Politecnico di Torino

III Facoltà di Ingegneria

Integrated Systems Architecture Laboratory Reports

Master degree in Electrical Engineering

02GQCOQ

Antonio Lattanzio, Federico Perenno, Gianfranco Sarcià

February 16, 2022

Contents

CHAPTER 3

Design of a RISC-V-lite processor

The goal of this third laboratory experience is to design a RISC-V-lite processor with 5 pipeline stages, but with a limited amount of 32-bit instructions to handle, based on the RV32I instruction set. The instructions are 12 in total and belong to one of these six categories:

- R or *register*:
  - `add`: the content of (the registers addressed by) *rs1* and *rs2* are summed and stored in (the register addressed by) *rd*
  - `xor`: the content of *rs1* and *rs2* are sent to an XOR gate bit by bit and the result is stored in *rd*
  - `slt`: it stores '1' in *rd* if the value of *rs1* is smaller than *rs2*'s, otherwise it stores a '0'
- I or *immediate*:
  - `addi`: the content of *rs1* is added to the immediate value and stored in *rd*
  - `andi`: the content of *rs1* and the immediate value are sent to an AND gate bit by bit and the result is stored in *rd*
  - `lw`: it stores in *rd* a value taken from the data memory at an address retrieved by adding together the immediate value and the content of *rs1*
  - `srai`: only the bottom five bits of the immediate value indicate by how many positions the content of *rs1* needs to be shifted to the right
- S or *store*:
  - `sw`: the contents of *rs2* is stored in the data memory at the address given by the sum of the content of *rs1* and the immediate value
- SB or *conditional branch*:
  - `beq`: if and only if the content of *rs1* and *rs2* are equal, then the next instruction's address will be given by the sum between the program counter and twice the immediate value
- U or *upper immediate*:
  - `lui`: the immediate value has its 12 bottom bits filled with zeros and is then stored in *rd*
  - `auipc`: the sum between the program counter and the immediate value with its 12 bottom bits filled with zeros is stored in *rd*

- UJ (or J) or *unconditional jump*:
  - `jal`: the next instruction's address is given by the sum between the program counter and the immediate value; also, the value of the program counter, incremented by 4, is stored in *rd*

The instructions format is reported in *figure 3.1*.

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

Figure 3.1: *RV32I Instruction Format*

3.1 Design

We designed a processor that can handle all the aforementioned instructions and that is able to detect and manage hazards in the most efficient way possible.
In particular:

- a *forwarding unit* is able to handle Read-After-Write hazards
- a *hazard detection unit* is able to detect the Load-Use hazard, which requires the insertion of a *NOP* and the freezing of the next instruction
- when a conditional jump occurs a *NOP* must be inserted after
- since we decided to compute the branching condition in the *Execute* stage of the pipeline, two *NOP*s have to be inserted.

Notice that we decided to compute the branching condition in the *Execute* stage rather than in the *Instruction Decode* stage and this is due to the fact that, after simulating both versions of the processor exploiting the *Synopsys Design Vision* logic synthesys tool, we found the critical path to be quite shorter in the first case. The significance of this improvement justified the occasional insertion of an additional *Null Operation*.
The timing reports related to these two different implementations are discussed in *section 3.1.2*.

The complete architecture of our RISC-V-lite processor is reported in *fig. 3.2* and the functionalities and implementations of the various components in each pipeline stage are described shortly after.

