Politecnico di Torino

Collegio di Elettronica, Telecomunicazioni e Fisica

Report for the course Integrated Systems Architecture

Lab 3

Master degree in Electrical Engineering

Group: 12

Carpentieri Nicolò, s301159
D'Agostino Maria Elena, s291097
Marra Angelina, s291099

February 17, 2023

Contents

CHAPTER 1

Lab 3: RISC-V-lite

1.1 Architecture definition

In this lab a RISC-V lite processor is designed, and the instructions developed are the ones required to execute the *minv* application. The entire application is divided into 3 files:

- main.c
- minv.h
- minv.c

In order to generate the full instructions, the compiler requires informations about the memory organization, and a preamble to initialize registers, such as the stack pointer The code to run on the RISC-V-lite processor is:

```
.section .init, "ax"
.global _start
_start:
    .cfi_startproc
    .cfi_undefined ra
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop
    la sp, __stack_top
    add s0, sp, zero
    jal ra, main
el:
    j el
    .cfi_endproc
    .end
```

The simple crt0 file defines the **_start** symbol, which represents the address of the first instruction to execute. Then it loads the global pointer and stack pointer registers (**gp** and **sp**). Finally, it calls the *main* function with jump-and-link instruction **jal** and the execution terminates with an infinite loop (**j el**)

Once done all the steps described before, the following sequence of instructions are obtained:

```
                .data
v:
                .word 0x00000009
                .word 0xffffffd2
                .word 0x00000015
                .word 0xfffffffe
                .word 0x0000000e
                .word 0x0000001a
                .word 0xfffffffd

                .text
                .global _start



_start:
auipc gp,0x1fc18
addi gp,gp,28
auipc sp,0x7fbff
addi sp,sp,-12
add s0,sp,zero
jal ra, main

el:
j el



main:
lui a0,0x10010
addi sp,sp,-16
li a1,7
mv a0,a0
sw ra,12(sp)
jal ra, minv
lw ra,12(sp)
lui a5,0x10010
sw a0,28(a5)
li a0,0
addi sp,sp,16
ret

minv:
lw a2,0(a0)
li a4,1
srai a5,a2,0x1f
xor a2,a5,a2
sub a2,a2,a5
ble a1,a4, branch_first
slli a1,a1,0x2
addi a4,a0,4
add a0,a0,a1
branch_thr:
lw a5,0(a4)
addi a4,a4,4
srai a3,a5,0x1f
xor a5,a3,a5
sub a5,a5,a3
ble a2,a5, branch_sec
mv a2,a5
branch_sec:
bne a0,a4, branch_thr
branch_first:
mv a0,a2
ret
```

1.2 VHDL model

The RISC-V lite processor designed in VHDL by group 12, is shown below.

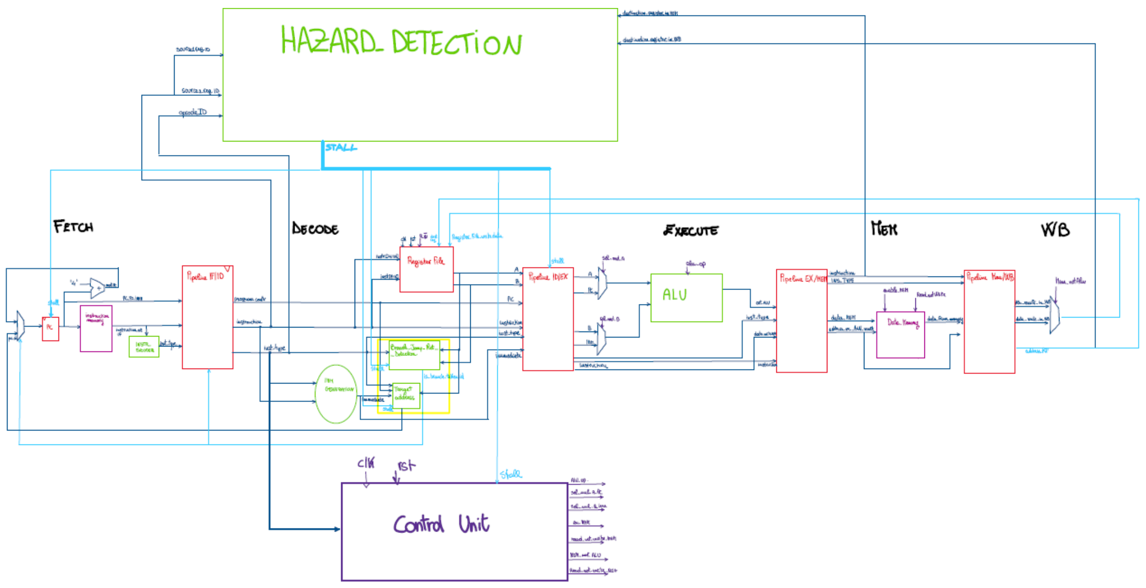

Figure 1.1: RISC V architecture

The model is mainly composed by five pipeline registers and a control unit. The pipeline is a Harvard architecture (separate physical memory for instructions and data). The hardware features that are added to the standard architecture to manage stalls and hazards are Branch Detection and Hazard detection.
Now is presented a description of the various components.

1.2.1 my_pkg.vhd

The instructions supported by the RISC-V-lite processor is a subset of the whole RV32I, required to execute the minv application. The instruction set of RV32I is composed of 32-bit fixed-width instructions, that are organized into six classes (R, I, S, B, U, and J).
An apposite package has been created, that contains all the instructions to be executed in the architecture and some constants useful for the design parameters. The instruction set analyzed is listed below:

AUIPC
ADDI
ADD
JAL
J
LUI
LI
MV
RET
SW
LW
SRAI
XOR
SUB
BLE
SLLI
BNE

1.2.2 Fetch stage

The analysis starts with the fetch stage, represented as black box in the following figure:

