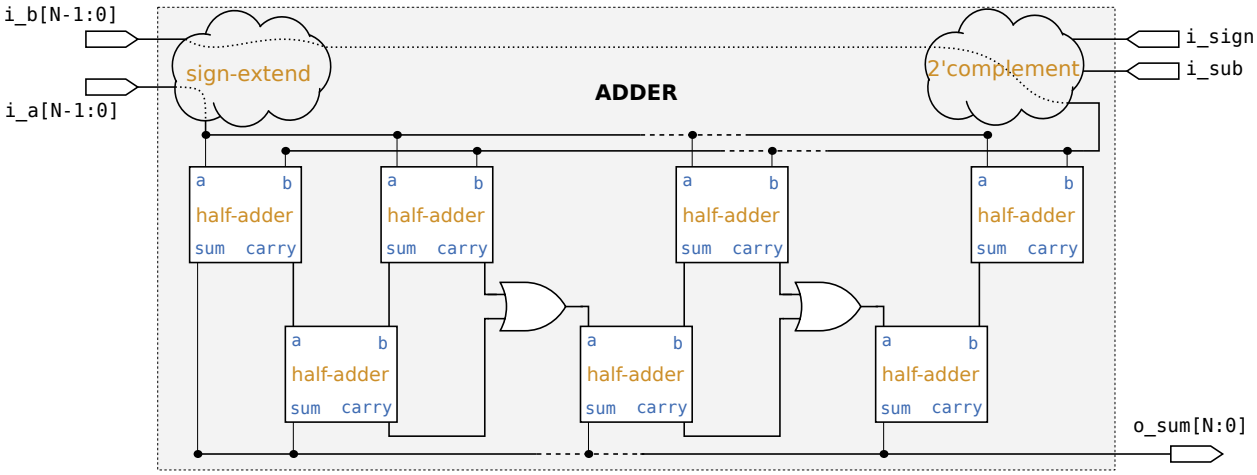

FIGURE 14: *adder*

The adder module is combinational. It uses a generic parameter `N` which defines the size of the data signals, as well as the number of half-adder that are required. By default, `N = 32`, The adder perform the sum of `i_a` (`N` bits) and `i_b` (`N` bits) and puts the result on the `o_sum` (`N+1` bits) signal. When the `i_sign` input equals 0, the operations are performed on *unsigned* values. Similarly, when the the `i_sign` input equals 1, the operations are performed on *signed* values. When the `i_sub` input equals 0, the operation to perform is an *addition* (`i_a + i_b`). Similarly, when the `i_sub` input equals 0, the operation to perform is a *subtraction* (`i_a - i_b`). The subtraction is performed by using the 2's complement of the `i_b` signal.

SOURCE 3: VHDL entity of the *adder*

```
entity riscv_adder is
  generic (N : positive := 32);
  port (
    i_a    : in  std_logic_vector(N-1 downto 0);
    i_b    : in  std_logic_vector(N-1 downto 0);
    i_sign : in  std_logic;
    i_sub  : in  std_logic;
    o_sum  : out std_logic_vector(N downto 0));
end entity riscv_adder;
```

3.1.2 ALU

The ALU module perform arithmetical and logical operations on the `i_src1` and `i_src2` data inputs, and puts a result on the `o_res` output, which varies according the values of the `i_opcode` and the `i_arith` control inputs. The ALU does not contain generic parameters. FIGURE 15 shows the diagram of the ALU, and SOURCE 4 shows its VHDL entity description.