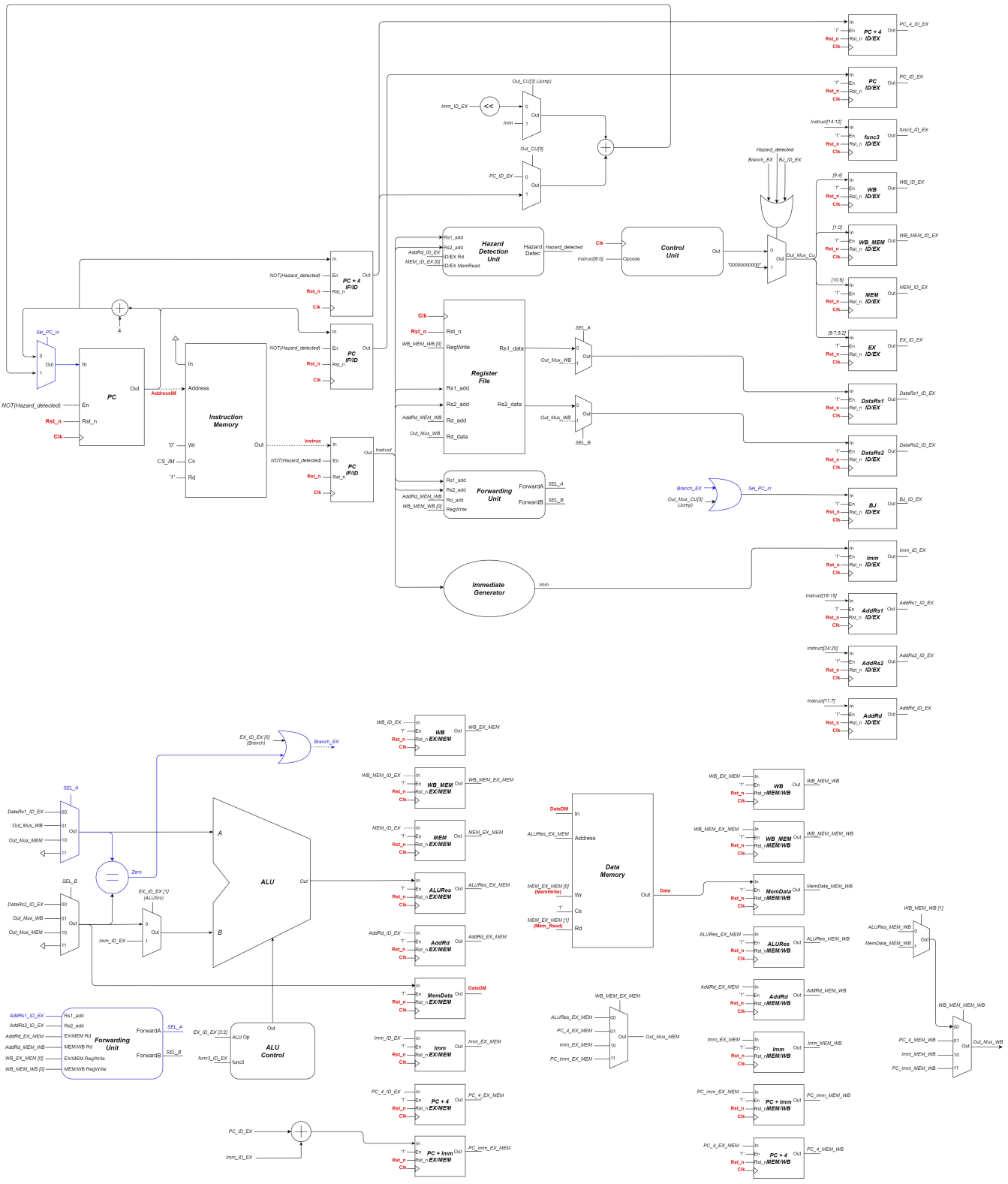

Figure 3.2: *RISC-V-Lite Full Architecture*

Instruction Fetch

It is the first stage of the pipeline and, as the name suggests, it is devoted to the fetching of a new instruction. Instructions are stored in an instruction memory, which in our case is external to the processor and defined as an asynchronous memory in the testbench, and is addressed by a value which is stored in a special 32-bit register: the *Program Counter*.
The value contained in the program counter (PC) is usually incremented by 4 at each clock cycle, unless an unconditional jump or a branch occurs, which can radically change the value stored inside the PC. The input of this special register is selected by a multiplexer.
This register is always enabled, unless the reset is active (which means that the processor hasn't been started yet) or if a *Load-Use* hazard is detected. In fact, as previously stated, this hazard requires the insertion of a *NOP* and the content related to the following instruction must be frozen, and the program counter, as a consequence, can't be updated and must be disabled. Hence the reason why the enable signal is nothing but the inverted output of the hazard detection unit.

Instruction Decode

In this second stage, all the operations regarding the decoding of the instructions are handled. The main units are the following:

- *Register File*:

it contains thirty-two 32-bit registers. One register (at address *rd*) can be written while two registers (at addresses *rs1* and *rs2*) are being read. However, we implemented the writing operation to be synchronous and the reading operation to by asynchronous. This means that when the content of a register is updated it won't be available until the next clock cycle.
If the content being written on the register is needed to be read a *bypass* mechanism allows to take this value directly. This operation is performed by a forwarding unit.

- *Forwarding Unit*:

this small forwarding unit has the only purpose to bypass the synchronous writing of the register and this is done only when the content being stored at *rd* needs to be read (hence *rd* equals either *rs1* or *rs2* and the register file is being written).
Two multiplexers driven by the results of the forwarding unit choose whether to take the value from the register or to bypass it.

- *Hazard Detection Unit*:

it checks the conditions for which a *Load-Use* hazard occurs. When it does, a *Null Operation* must be inserted (which is done by setting to '0' all control signals) and anything related to the next instruction must be frozen so that it can be decoded after the NOP.

- *Immediate Generator*:

the immediate value is needed by all instruction types with the exception of the *R* type. Each one of these five possible instruction types has a different format for the immediate value and it's the immediate generator's purpose to send, based on the opcode values, the correct one as an output.
This means all possible values are computed starting from the 32-bit instruction and then sent to a

multiplexer. Then the right output is selected based on the operation code.

- *Control Unit*:

our processor needed a total of nine different control signals (11 bits overall):

- `MemRead`: enables the reading from the *Data Memory*
- `MemToReg`: makes it so that the value to be written on the register file is taken from the data memory and not from the *ALU*
- `ALUOp[1:0]`: sent to the *ALU Control* and, together with *func3* (taken directly from the instruction), it determines the operation the *ALU* has to perform
- `MemWrite`: enables the writing on the *Data Memory*
- `ALUSrc`: determines whether the second input of the *ALU* is taken from the register file or from the immediate generator
- `RegWrite`: enables the writing on the *Register File*
- `Jump`: flags the presence of an unconditional jump
- `Branch`: flags the presence of a conditional branch
- `Mux_WB[1:0]`: selects which value needs to be stored in the register file between the *ALU result/ Data Memory output*, $PC + 4$, the *immediate value* and $PC + immediate$

Despite the fact that 12 instructions overall need to be handled, there are only 8 different opcodes and each operation code has its corresponding control signals, which are reported in *table 3.2*. The control unit is made of two main blocks, the Read Only Memory (ROM) containing the values related to each opcode, and the 7-to-4 decoder taking the opcode and addressing the ROM as shown in *tab.3.1*. The memory is made of 9 rows (one for each opcode plus one representing the idle state) and 11 columns.

Instruction	**Opcode**	**Address**
`ADD, SLT, XOR`	0110011	000
`SRAI, ADDI, ANDI`	0010011	001
`SW`	0100011	010
`LW`	0000011	011
`BEQ`	1100011	100
`JAL`	1101111	101
`AUIPC`	0010111	110
`LUI`	0110111	111

Table 3.1: *7-to-4 Decoder addressing the ROM in our Control Unit*

Add	**MemRead**	**MemToReg**	**ALUOp[1]**	**ALUOp[0]**	**MemWrite**	**ALUSrc**	**RegWrite**	**Jump**	**Branch**	**Mux_WB[1]**	**Mux_WB[0]**
000	0	0	0	0	0	0	1	0	0	0	0
001	0	0	0	1	0	1	1	0	0	0	0
010	0	0	1	0	1	1	0	0	0	0	0
011	1	1	1	1	0	1	1	0	0	0	0
100	0	0	1	1	0	0	0	0	1	0	0
101	0	0	1	1	0	0	1	1	0	0	1
110	0	0	1	1	0	0	1	0	0	1	1
111	0	0	1	1	0	0	1	0	0	1	0
stage	**MEM[1]**	**WB[1]**	**EX[3]**	**EX[2]**	**MEM[0]**	**EX[1]**	**WB[1]**	**ID**	**EX[0]**	**MEM/WB[1]**	**MEM/WB[0]**

Table 3.2: *Control Unit of our RISC-V-lite processor*

- *Other components*:

additionally, at the decode stage other operations are performed; for instance a NOP must be inserted (by setting all control signals to zero) when one of these conditions is satisfied:

- the Hazard Detection Unit finds a *Load-Use* hazard;
- at the previous step an unconditional jump instruction is decoded;
- one or two clock cycles before a satisfied branching condition has been decoded.

If there is either an unconditional jump or a satisfied branching condition the content of the program counter must be update with either $PC+immediate$ (in the case of a jump) or with $PC+2 \cdot immediate$ in case of a branch. To do so, an OR gate is needed along with 2 multiplexers and a left-shifting operation, as shown in the datapath.

Execute

At this stage, the vast majority of arithmetical operations are performed. The branching condition is checked with a comparator which compares the values of *rs1* and *rs2* and if they're equal a `Zero` flag is set to '1'. If this flag is up and a Branching Condition has been decoded at the previous stage, it means that a branch is taken. An *Arithmetic Logic Unit* (ALU) does the rest.
The main functional units are the following:

- *ALU Control*:

it chooses which operation the ALU needs to perform, based on the `ALUOp` control signal and on the *func3* bits. The decoding done by this particular unit is summarized in *tab.3.3*.