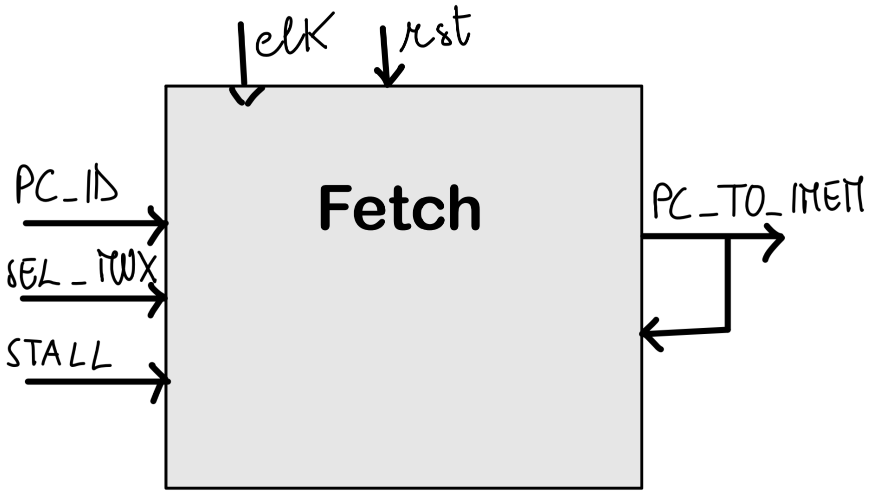

Figure 1.2: Fetch stage block

In the fetch stage is managed the correct selection of program counter that will be provided, as an input, to the Instruction Memory (external to RISC-V architecture).
The signals of this first block are:

- CLK: clock signal;
- RST: asynchronous active high reset signal;
- PC_ID: target address (T.A.);
- SEL_MUX: selection signal of the input multiplexer;
- STALL: input signal to inform that is needed a stall.
- PC_TO_IMEM: that is at the same time an input and an output of the block. As an input signal because it become the next program counter value ($PC = PC + 4$) and as an output because become the address that points to the 'Instruction Memory' that is external to RISC-V architecture.

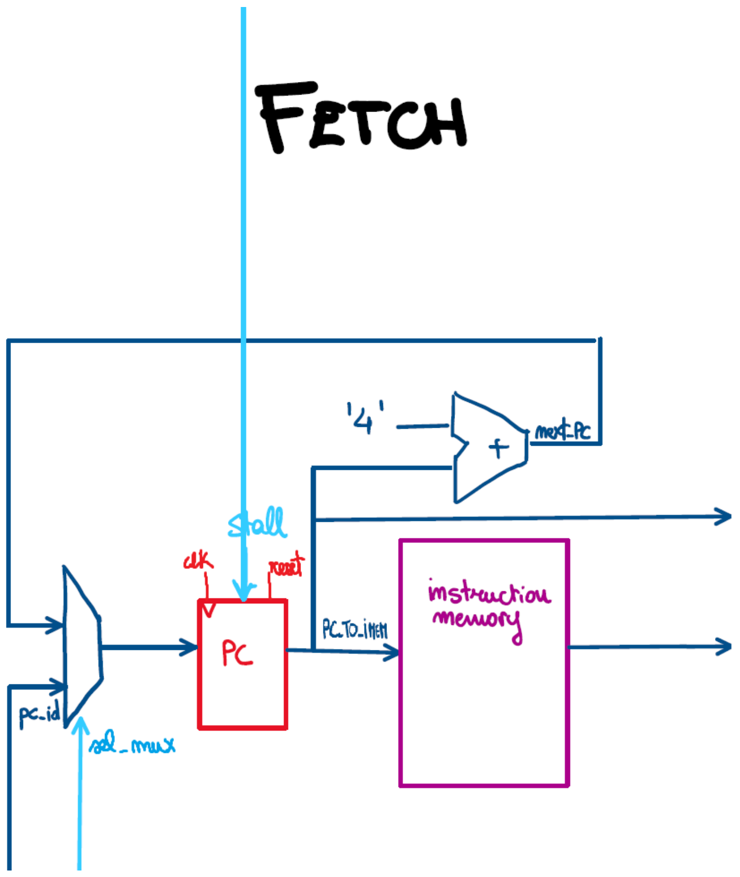

Figure 1.3: Fetch stage

To achieve the aim of the fetch stage is used a 2:1 multiplexer, that chooses the next value of PC between 'next_PC' signal ($PC = PC + 4$), or 'PC_ID' that is the target address ($T.A.$). The selected value, stored in the 'Program Counter' register, will be the address that points to the 'Instruction Memory', which is external to RISC-V architecture.
The first input of the multiplexer is the normal value of the PC that points sequentially to the 'Instruction Memory'. It is generated by incrementing with '4' the previous value of the PC, so the 'PC_TO_IMEM' value become 'next_PC' after the addition. The '4' is due to the byte addressing of the memory.
The second input of the multiplexer comes from the target address (T.A.), coming from the 'Target Address' block contained in the decode stage.
The selection signal of the input multiplexer comes from the 'Branc_Jump_Ret_Detection' unit contained also in the Decode stage, which is responsible for the detection of a jump instruction, a branch instruction, or a RETURN instruction.

1.2.3 Instruction Encoder

This module receives the instruction, and then the opcode and funct field were extrapolated, and generates a signal whose type is *Instruction*, the one defined in *my_pkg*. The figure is shown 1.4

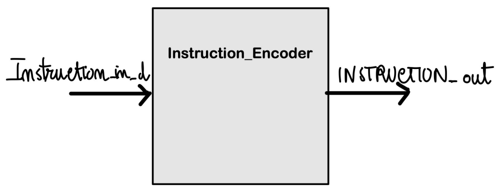

Figure 1.4: Instruction Encoder

1.2.4 Pipeline Register IF/ID

The developed processor is characterized by 5 pipeline registers. The first one is the pipeline register between the fetch stage and decode stage. In this register the sensitive signals are:

- CLK: clock signal;
- reset: reset signal;
- is_branch_taken_ID: this signal is used in order to decided if the instruction in decode stage has to be stalled or not.

During the rise of the clock, if the reset is low, the input sampling occurs; as consequence PC, instruction and instruction type are propagated. If a stall is asserted in our processor, a **LUI** instruction occurs. In this register, this behaviour is notified by means of the signal *is_branch_taken_ID*. This signal is generated by the *BRANCH_JUMP_RET_DETECTION* module, and it generates this signal high in 2 cases:

- if a *stall* signal is asserted;
- if a branch has to be executed.