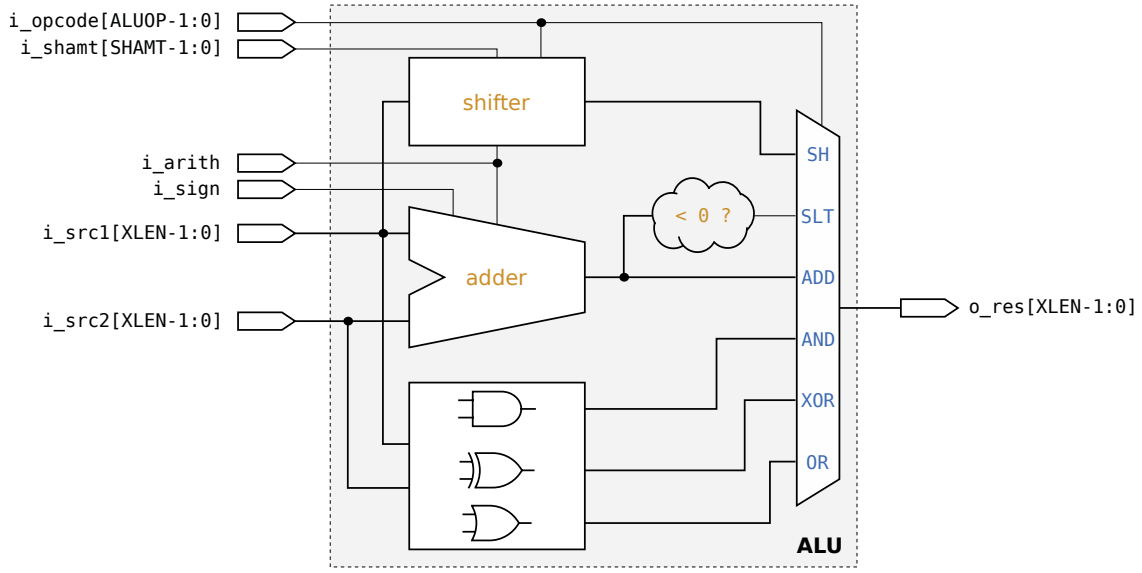

FIGURE 15: ALU

The ALU module is combinational. It is composed of three main blocs that perform the following operations (see TABLE 2 for a summary):

- **The adder bloc** is an instance of the adder module. It performs the addition and the subtraction operations on the `i_src1` and the `i_src2` signals. Note that the `i_arith` signal of the ALU is connected to the `i_sub` signal of the adder. Part of the *set-less-than* (SLT) instructions are performed in the adder: when the result of the adder is negative, the output equals 1, otherwise it equals 0.

- **The shifter bloc** performs the left shifts (SL) and the right shifts (SR) instructions on the `i_src1` signals. The number of bits to shifts are specified by the `i_shamt` input. The type of shift is derived from the control inputs `i_opcode` and `i_arith`: The right shift must be *logical* (zero padding) when `i_arith` equals 0, and arithmetic (MSB padding) when `i_arith` equals 1. The left shift is always logical.
- The logical bloc performs the logical operations AND, OR, and XOR on the `i_src1` and `i_src2` signals.

SOURCE 4: VHDL entity of the ALU

```
entity riscv_alu is
  port (
    i_arith  : in  std_logic;
    i_sign   : in  std_logic;
    i_opcode : in  std_logic_vector(ALUOP_WIDTH-1 downto 0);
    i_shamt  : in  std_logic_vector(SHAMT_WIDTH-1 downto 0);
    i_src1   : in  std_logic_vector(XLEN-1 downto 0);
    i_src2   : in  std_logic_vector(XLEN-1 downto 0);
    o_res    : out std_logic_vector(XLEN-1 downto 0));
end entity riscv_alu;
```

TABLE 2: Operations performed in the ALU

Opcode	Bloc	Condition	Opération
ALUOP_ADD	Adder	i_arith = 0	addition
ALUOP_ADD	Adder	i_arith = 1	subtraction
ALUOP_SLT	Adder		1 if adder_res < 0, else 0
ALUOP_SL	Shifter		left shift
ALUOP_SR	Shifter	i_arith = 0	logical right shift
ALUOP_SR	Shifter	i_arith = 1	arithmetical right shift
ALUOP_XOR	Logique		XOR
ALUOP_OR	Logique		OR
ALUOP_AND	Logique		AND

3.1.3 PROGRAM COUNTER

The program counter (PC) is used in the *mini-riscv* to control the instruction memory address bus. The output of the PC always points to the address in memory where the next instruction to be executed is stored. The PC contains two generic parameters: `RESET_VECTOR` and `XLEN`. by default, `XLEN = 32`, and `RESET_VECTOR = 16#00000000#`. FIGURE 16 shows the diagram of the PC, and SOURCE 5 shows its VHDL entity.