Instruction	ALUOp	func3	ALU operation	ALU_c
`ADD`	00	000	addition	000
`SLT`	00	010	comparison	100
`XOR`	00	100	logical XOR	010
`SRAI`	01	101	right-shift	011
`ADDI`	01	000	addition	000
`ANDI`	01	111	logical AND	001
`SW`	10	010	addition	000
`LW`	11	010	addition	000

Table 3.3: *7-to-4 Decoder addressing the ROM in our Control Unit*

- *Forwarding Unit*:

it has the important role of handling *Read-After-Write* hazards. It does so by checking if the content of the register at the *rd* address that has been set to be updated in one of two previous clock cycles has to be used in the current *Execute* stage. If so, the *Write Back* to the register file is bypassed and the content to be stored at *rd* is sent to a multiplexer, properly addressed by the forwarding unit.

- *ALU*:

this specific Arithmetic Logic Unit performs five different operations on the two received inputs and sends out only one result based on the signal coming from the *ALU Control* block. The operations performed are the following:

- `Addition`: standard signed addition between the two inputs.
- `Logical AND`: each bit from the first input is sent to an AND gate along with the corresponding bit coming from the second input.
- `Logical XOR`: each bit from the first input is sent to an XOR gate along with the corresponding bit coming from the second input.
- `Right Shift`: the 5 least significant bits of the second input correspond to the amount of bits the first input needs to be shifted to the right. A barrel-shifter is able to perform this operation.
- `Comparator`: it performs the subtraction between the first and the second input and if the result is negative it means that the former is smaller than the latter. If $A < B$ the comparator returns the value '1' on 32 bits, otherwise it returns all zeros.

- *Other components*:

An additional multiplexer, driven by the `ALUSrc` control signal selects whether the second input of the ALU corresponds to *rs2* or to the immediate value.

Memory

At this stage, all Data Memory related operations are performed. More specifically, the value contained at address *rs2* corresponds to the input data of the memory, while its output is sent to the corresponding pipe register. Meanwhile, two control signals, `MemWrite` and `MemRead`, indicate the operation to perform.

Apart from memory related operations, the value to be sent to the Execute stage, which can be recovered by the Forwarding Unit in case a Read-After-Write hazard occurs, must be properly set. In fact, the value to be written in the register file at address *rd* can be either coming from the ALU, but in case of a `JAL`, `LUI` or `AUIPC` instruction, $PC + 4$, *the Immediate value* or $PC + immediate$ must be stored respectively.
A multiplexer, controlled by the appropriate control signal `Mux_WB`, serves this purpose.

Write Back

In this final stage, the data needed to overwrite the register file are finally available. In particular, we send the `RegWrite` control signal, the address *rd* and the input data, which is nothing but the output of two multiplexers.
The first one, controlled by the `MemToReg` control signal, chooses whether the value to be stored has to be taken from the ALU or from the Data Memory. The output of this first *mux* is the first input of the second multiplexer which is otherwise identical to the one adopted in the *Memory* stage.
The output of this second multiplexer is sent to the Register File as value to be written at address *rd*, but it is also sent to the multiplexers in the *Execute* stage since it may be needed if a Read-After-Write hazard occurred.

3.1.1 Simulation

We exploited the *Modelsim* simulation tool to check the accuracy of our design. The test run on our processor concerns the computation of the minimum absolute value among seven integers stored in a vector. The description of the steps our processor needs to follow is thoroughly described in *sec.3.3*.

By looking at how the interested registers are updated overtime and by verifying that the final result is correctly stored at the corresponding address in the data memory once the loop has been completed, we are confident to say that our processor can perfectly handle all the tasks it has been designed to perform.
Three critical simulation snippets are reported in the following figures.

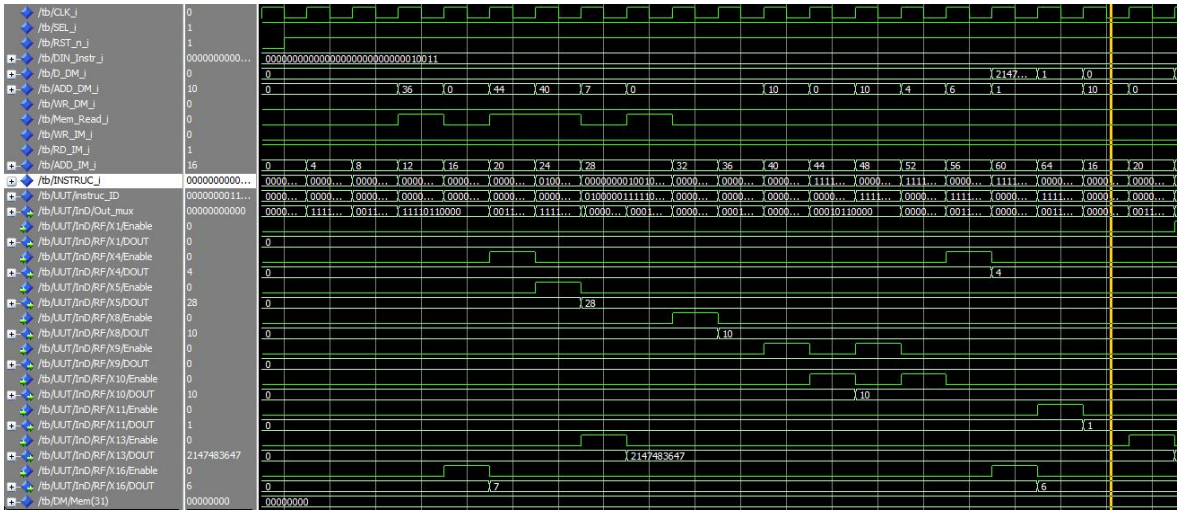

Figure 3.3: *Processor's start with complete first loop*

In *fig.3.3* the processor starts functioning as soon as the *Reset* signal is no longer active. In the first clock cycles the registers are initialized, then the first loop begins. We can see that a NOP is inserted and the successive instruction is frozen when a *Load-Use* hazard is detected. Read-After-Write hazards instead, are handled without the addition of any null operation, thanks to the forwarding units. Since the value under test is a new minimum absolute value, the loop continues until there is an unconditional jump, which requires the insertion of one NOP right after, back to the beginning of the loop.