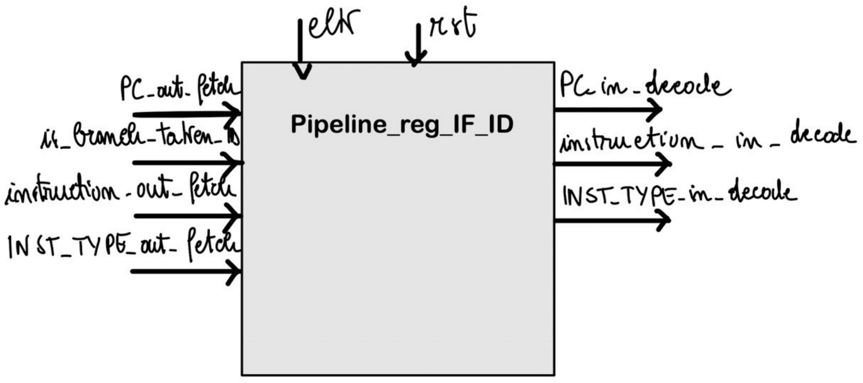

Figure 1.5: pipeline register IF-ID

1.2.5 Decode stage

The second stage of the RISC-V is the Decode stage. The information that come from the Instruction Memory, stored in pipeline register IF/ID, are now used to generate the operands for the Execute stage.

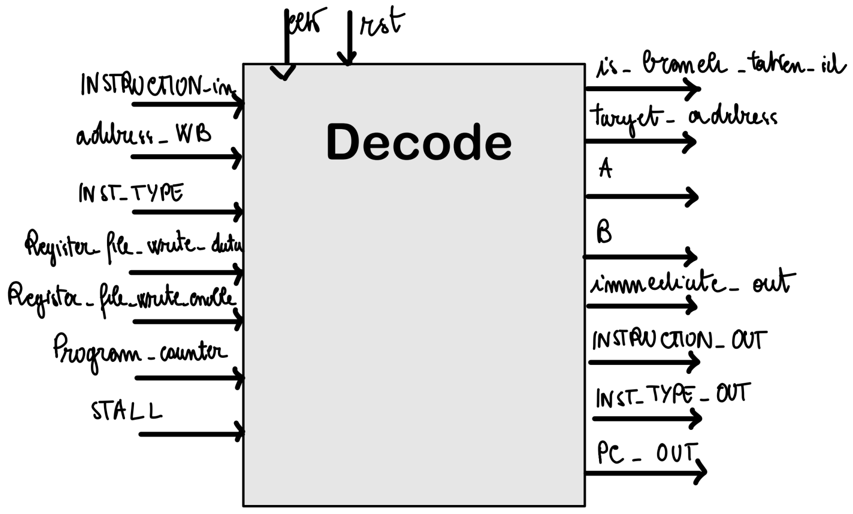

Figure 1.6: Decode stage block

The Decode stage unit has the following interface:

- CLK: clock signal;
- RST: asynchronous active high reset signal;
- INSTRUCTION_in: instruction in the decode stage;
- address_WB: address to write in register-file;
- INST_TYPE: type instruction used in BRANCH_JUMP_RET_DETECTION;
- Register_file_write_data: data to write in register file;
- Register_file_write_enable: write enable for register file from control unit (when is equal to '1' the register file is in read mode);
- Program_counter: program counter used for the fetch of the instruction in the Decode stage;
- STALL: signal of stall;
- is_branch_taken_id: signal that determine if the branch is taken or not;
- target_address: target address for branch;
- A: operand A;
- B: operand B;
- immediate_out: immediate value;

- INSTRUCTION_OUT: instruction to be sent to execute stage;
- INST_TYPE_OUT: instruction type to be sent to execute stage;
- PC_OUT: program counter to be sent to execute stage, and then propagated until writeback stage;

The blocks that compose this stage are shown in figure 1.7 and are: Immediate generation, Branch Detection and Register File.

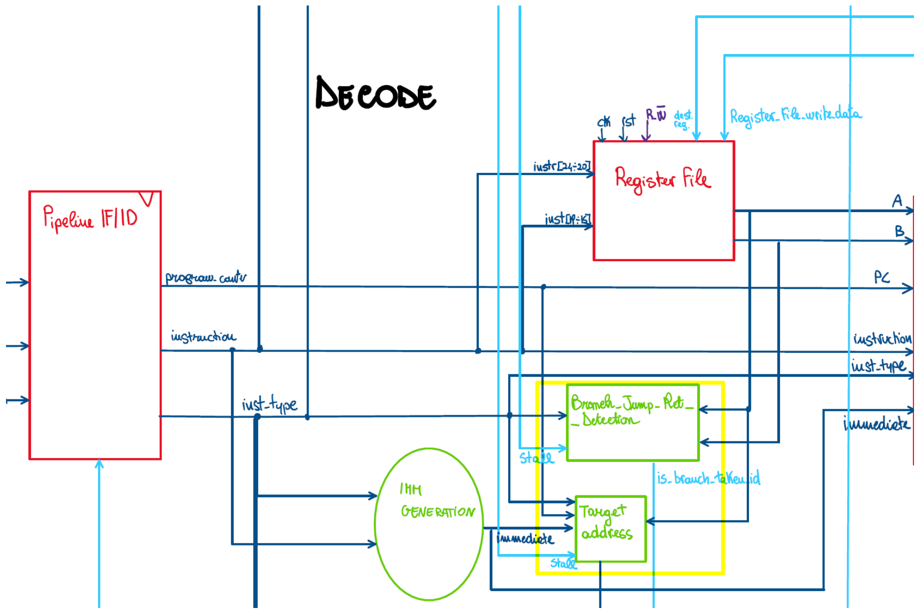

Figure 1.7: Fetch stage

Immediate generation

The immediate generation unit is a block that composes the value of the immediate on 32-bit. The instructions BLE, BNE, JAL, J, MV, SW, LW, LUI, AUIPC, ADDI, LI, SRAI and SLLI contain some fields with immediate values. Those are composed and mixed by the following block to provide the final immediate value that will be used in the Execute stage as one input of the ALU.

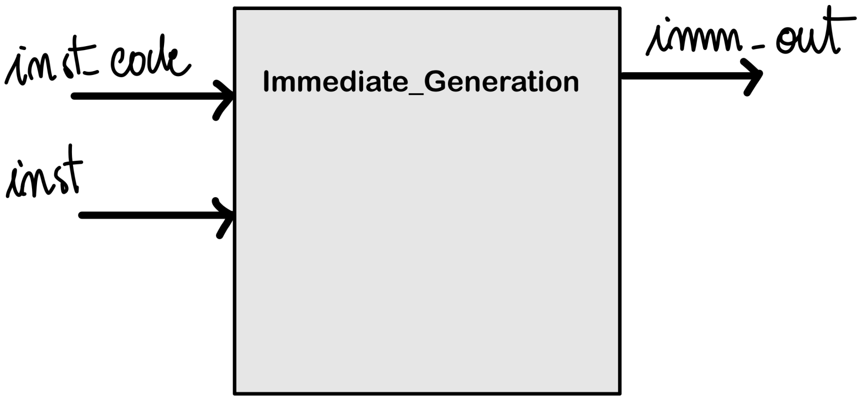

Figure 1.8: immediate generation block

The immediate generation unit is implemented in the Decode stage and has the following interface:

- INST_CODE: that is the input instruction on 32-bit that has been fetched;
- INST: type *instruction*, indicates the name of current instruction in the decode stage;
- IMM_OUT:decoded output signal on 32-bit.