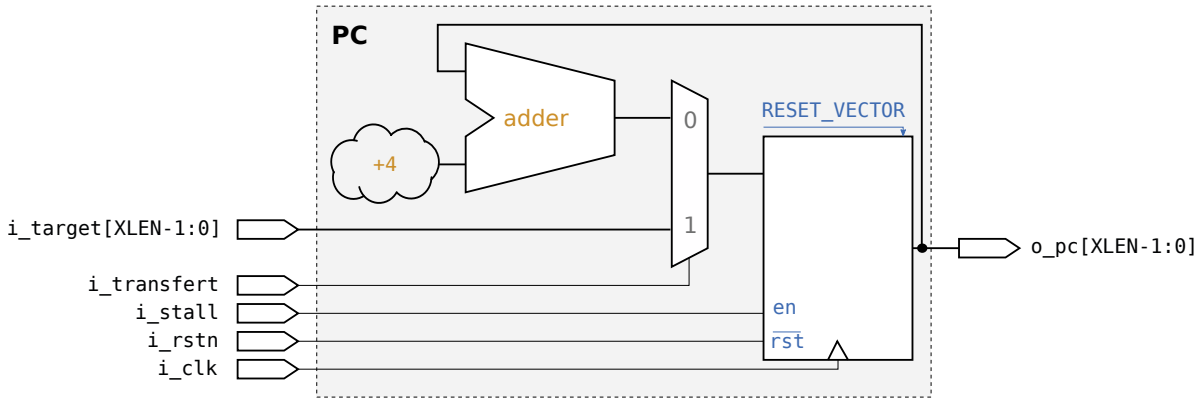

FIGURE 16: *Program Counter* (PC)

The PC is a sequential module. It must behave as follow: At each rising edge of the clock input (`i_clk`), the output (`o_pc`) is updated. We denote `pc` the signal at the output of the register. The reset is asynchronous: `pc` is initialized by `RESET_VECTOR` when `i_rstn` equals 0. If `i_transfert` equals 1, `pc` is driven by `i_target`, else `pc` is driven by the output of the adder (`pc + 4`). When `i_stall` equals 1, `pc` keep its previous value.

SOURCE 5: VHDL entity of the PC

```
entity riscv_pc is
  generic (RESET_VECTOR : natural := 16#00000000#);
  port (
    i_clk       : in  std_logic;
    i_rstn      : in  std_logic;
    i_stall     : in  std_logic;
    i_transfert : in  std_logic;
    i_target    : in  std_logic_vector(XLEN-1 downto 0);
    o_pc        : out std_logic_vector(XLEN-1 downto 0));
end entity riscv_pc;
```

3.1.4 REGISTER FILE

The register file (RF) controls the read and write accesses to the 32 32-bit registers of the *mini-riscv*. It contains two generic parameters, `REG` and `XLEN`, which define the size of the address and data signals. By default `REG = 5` and `XLEN = 32`. FIGURE 17 shows the diagram of the RF, and SOURCE 6 shows its VHDL entity description. The RF is a sequential module. It must behave as follow: each address pointed by the (`i_addr_*`) inputs corresponds to a register accessible by the (`i_data_w` and `o_data_*`) signals. The address `0x0` always contains the value 0. The reset is asynchronous: every register is reset to 0 when `i_rstn` equals 0. The writing of data is performed on the rising edge of the clock input (`i_clk`). The register pointed by the `i_addr_w` address is driven by: `i_data_w` is the `i_we` signal equals 1; its previous value otherwise. The reading of data is performed on the rising edge of the clock input (`i_clk`). The

outputs `o_data_ra/rb` are driven by the registers pointed by the `i_addr_ra/rb` addresses. When a reading address is equal to a writing address, the outputs must be driven by the value of `i_data_w`.