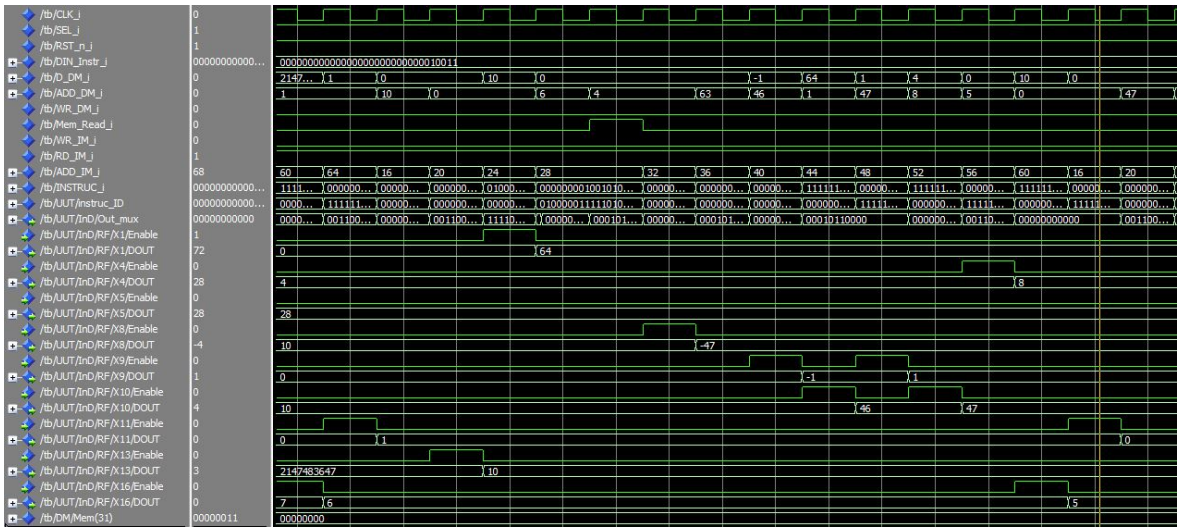

Figure 3.4: *Second loop*

In *fig.3.4* the handling of the second iteration of the loop is reported. In this case, the value under test is not a minimum absolute value and for this reason we jump back to the beginning of the loop before we can reach the unconditional jump. This is due to the fact that a branch is taken, which causes the insertion of two NOPs after the decoding of the `BEQ` instruction.

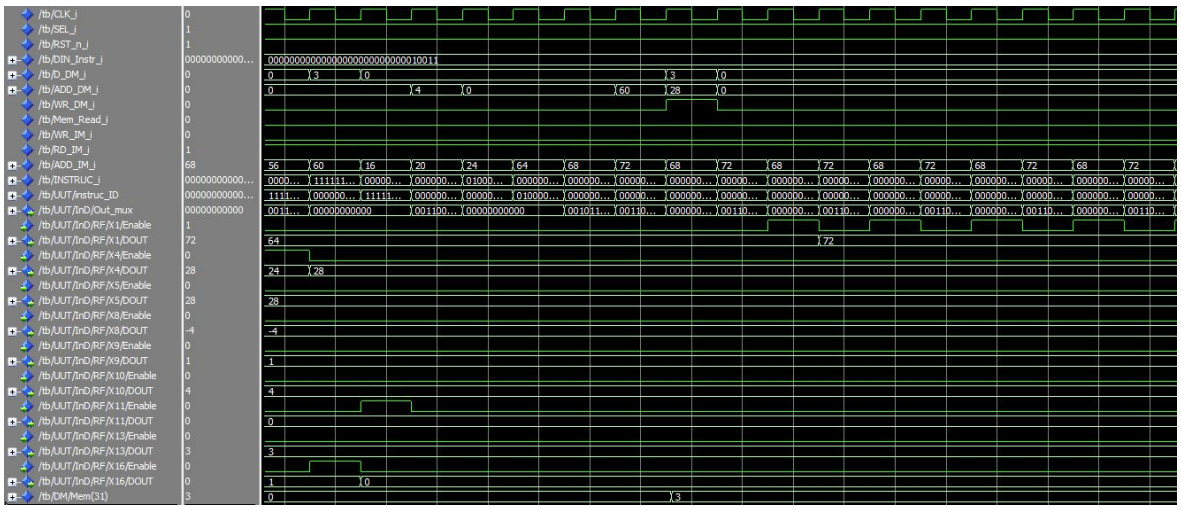

Figure 3.5: *End of the test*

In *fig.3.5* the end of the simulation is reported. The minimum absolute value (previously stored in x13) is correctly stored in the Data Memory at the right location. Finally, an endless loop performing no useful operations is present.

3.1.2 Synthesis

As already mentioned in the introduction, our initial idea was to verify the branching condition at the Instruction Decode stage, hence reducing to one the number of *Null Operations* to be inserted. However, upon further scrutiny, we found the critical path given by the logic synthesis to be quite lengthy. In fact, since Read-After-Write hazards could affect the values to be compared, a forwarding unit had to be implemented to avoid them. The results of this forwarding unit controlled a multiplexer where one of the inputs came directly from the ALU in the *Execute* stage. Hence, the critical path was given by the forwarding unit in the *Execute* stage, by the multiplexer related to it, then by the ALU computation and finally by the multiplexer and the comparator in *ID*. This result would then be used to select the input value of the Program Counter, further increasing its length.

On the other hand, by verifying the branching condition at the *Execute* stage, the critical path doesn't go through the ALU any longer, but it is only affected by the comparator delay. The critical path is highlighted with blue lines in the datapath shown in *fig.3.2*.

In *tab.3.4* the indicative values related to area and delay are reported for the two implementations.

Stage of Branching Condition Verification	**Delay** $[ns]$	**Area** $[\mu m^2]$	**NOPs if branch taken**
Instruction Decode	1.68	17,854	1
Execute	1.23	17,605	2

Table 3.4: *Comparison between two possible RISC-V-Lite implementations*

The improvement we get by postponing the branching condition verification outshines the negatives. As a matter of fact, the additional NOP is inserted only when a branch is taken, which doesn't happen often enough to justify the 36% delay increase that comes with it.

The simulation of the netlist generated by *Synopsys Design Vision* granted the same simulation results as the simulation of the source files.
The starting and final portions of the simulation are reported in *fig.3.6 and 3.7*.