Branch Detection

Thanks to this module, an attempt to anticipate the branch decision has done. Indeed, this is one of the module of decode stage, and in this stage by means of the *BRANCH_JUMP_RET_DETECTION* module, the processor will know whether a branch has to be executed (and also the target address) one cycle before with respect to an architecture where the target address is evaluated in execute phase. In this module, the sensitive signals are: *STALL* and *INST_TYPE*. If a BLE or a BNE is demanded, a check of the content of the 2 source registers is done, and then the evaluation of the correspondent target address is computed. In this module, not only the branch instructions are considered, but also the return and jal instruction is taken into account.

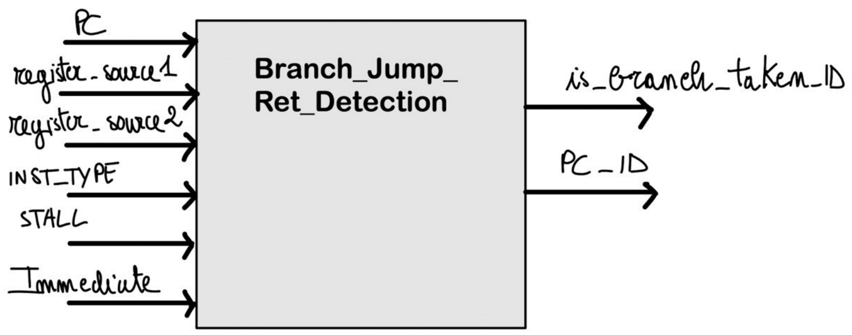

Figure 1.9: Branch detection

Register File

The register file developed in this project is synchronous in writing phase, and asynchronous in reading phase. Also in this module, an attempt of optimization is performed: during the reading phase, if the address of the writing phase is equal to the address of reading phase, the content of the reading data will be available. In this way, a stall between an instruction in WB phase, and another in DECODE phase is prevented. As rule of thumb, if a write enable signal is asserted (generated by the Control Unit), the writing phase is performed, and it will work on the rising of the clock edge. Otherwise, a reading phase is performed, and this stage is developed without any check of the clock edge.

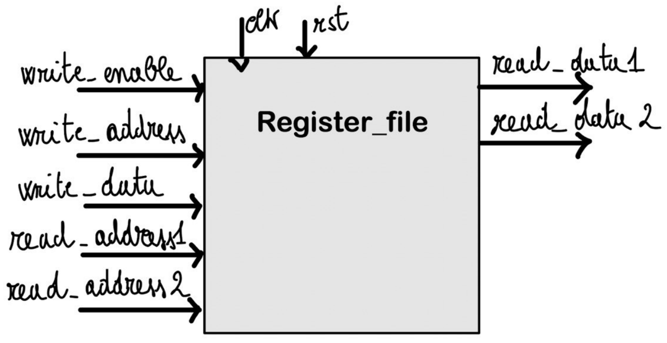

Figure 1.10: Register file

1.2.6 Pipeline Register ID/EX

The developed processor is characterized by 5 pipeline registers. The second one is the pipeline register between the decode stage and execute stage. In this register the sensitive signals are:

- CLK: clock signal;
- reset: reset signal;
- stall this signal is used in order to decided if the instruction in decode stage has to be stalled or not.

During the rise of the clock, if the reset is low, the input sampling occurs; as consequence PC, instruction, instruction type, first operand A, second operand B, immediate are propagated. If a stall is asserted in our processor, a **LUI** instruction occurs.

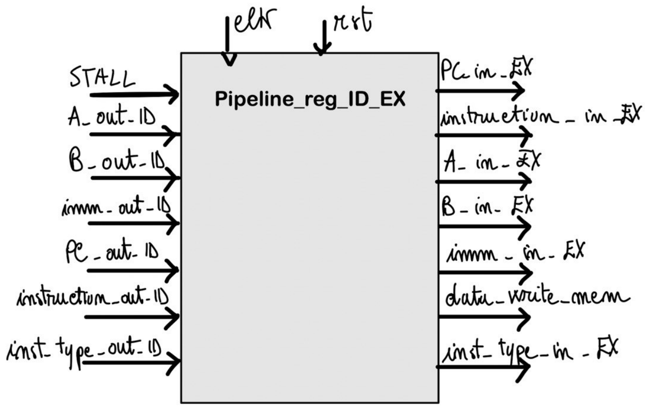

Figure 1.11: pipeline register ID-EX

1.2.7 Execute stage

The execute stage is implemented as shown in Figure 1.13.