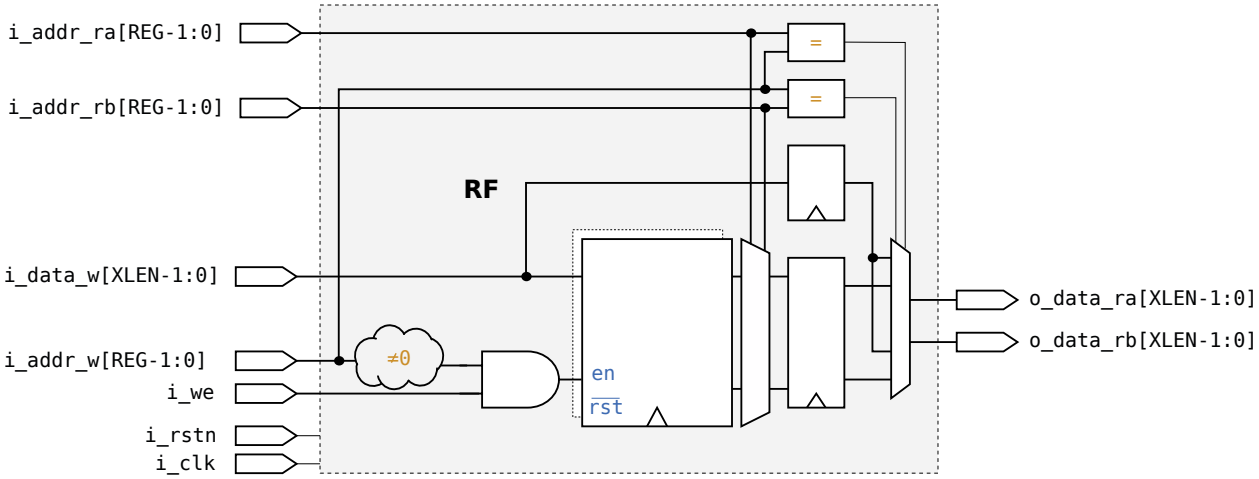

FIGURE 17: *Register File* (RF)

SOURCE 6: VHDL entity of the RF

```
entity riscv_rf is
  port (
    i_clk     : in  std_logic;
    i_rstn    : in  std_logic;
    i_we      : in  std_logic;
    i_addr_ra : in  std_logic_vector(REG-1 downto 0);
    o_data_ra : out std_logic_vector(XLEN-1 downto 0);
    i_addr_rb : in  std_logic_vector(REG-1 downto 0);
    o_data_rb : out std_logic_vector(XLEN-1 downto 0);
    i_addr_w  : in  std_logic_vector(REG-1 downto 0);
    i_data_w  : in  std_logic_vector(XLEN-1 downto 0));
end entity riscv_rf;
```

3.2 PIPELINE

Instruction level parallelism (ILP) consists in concurrently computing the steps of an instruction with the steps of previous instructions in a program. *Pipelining* is a design technique implementing the ILP at the hardware level: each step of an instruction is associated with a stage of the pipeline. The microarchitecture of the *mini-riscv* is based on a 5-stages pipeline, splitting the instructions in 5 steps as shown in FIGURE 18:

- ***Fetch* (IF)**: Read the next instruction from the instruction memory
- ***Decode* (ID)**: Determine the type of instruction and read operands in the RF.
- ***Execute* (EX)**: Perform the operations on the operands
- ***Memory* (ME)**: If required, access the data memory to read or write a value.
- ***Write-Back* (WB)**: Write the result in the RF.

FIGURE 18: 5-stages pipeline of the *mini-riscv*

3.2.1 INSTRUCTION FETCH (IF)

In the IF stage, the *mini-riscv* fetches a new instruction in the instruction memory, as shown in FIGURE 19. The PC provides the address of the next instruction to process at each cycle. Consequently, the output of the PC must be interfaced with the instruction memory bus. The value returned by the memory contains the next instruction to process, which is saved in the `IF/ID` state registers. Note that many control signals come from the EX stage.