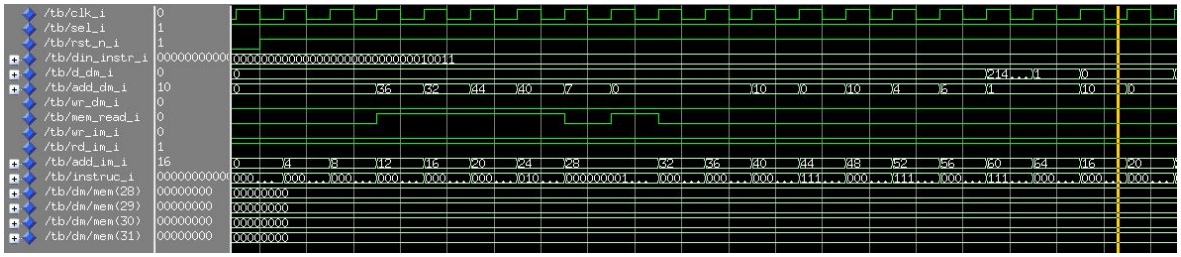

Figure 3.6: *Start of the simulation of the netlist generated by Synopsys*

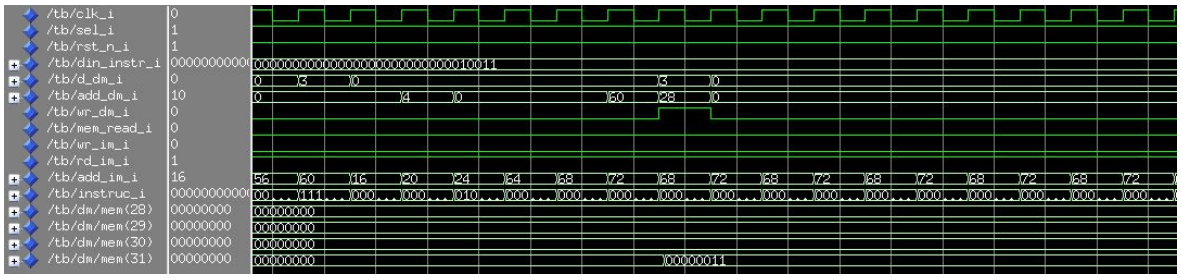

Figure 3.7: *End of the simulation of the netlist generated by Synopsys*

3.1.3 Place and Route

We performed the place and route of our architecture and generated the corresponding netlist. The simulation of the latter granted the same results as the previously obtained ones and when we veryfied connectivity and geometry, no warnings nor errors were detected.
The starting and final snippets of the simulation are shown in *fig.3.8 and 3.9*.

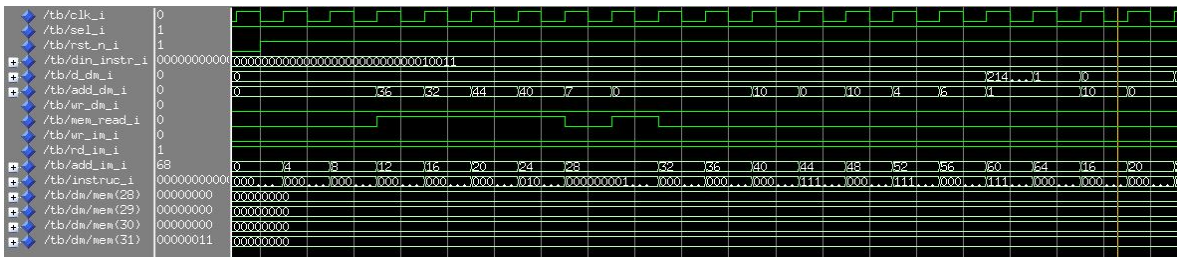

Figure 3.8: *Start of the simulation of the netlist generated by Cadence Innovus*

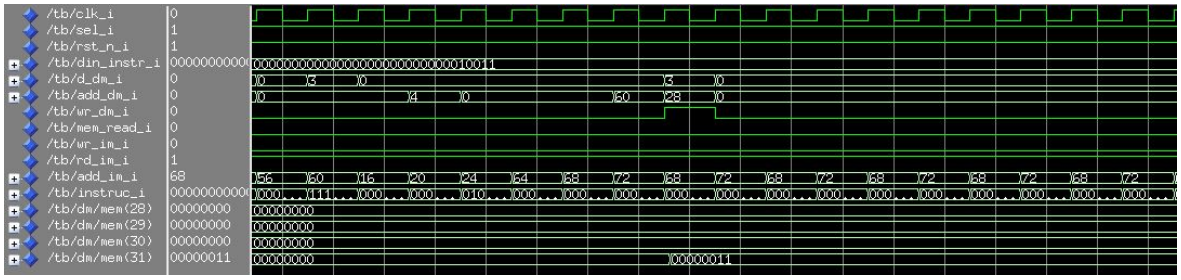

Figure 3.9: *End of the simulation of the netlist generated by Cadence Innovus*

3.2 Special function unit (absolute value)

In the second part of this laboratory experience, we modified our RISC-V-Lite processor by modifying the ALU so that it could handle the computation of the absolute value, given two inputs. With this goal in mind, we defined an `ABSI` instruction of Immediate type, performing the addition between the content of register *rs1* and the immediate value and by then retrieving the absolute value. Its opcode is the same as other instructions of the same type, but we set the *func3* bits to be different from any other Immediate instruction so the the ALU Control block could be slightly modified to generate the appropriate signal to select the output of the ALU.

The introduction of this new special instruction allowed to substitute four instructions from the original loop, thus apparently speeding up the execution of this specific test program.

3.2.1 Simulation

We exploited the `Modelsim` simulation tool, to analyze the same execution portions we analyzed for the standard processor.

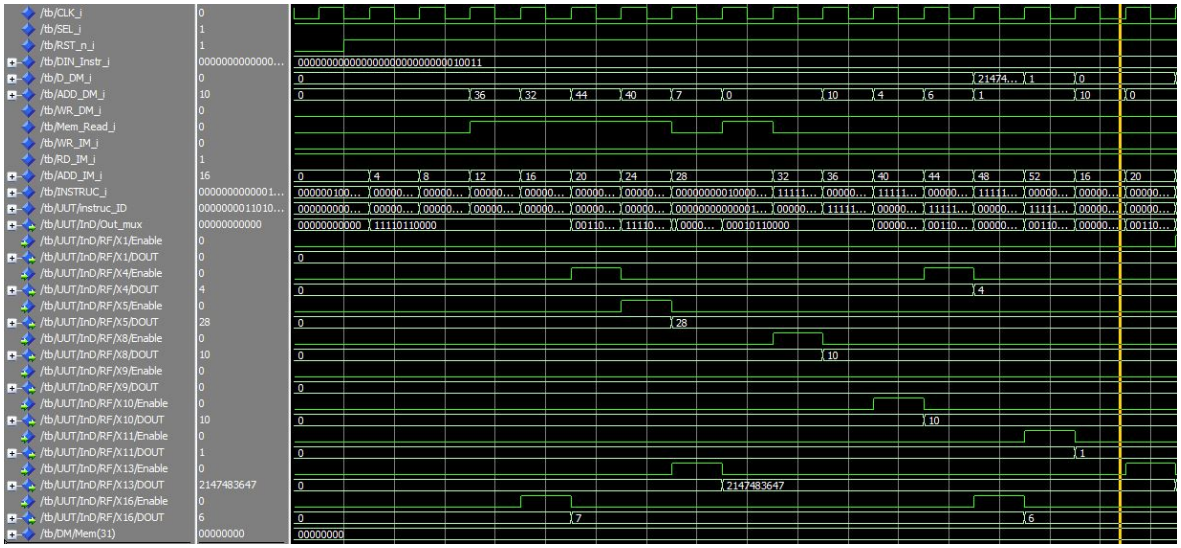

Figure 3.10: *Processor's start with complete first loop*

As shown in *fig.3.10* the processor performs the same steps before the loop starts. Afterwards, however it takes three less clock cycles, no matter what, to complete the first iteration of the loop.