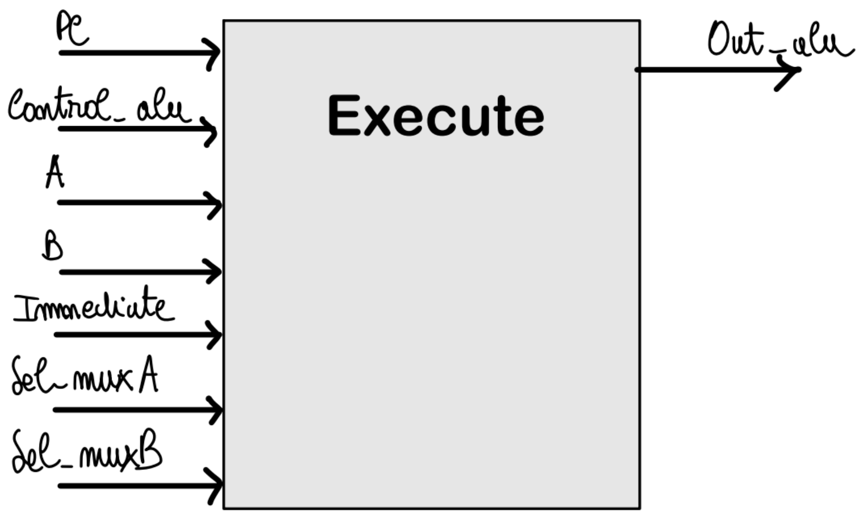

Figure 1.12: Execute stage block

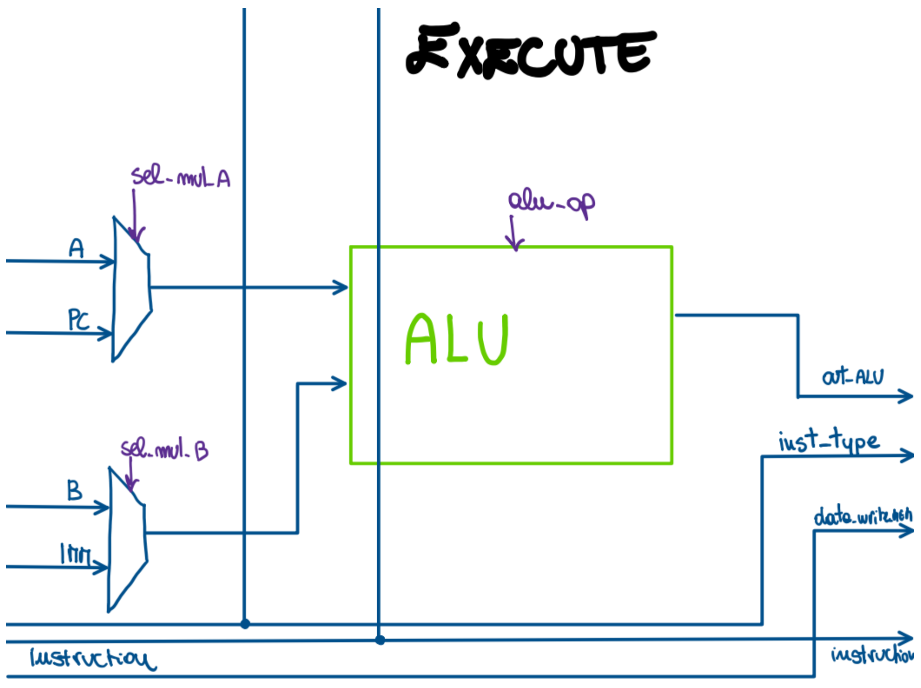

Figure 1.13: Execute stage

As it can be seen in Figure 1.12 the input and output signals for this block are:

- PC: Program Counter;
- Control_alu: control signal for the ALU;

- A: value of the first source register;
- B: value of the second source register;
- Immediate: immediate value;
- Sel_muxA: selection signal of the multiplexer above;
- Sel_muxB: selection signal of the multiplexer below;
- Out_alu: result of the ALU.

The first multiplexer selects between the content of the first data read from the *Register File* (source register 1) and the Program Counter. This second one is propagated through the Decode stage from the Fetch stage, where it was used the fetch the instruction in the Execute stage.
The second multiplexer selects between the content of the second data read from the *Register File* (source register 2) and the Immediate value generated in the decode stage.
The outputs of the two multiplexer are the inputs of the ALU. This last component carries out the operations required by the instruction in execution, according to the signal received from the *Control Unit*. It is able to perform addition, subtraction, logic operations and arithmetic right and left shifts. In particular two components are used to perform the shifts: a *Right Shifter* and a *Left Shifter*; they shift the input operand of a given number of positions.
At the output of the ALU, according on the instruction executed, an address to point the *Data Memory* or a data to be stored in the *Register File* is obtained.
There are signals that just pass through the Execute stage, they are:

- inst_type: the name of the current instruction in the Execute stage;
- instruction: current instruction in the Execute stage;
- data_write_MEM: value of the second source register, to be written in the *Data Memory* in the case of a *sw* instruction.

1.2.8 Pipeline Register EX/MEM

The developed processor is characterized by 5 pipeline registers. The third one is the pipeline register between the Execute stage and Memory stage and it it shown in Figure 1.14. In this register the sensitive signals are:

- CLK: clock signal;
- reset: reset signal;

During the rise of the clock, if the reset is low, the input sampling occurs; as a consequence the result of the ALU (Data_or_address_MEM), the current instruction in the Execute stage (instruction_out_EX), the current instruction type in the Execute stage (inst_type_out_EX) and the data to be written in the *Data Memory* in the case of a *sw* instruction (data_write_out_EX) are propagated.

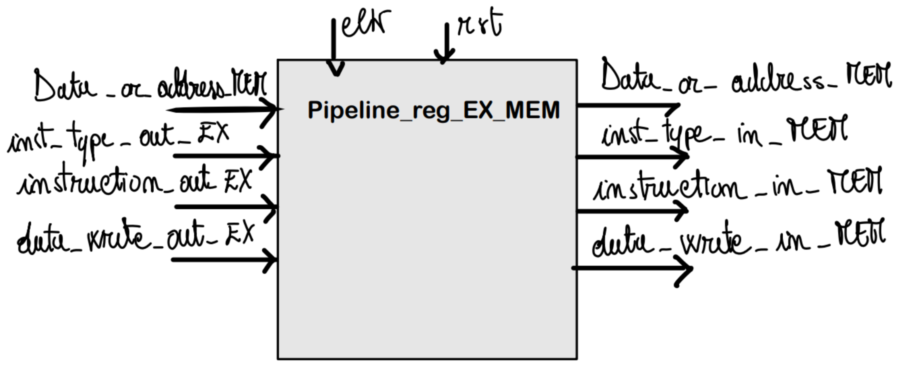

Figure 1.14: pipeline register EX-MEM

1.2.9 Memory stage

The Memory stage unit has the following interface (figure 1.15):

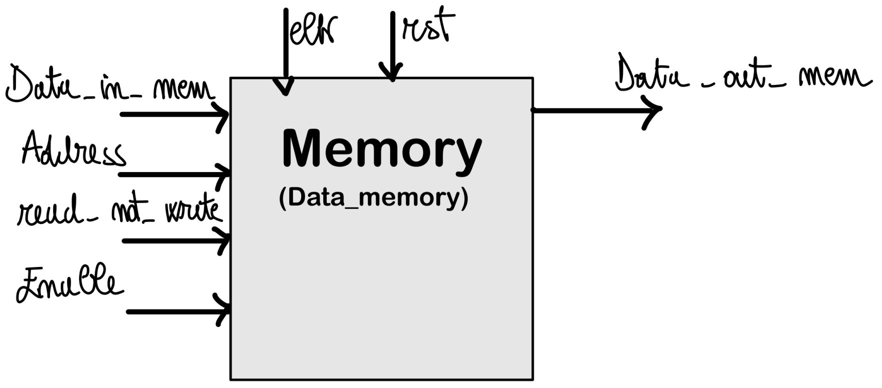

Figure 1.15: Memory stage block

where:

- CLK: clock signal;
- RST: asynchronous active high reset signal;
- Data_in_mem: data to write in memory (during the STORE operation);
- Address: address that points the memory;
- read_noy_write: signal that distinguish if the memory is used in write mode or in read mode;
- Enable: enable for the Data Memory;
- Data_out_mem: data at the out of the memory, after a read operation (LOAD);

In the Memory stage is contained the Data Memory. It is activated if it is needed to perform a *load* or a *store* operations, otherwise the data and the address go directly in the next stage, that is the Write-back.