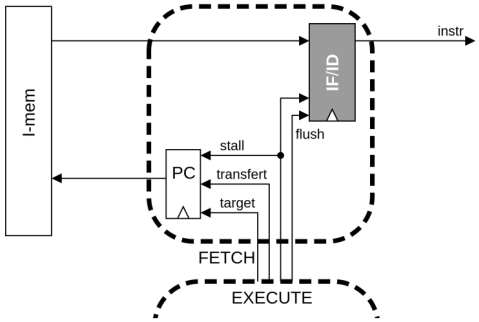

FIGURE 19: **IF** pipeline stage

3.2.2 INSTRUCTION DECODE (ID)

In the IS stage, the *mini-riscv* generates all the control and data signals from the instruction word provided by the IF stage, and produce the operands to be used in the EX stage. In particular, this stage:

- Generates the signals that identify an instruction (`opcode`, `funct3`, `funct7`).
- Generates the address signals allowing to read and write operands in the RF (`rs1_addr`, `rs2_addr`, and `rd_addr`). The write address must be passed along the pipeline until the WB stage.
- Access the RF from the read addresses to fetch the operands. Note that the RF a 1 cycle latency.
- Generate the immediate values with the right formats (see FIGURE 2).
- Generate all the control signals (*e.g.* is it a branch ? is it a data memory access ? should the result be written back in the RF ? *etc.*).

All the control signals and immediate values must then be saved in the `ID/EX` state registers.

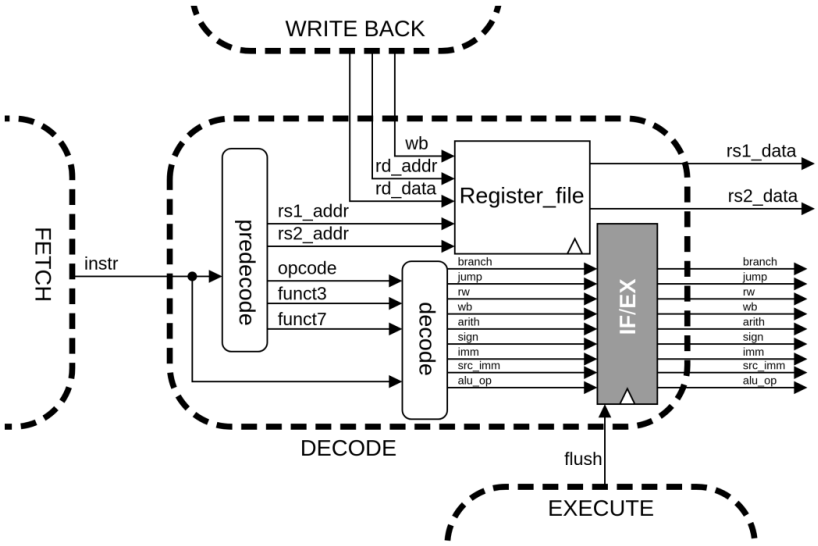

FIGURE 20: **ID** pipeline stage

3.2.3 EXECUTE (EX)

In the EX stage, the *mini-riscv* executes the operation of the instruction. Arithmetical and logical operations are performed in the ALU, and branches outcome are decided, as shown in FIGURE 21. In particular, this stage is responsible for:

- Generating the arithmetical and logical operations results.
- Generating the read or write address for the data memory access.
- Determine the branch outcome.
- Generate the branches destination address.

Note that the operands on which the operations are performed must be computed prior to this stage, depending on the instruction format and control signals, from the ID stage.

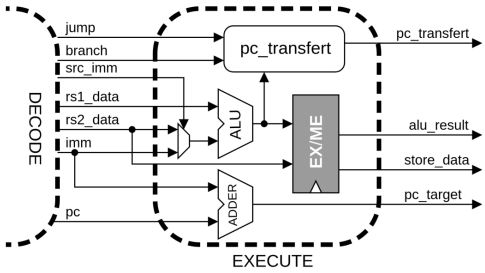

FIGURE 21: **EX** pipeline stage

3.2.4 MEMORY ACCESS (ME)

In the ME stage, the *mini-riscv* accesses the data memory only in case of a load (`lw`) or a store (`sw`) instruction. Results are transmitted to the `ME/WB` registers as shown in FIGURE 22.