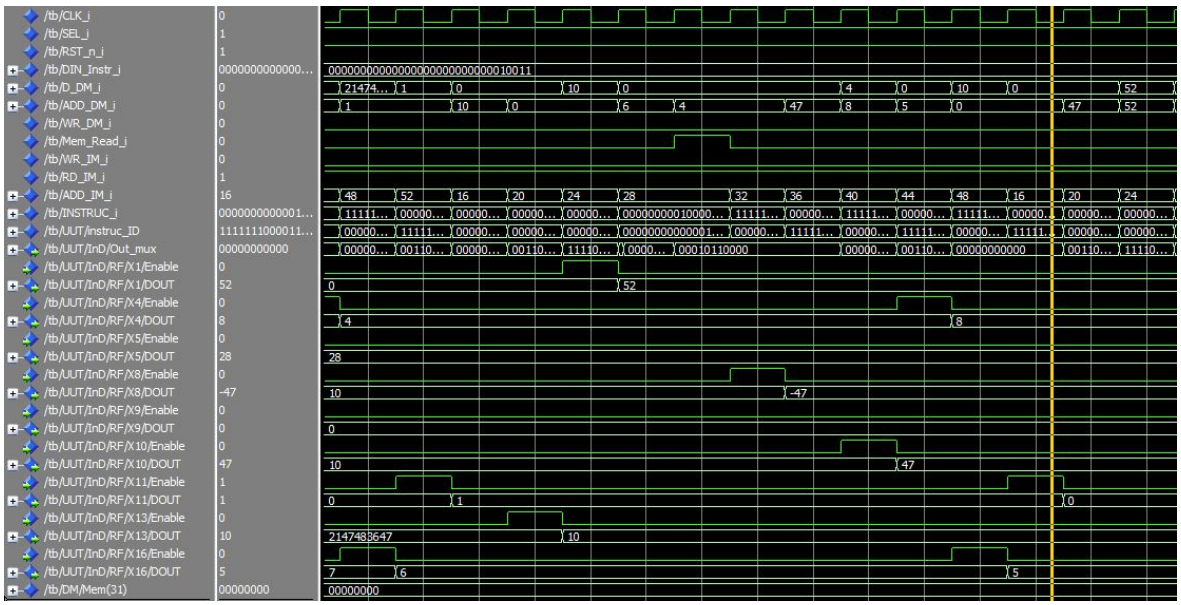

Figure 3.11: *Second loop*

In *fig.3.11* we observe the same phenomenon, since the loop only takes one clock cycle to compute the absolute value, instead of four.

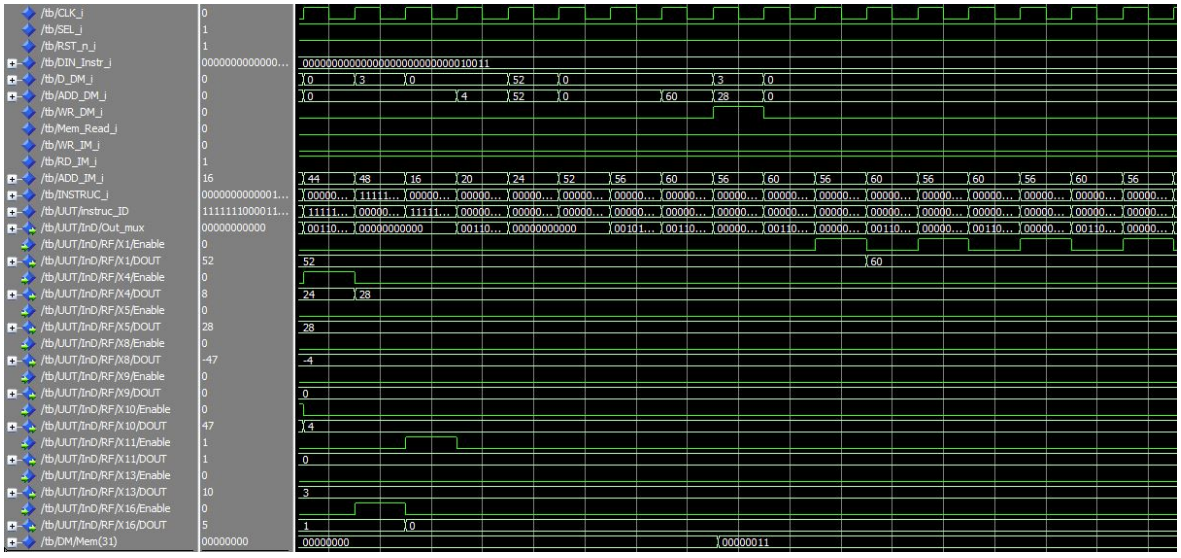

Figure 3.12: *End of the test*

In *fig.3.12* the end of the simulation is reported, and the correct value is stored in the data memory, which proves the correct functioning of our processor.

3.2.2 Synthesis

Thanks to the `Synopsys` logic synthesis tool we were able to identify the critical path which highlighted how the addition of a special function unit comes indeed with its costs. As a matter of fact, the critical path isn't given by the verification of the branching condition any longer, but the computation of the absolute value becomes critical.

RISC-V-Lite Implamantation	**Delay** $[ns]$	**Area** $[\mu m^2]$
Standard	1.37	17,849
Additional `ABS` special function unit	1.23	17,605

Table 3.5: *Comparison between the standard RISC-V-Lite implementation and the one with the additional special function unit*

As reported in *tab.3.5*, the addition of the special function unit increases the delay on the critical path and, to a lower degree, the area.

This delay increase has the effect to lower the maximum achievable frequency, so it turns out that this new processor implementation worsens the performance in most cases, and it might not be as viable as we initially thought.

The simulation of the netlist returns the same results as the simulation performed on the source files, as reported in *fig.3.13 and 3.14*.