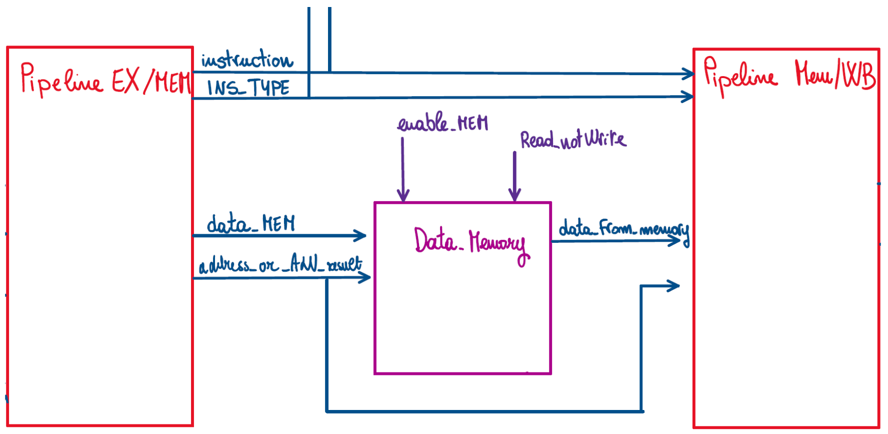

Figure 1.16: immediate generation block

1.2.10 Pipeline Register MEM/WB

The fourth pipeline register is located between the Memory stage and Write-Back stage and it it shown in Figure 1.17. In this register the sensitive signals are:

- CLK: clock signal;
- reset: reset signal;

During the rise of the clock, if the reset is low, the input sampling occurs; the data that comes or from the result of the ALU (ALU_result_out_MEM), the current instruction in the Memory stage (instruction_out_MEM), the data that come from the memory after a reading operation (data_write_out_MEM); and the address that point to the register file for a *lw* operation (address_RF), the result of the ALU that comes from Execute stage and bypass the Data Memory (ALU_result_in_WB) and the data that come from the Data Memory (data_write_in_WB) are propagated to Write-Back stage.

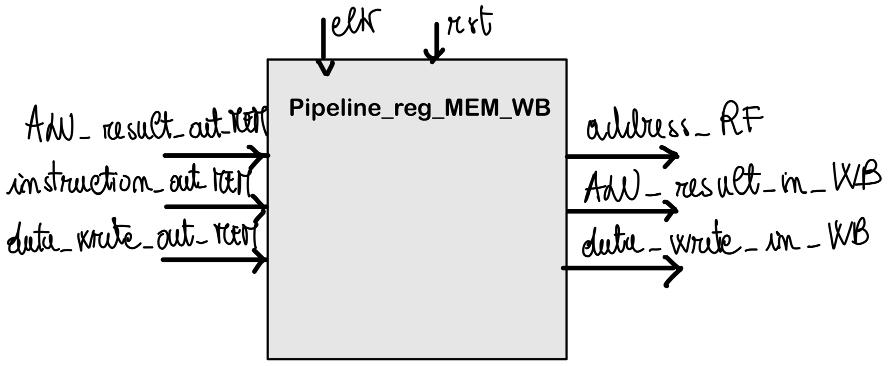

Figure 1.17: pipeline register MEM-WB

1.2.11 Write-back stage

The execute stage is implemented as shown in Figure 1.19. To perform the operations required in this stage, also the *Register File* is required.

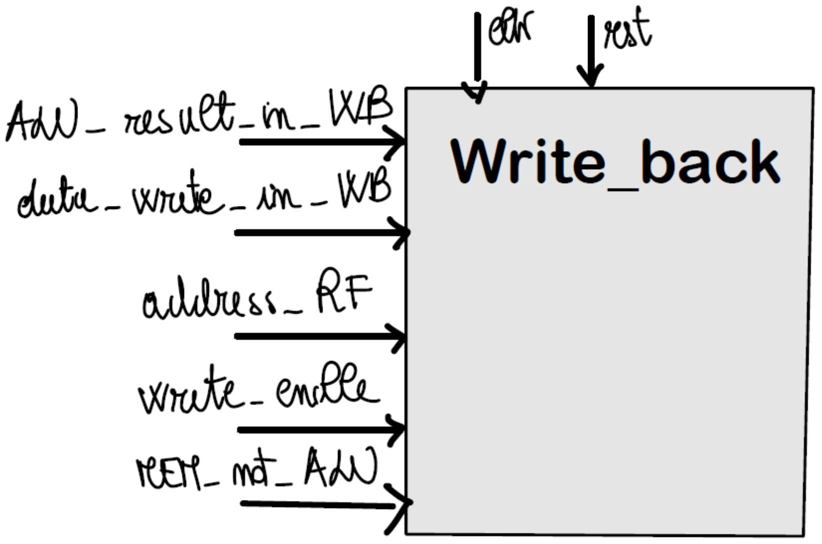

Figure 1.18: Write-back stage block

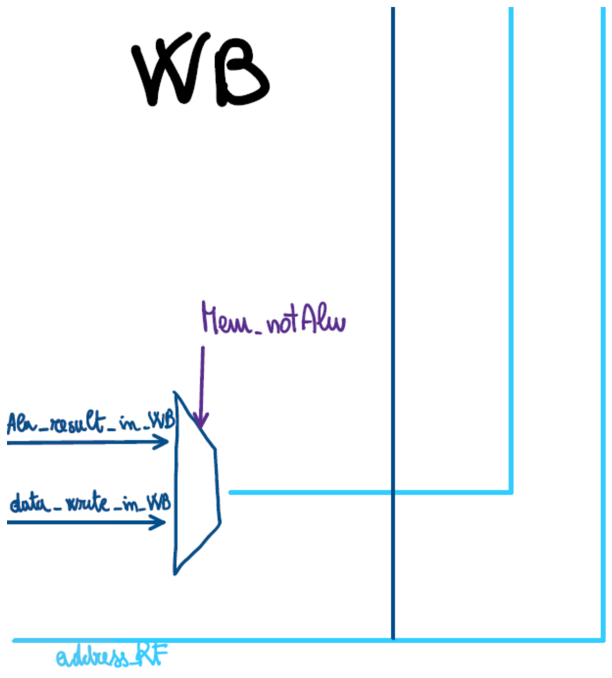

Figure 1.19: Write-back stage

As it can be seen in Figure 1.18 the input and output signals for this block are:

- CLK: clock signal;

- reset: reset signal;
- ALU_result_in_WB: result of the computation of the ALU;
- dat_write_in_WB: value read from the *Data Memory* in the case of a *lw* instruction;
- address_RF: address of the *Register File* where the proper value between the two above has to be stored;
- Mem_notAlu: selection signal of the multiplexer that selects the proper value to be stored in the *Register File*;
- write_enable: control signal from the *Control Unit* for the *Register File* to select a write operation;

In conclusion, the components required to implement the Write-back stage are the *Register File* and a *Multiplexer*.

1.2.12 Control Unit

The Control Unit is a component of the Decode Stage, that generates the proper control word based on the current decoded instruction and ALU_OP that is the control signal (of type 'instruction') for the ALU.

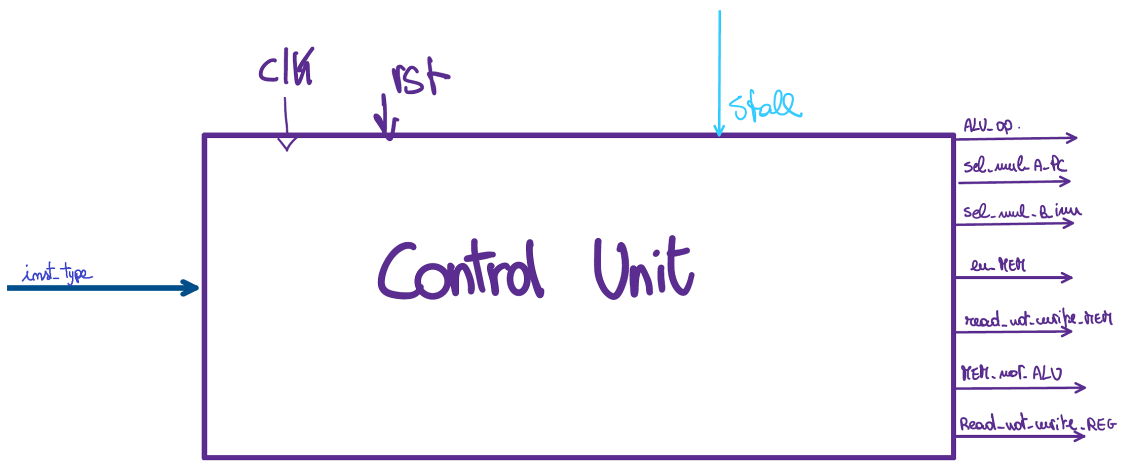

Figure 1.20: Control Unit

The total amount of control signal is six and so the control word is organized in the following way:

- sel_mux_A_PC: control signal of the first multiplexer of Execute stage (if it is 0 selects A operand, if it is 1 selects the value of Program Counter);
- sel_mux_B_imm: control signal of the second multiplexer of Execute stage (if it is 0 selects B operand, if it is 1 selects the value of immediate);
- enable_MEM: enable signal for the Data Memory;
- read_not_write_MEM: if this control bit is 1 the Data Memory is in Read Mode, if it is 0 the Data Memory is in Write mode;
- MEM_not_ALU: control signal of the multiplexer of Write-Back stage (if it is 0 selects data that comes from the ALU operation, if it is 1 selects the data that comes from Memory);
- read_not_write_REG: signal that indicates that the Register File is in write mode (read_not_write_REG = 0) or in read mode (read_not_write_REG = 1).

In order to dispatch the signals to the proper stage, the control word has to be delayed. After the generation of the control word, in the Execute stage are used sel_mux_A_PC and sel_mux_B_imm, while the others bits are propagated. In the Memory stage are used enable_MEM and read_not_write_MEM, and the others two bits are propagated. In the Write-Back stage are used the remaining two control bits MEM_not_ALU and read_not_write_REG.

The ALU control signal is menaged separately and it is delayed only by one clock cycle to be used properly in the Execute Stage.

1.2.13 Hazard Detection Unit

The *Hazard Detection Unit* allows the detection of possible hazards and prevents them inserting a stall.

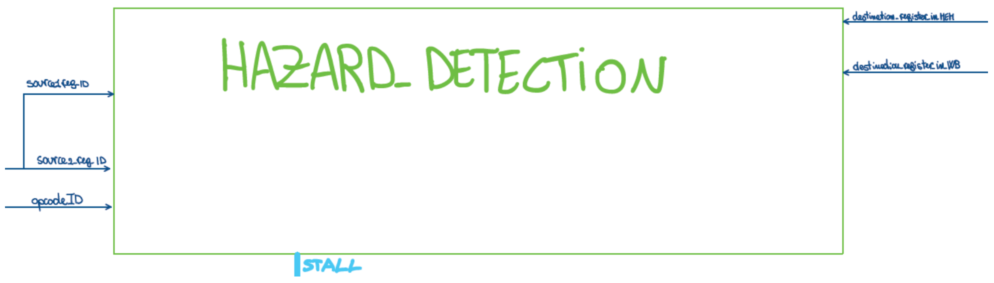

Figure 1.21: Hazard-detection Unit

The situations analyzed by the *Hazard Detection Unit* are the following:

- only the instructions in the pipeline register IF-ID with at least one source register are analyzed, thus AUPC, JAL and LUI are not analyzed;
- only the instructions in the pipeline register EX-MEM with destination register different from 0 are analyzed.

Between the previously indicated situations the stall is asserted if:

- the first source register of the instruction in the pipeline register IF-ID is equal to the destination register of the instruction in the pipeline register ID-EX or to the destination register of the instruction in the pipeline register EX-MEM;
- the instruction in the pipeline register IF-ID has a second source register (so it is not the case of ADDI, SRAI, LW and SLLI) and it is is equal to the destination register of the instruction in the pipeline register ID-EX or to the destination register of the instruction in the pipeline register EX-MEM.