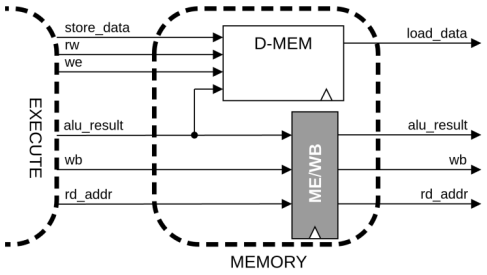

FIGURE 22: **ME** pipeline stage

3.2.5 WRITE-BACK (WB)

In the WB stage, the *mini-riscv* write (if necessary) the result of the instruction in the RF at the address pointed by `rd_addr`. In case of a load instruction, the value read from the data memory (`dmem_read`) must be used, otherwise, the value coming from the ALU (`alu_result`) must be used, as shown in FIGURE 23.

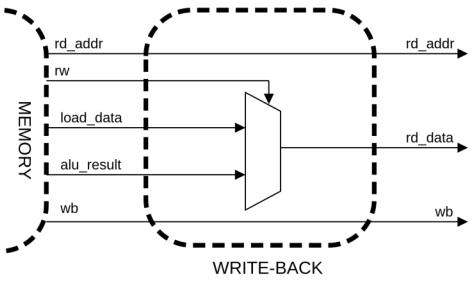

FIGURE 23: Étage de pipeline **WB**

3.3 HAZARDS

Instruction level parallelism implemented in a pipeline causes issues, called hazards, that alter the sequentiality of operations in a program. To restore the original sequentiality of the program, some results may be *forwarded* between pipeline stages, some pipeline stages may be *stalled*, and some other may be *flushed*. The microarchitecture of the *mini-riscv* must implement the hardware mechanism enabling the proper management of hazards. Note that the test program (`riscv_fibo.asm`) was designed to highlight the proper behavior of the mini-riscv in the presence of hazards. In the *mini-riscv*, there are three categories of hazards:

- **Structural hazards** occur when two instructions try to access the same resource at the same cycle. In the *mini-riscv*, it occurs when two instructions concurrently access a value at the same address in the RF. To alleviate this issue, the RF was designed such that the written value is duplicated in an independent register.

- **Data hazards** occur when an instruction depends on the result of a previous instruction. In the *mini-riscv*, it occurs when an instruction tries to read an operand in the RF (ID stage) that was not yet updated by a preceding instruction (WB stage). The majority of these hazards are solved by forwarding the result of an instruction from the MW and WB stages to the EX stages. In the case of a load (`lw`) instruction, forwarding is not enough because the result is only known at the ME stage. In the *mini-riscv*, this type of hazard is solved by applying a 1 cycle stall to the pipeline, in addition to forwarding. FIGURE 24 shows how stalls should be implemented in the pipeline.

- **Control hazards** occur each time the PC is altered by a branch. 3 cycles are required before a `jal`, `jalr`, or `beq` instruction alters the PC, because the outcome is computed at the EX stage. During

the two previous cycles, two instructions are fetched into the pipeline. If the branch is indeed taken, these two instructions must be flushed out of the pipeline to prevent them from altering the state of the processor. The *mini-riscv* resolves conditional branches (`beq`) with a *predict-not-taken* strategy, which assumes that, by default, the branch is not taken. Instructions following the branch are fetched and start their execution. At the EX stage, if the branch is taken, the two instructions are flushed, otherwise (as assumed) they can continue their normal execution.

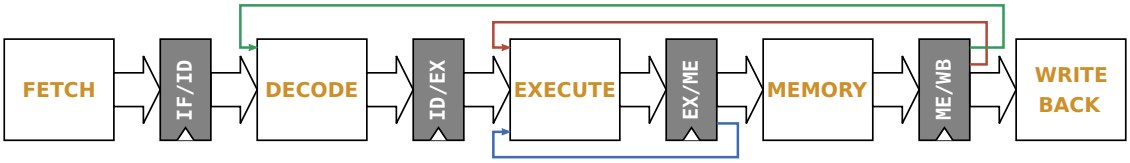

FIGURE 24: Implementation of the *forwarding* in the pipeline