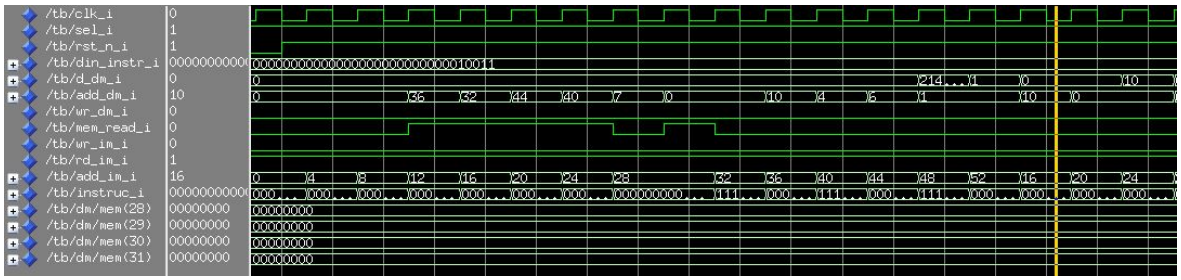

Figure 3.13: *Start of the simulation of the netlist generated by Synopsys*

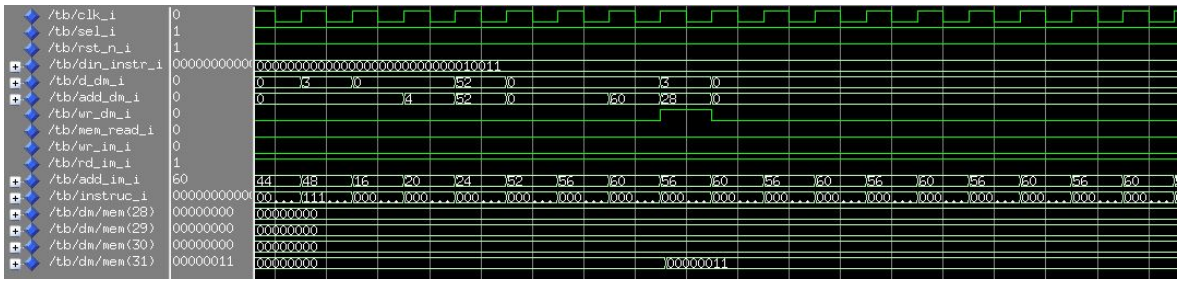

Figure 3.14: *End of the simulation of the netlist generated by Synopsys*

3.2.3 Place and Route

As for the standard implementation of our processor, we performed the place and route of our architecture and generated the corresponding netlist. The simulation of the latter granted the same results as before and when we verified connectivity and geometry, no warnings nor errors were detected.
Fig.3.15 and fig.3.16 show the starting and final snippets of the simulation.

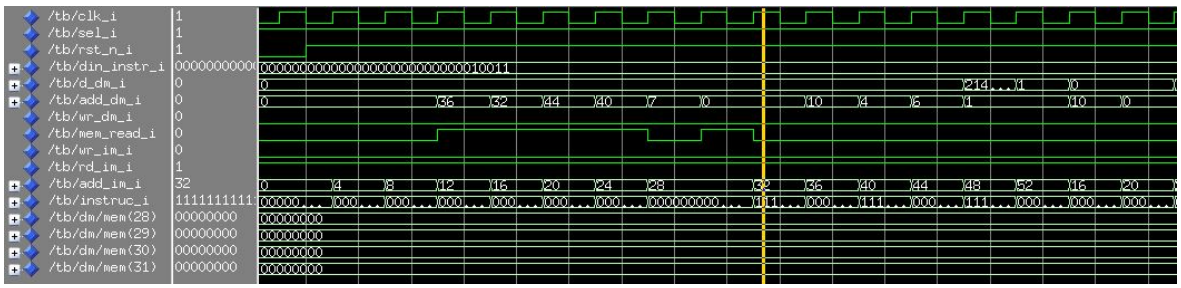

Figure 3.15: *Start of the simulation of the netlist generated by Cadence Innovus*

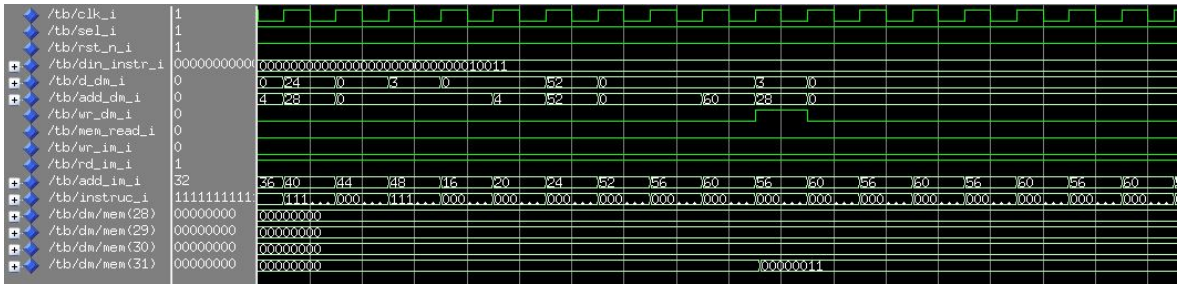

Figure 3.16: *End of the simulation of the netlist generated by Cadence Innovus*

3.3 From assembly to machine code

Before writing in the memories, there is the problem of understanding which steps the algorithm performs.The assembly code related to the test program had to be slightly modified, since we could use only a reduced set of instructions.
We obtained the following assembly code:

```
start:
0       li x16,7                        ->      LW x16,36(x0)
4       la x4,v                         ->      LUI x4,0
8       la x5,m                         ->      LW x5,44(x0)
12      li x13,0x3fffffff               ->      LW x13,40(x0)

loop:
16      beq x16,x0,done                 ->      BEQ x16,x0,12
20      lw x8,0(x4)                     ->      LW x8,0(x4)
24      srai x9,x8,31                   ->      SRAI x9,x8,31
28      xor x10,x8,x9                   ->      XOR x10,x8,x9
32      andi x9,x9,0x1                  ->      ANDI x9,x9,0x1
36      add x10,x10,x9                  ->      ADD x10,x10,x9
40      addi x4,x4,0x4                  ->      ADDI x4,x4,0x4
44      addi x16,x16,-1                 ->      ADDI x16,x16,-1
48      slt x11,x10,x13                 ->      SLT x11,x10,x13
52      beq x11,x0,loop                 ->      BEQ x11,x0,-9
56      add x13,x10,x0                  ->      ADD x13,x10,x0
60      jal loop                        ->      JAL x1,-22

done:
64      sw x13,0(x5)                    ->      SW x13,0(x5)

endc:
68      jal endc                        ->      JAL x1,0
72      addi x0,x0,0                    ->      ADDI x0,x0,0

NOTE: we store the PC+4 jal value in x1, since it isn't specified
```

The procedural steps to be performed are reported here:

- `START`:
  - load 7 in x16 (it's the index to be decreased) (stored at address 36 of Data Memory);
  - load the starting address of v[i] in x4 (loaded exploiting `LUI`);
  - load the address of m in x5 (stored at address 44 of Data Memory);
  - load the maximum positive 32-bit signed number in x13 (stored at address 40 of Data Memory);

NOTE : The `LUI` or `AUIPC` instructions can be used instead of `LW` only for storing address '0' (address of v[i]), because it's the only value whose 12 bottom bits are all zeros.

- `LOOP` to find the minimum absolute value:
  - check if all elements have been tested, if so jump to `DONE`;
  - load new element in x8 (load it from data memory, reading address from x4);
  - apply 31-bit shift to get sign mask in x9 (every bit in x9 corresponds to the sign bit);
  - x10 = sign(x8) XOR x8 $\longrightarrow$ first step to get 2's complement in case the number is negative;
  - x9 = x9 AND 0x1 (carry in) $\longrightarrow$ AND between "11...11"(or "00...00") AND "00...01" so that in x9 we have "00...01" only when needed;
  - x10 += x9 (add the carry in) $\longrightarrow$ in x10 we store the 2's complement;
  - point to next element $\longrightarrow$ we add 4 to x4 to address v[i+1];
  - decrease x16 by 1 $\longrightarrow$ reduce the index;
  - x11 = 1 or 0 depending on (x10 < x13)$\longrightarrow$ if the current absolute is less than the absolute minimum we store 1 in x11, 0 otherwise;
  - go back to the start of the loop if there isn't a new min (x11 = 0);
  - otherwise update x13 with the new minimum absolute value;
  - go back to the beginning of the loop no matter what and save the $PC+4$ value in x1, since in the original assembly code no register is specified.