IRAM

The IRAM is one of the 2 module which communicate with RiscV processor. In this case, this module contain the sequence of instructions in h.exadecimal format of our initial program. The input of this module is the program counter, generated by our architecture, and the output is the instruction, which will be executed in our processor in the next stage. The sequence of instruction is the following one:

```
1fc18197
01c18193
7fbff117
ff410113
00010433
008000ef


0000006f



10010537
ff010113
00700593
00050513
00112623
01c000ef
00c12083
100107b7
00a7ae23
00000513
01010113
00008067


00052603
00100713
41f65793
00c7c633
40f60633
02b75863
00259593
00450713
00b50533
00072783
00470713
41f7d693
00f6c7b3
40d787b3
00c7d463
00078613
fee512e3
00060513
00008067
```

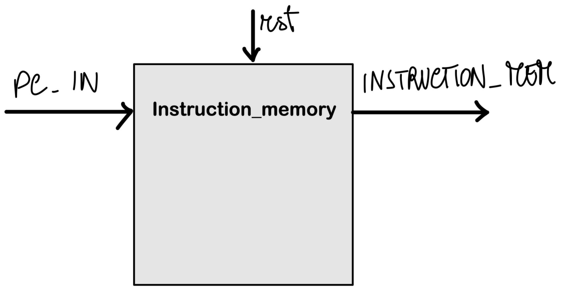

Figure 1.22: IRAM

DRAM

The DRAM is the second module that communicate with RiscV processor. The inputs and output of this module are already explained in the Memory stage. When the reset signal is asserted, the initialization of the memory is performed, with the values contained in the *data.hex* section:

```
main:      file format elf32-littleriscv


Disassembly of section .data:

10010000 <v>:
10010000:      0009              c.addi  zero,2
10010002:      0000              unimp
10010004:      ffd2              fsw     fs4,252(sp)
10010006:      ffff              0xffff
10010008:      0015              c.addi  zero,5
1001000a:      0000              unimp
1001000c:      fffe              fsw     ft11,252(sp)
1001000e:      ffff              0xffff
10010010:      000e              0xe
10010012:      0000              unimp
10010014:      001a              0x1a
10010016:      0000              unimp
10010018:      fffd              bnez    a5,10010016 <v+0x16>
1001001a:      ffff              0xffff
```

Also in this case, the reading phase is performed in asynchronous way, while the writing phase is performed in synchronous way.

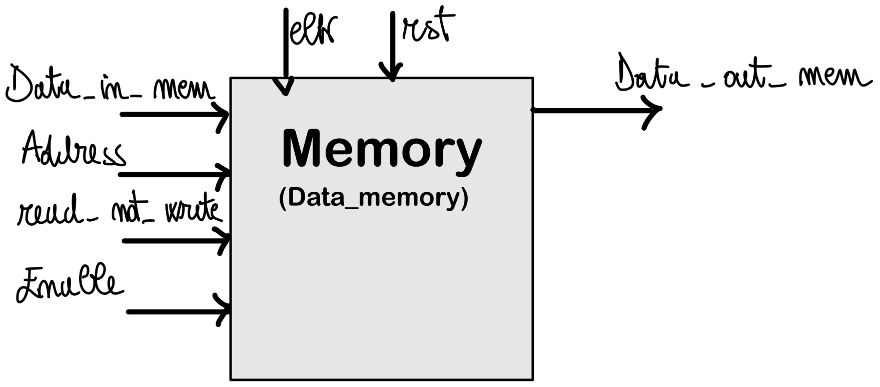

Figure 1.23: Data Memory

1.2.14 Testbench & Simulation

Figure shows a snapshot of the simulation of the designed processor as a standalone block. It shows that the processor works properly: the is related to the initial PC, and the shows the reached final PC.

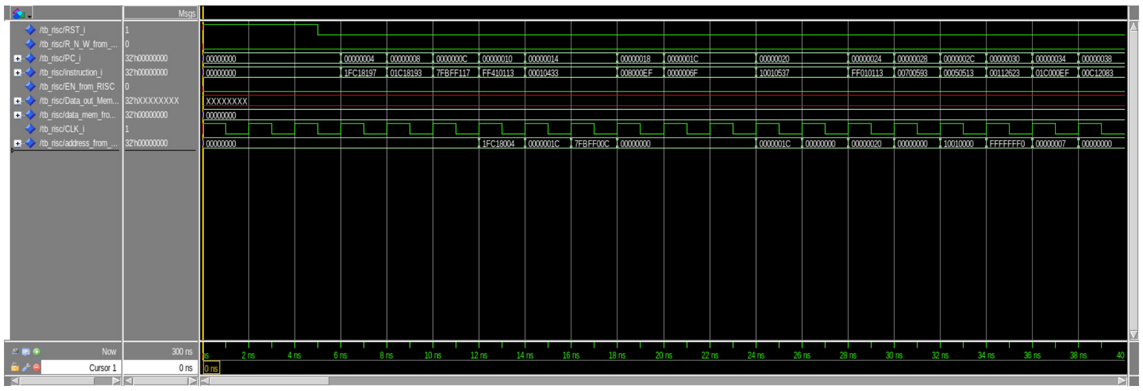

Figure 1.24: initial phase

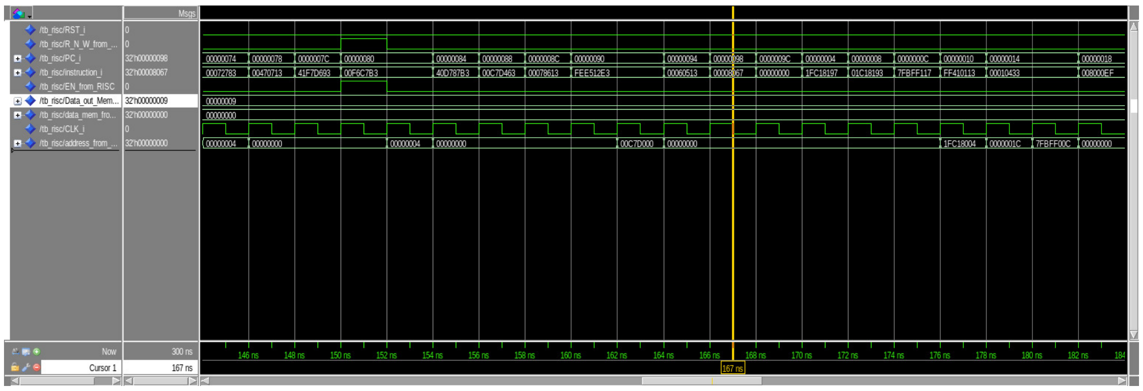

Figure 1.25: final phase