- Once the loop is `DONE`:
  - store the result in the memory $\longrightarrow$ exploit `SW` to store the value of x13 in the data memory at the 'm' address contained in x5;

- `INFINITE Loop`:
  - jump to itself;
  - NOP.

Each assembly instruction can be converted into a 32-bit word, according to the RV32I format. These 32-bit instructions are saved in an `Instruction.txt` file which can be read in the testbench to initialize the memory.

```
Instruction Memory:
        --> 1 byte rows: 1 instruction --> 4 rows --> program starts from address 0

0:      0000000         --> "00000010""01000000""00101000""00000011"
4:      0000100         --> "00000000""00000000""00000010""00110111"
8:      0001000         --> "00000010""00000000""00100010""10000011"
12:     0001100         --> "00000010""10000000""00100110""10000011"
16:     0010000         --> "00000000""00001000""00001100""01100011"
20:     0010100         --> "00000000""00000010""00100100""00000011"
24:     0011000         --> "01000001""11110100""01010100""10010011"
28:     0011100         --> "00000000""10010100""01000101""00110011"
32:     0100000         --> "00000000""00010100""11110100""10010011"
36:     0100100         --> "00000000""10010101""00000101""00110011"
40:     0101000         --> "00000000""01000010""00000010""00010011"
44:     0101100         --> "11111111""11111000""00001000""00010011"
48:     0110000         --> "00000000""11010101""00100101""10110011"
52:     0110100         --> "11111110""10110000""00000111""11100011"
56:     0111000         --> "00000000""00000101""00000110""10110011"
60:     0111100         --> "11111101""01011111""11110000""11101111"
64:     1000000         --> "00000000""11010010""10100000""00100011"
68:     1000100         --> "00000000""00000000""00000000""11101111"
72:     1001000         --> "00000000""00000000""00000000""00010011"
```

Figure 3.17: *Data to be written in the Instruction Memory before starting the program execution*

The same thing can be done for the values that must be inserted in the Data Memory. These values are converted into 32-bit words and reported in a *Data.txt* file.

```
Data Memory:
        --> 1 byte rows:
                v[] values (10,-47,22,-3,15,27,-4)  ->  (0,4,8,12,16,20,24)
                value m                 ->      (32)
                constant 7              ->      (36)
                max signed constant     ->      (40)

0:      000000          --> "00000000""00000000""00000000""00001010"    ->      v[0] = 10
4:      000100          --> "11111111""11111111""11111111""11010001"    ->      v[1] = -47
8:      001000          --> "00000000""00000000""00000000""00010110"    ->      v[2] = 22
12:     001100          --> "11111111""11111111""11111111""11111101"    ->      v[3] = -3
16:     010000          --> "00000000""00000000""00000000""00001111"    ->      v[4] = 15
20:     010100          --> "00000000""00000000""00000000""00011011"    ->      v[5] = 27
24:     011000          --> "11111111""11111111""11111111""11111100"    ->      v[6] = -4
28:     011100          --> "00000000""00000000""00000000""00000000"    ->      m = 0
32:     100000          --> "00000000""00000000""00000000""00000000"    ->      const = 0
36:     100100          --> "00000000""00000000""00000000""00000111"    ->      const = 7
40:     101000          --> "01111111""11111111""11111111""11111111"    ->      const = 2^(31) - 1
44:     101100          --> "00000000""00000000""00000000""00011100"    ->      const = 28
```

Figure 3.18: *Data to be written in the Data Memory before starting the program execution*

Let us now consider the case in which the processor is modified to insert the new absolute value instruction. This modification will lead to the erasing of 4 other instructions which were previously used to compute the absolute value.

```
__start:
0       li x16,7                            ->      LW x16,36(x0)    :      "000000100100-00000-010-10000-0000011"
4       la x4,v                             ->      LW x4,32(x0)     :      "000000100000-00000-010-00100-0000011"
8       la x5,m                             ->      LW x5,44(x0)     :      "000000101100-00000-010-00101-0000011"
12      li x13,0x3fffffff                   ->      LW x13,40(x0)    :      "000000101000-00000-010-01101-0000011"

loop:
16      beq x16,x0,done                     ->      BEQ x16,x0,9     :      "0000000-00000-10000-000-10010-1100011"
20      lw x8,0(x4)                         ->      LW x8,0(x4)      :      "000000000000-00100-010-01000-0000011"
24      absi x10,x8,0(x0)                                                   "000000000000-01000-110-01010-0010011"
28      addi x4,x4,0x4                      ->      ADDI x4,x4,0x4   :      "000000000100-00100-000-00100-0010011"
32      addi x16,x16,-1                     ->      ADDI x16,x16,-1  :      "111111111111-10000-000-10000-0010011"
36      slt x11,x10,x13                     ->      SLT x11,x10,x13  :      "0000000-01101-01010-010-01011-0110011"
40      beq x11,x0,loop                     ->      BEQ x11,x0, -6   :      "1111111-01011-00000-000-10101-1100011"
44      add x13,x10,x0                      ->      ADD x13,x10,x0   :      "0000000-00000-01010-000-01101-0110011"
48      jal loop                            ->      JAL x1,-16       :      "11111110000111111111-00001-1101111"

done:
52      sw x13,0(x5)                        ->      SW x13,0(x5)     :      "0000000-01101-00101-010-00000-0100011"
endc:
56      jal endc                            ->      JAL x1,0         :      "00000000000000000000-00001-1101111"
60      addi x0,x0,0                        ->      ADDI x0,x0,0     :      "000000000000-00000-000-00000-0010011"
```

The number of instructions has been reduced, so we must also modify by how much you have to move with respect to the current instruction when an uncoditional jump or a branch occur.
The content of the *Instruction.txt* file has therefore less lines and the execution of the loop takes less steps.

3.4 Memories implementation

The Instruction and Data Memories are included in the testbench, as specified in the assignment, and are both implemented as asynchronous memories. Each memory has 128 words of 8 bits (1 byte), despite the input and output data being on 32 bits.
During the writing phase, the 32-bit input data is portioned into 4 groups of 8 bits each: the spaces corresonding to the `ADD` address passed to the memory and the subsequent positions ADD+1, ADD+2 and $ADD + 3$ are filled.
In the reading phase, the process is similar: the output data are generated by combining the information located at the position passed as the ADD address plus the three subsequent positions $ADD + 1$, $ADD + 2$ and $ADD + 3$.

In the testbench, the initialization of these memories is done by reading at each clock cycle the input data from two text files, one for each memory, so that they're filled as reported in *fig.3.17* and *fig.3.18*.
Once they're initialized, memories can be connected to our processor which can be started, by disabling the reset. In particular, the *data maker* is used to fill the memories before giving the start signal to the processor.