Politecnico di Torino

Integrated Systems Architecture

A.A. 2020/2021

Prof. Guido Masera
Prof. Maurizio Martina

Design of a RISC-V-lite processor

21 Feb 2021

Antona Gaspare	275336
Barrera Alessandro	275337
Liu Huicai	273375

Link to GitHub: `https://github.com/Dragosk97/ISA-laboratories.git`

Contents

1 Introduction

The aim of this laboratory is to design in VHDL a RISC-V-lite processor with 5 pipeline stages. The version of RISC-V ISA module chosen for this laboratory is the RV32I, which has a 32-bit fixed-width instruction set, where instructions are organized in six classes:

- R or register;
- I or immediate;
- S or store;
- SB or conditional branch;
- U or upper immediate;
- UJ or unconditional jump

A subset of instructions of the whole RV32I supported in the RISC-V-lite processors are the following:

- arithmetic add, addi, auipc, lui
- branches beq
- loads lw
- shifts srai
- logical andi, xor
- compare slt
- jump and link jal
- stores sw

1.1 Instruction Set

From **The RISC-V Instruction Set Manual** it is possible to get the formats of the instructions, as shown in the following table:

7	5	5	3	5	7	
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

The 32 bits are divided mainly in 6 fields. Each one has a fixed number of bits and an almost fixed functionality, with some variations from one format to another. In particular for the interested instructions, the bits are organized as shown in the following table:

0000000	rs2	rs1	000	rd	0110011	ADD
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[31:12]				rd	0010111	AUIPC
imm[31:12]				rd	0110111	LUI
imm[12—10:5]	rs2	rs1	000	imm[4:1—11]	1100011	BEQ
imm[11:0]		rs1	010	rd	0000011	LW
0100000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	010	rd	0110011	SLT
imm[20—10:1—11—19:12]				rd	1101111	JAL
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

From the **Manual**, the RV32I XLEN is 32. It is also possible to read what each instruction does:

ADD It performs the addition of rs1 and rs2. Overflows are ignored and the low XLEN bits of results are written to the destination rd.

ADDI It adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo instructions.

AUIPC (add upper immediate to pc) it is used to build pc relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register rd.

LUI (load upper immediate) it is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.

BEQ All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ± 4 KiB. Branch instructions compare two registers. BEQ take the branch if registers rs1 and rs2 are equal.

LW & SW Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

SRAI Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in bit 30. SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

ANDI It is a logical operation that performs bitwise AND on register rs1 and the sign-extended 12-bit immediate and places the result in rd.

XOR It perform bitwise logical operations between rs1 and rs2 and the result is placed in rd.

SLT It perform signed compares writing 1 to rd if rs1 $<$ rs2.

JAL The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ± 1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

2 RISC-V

In order to increase the throughput of a generic architecture, it is possible to apply the pipeline technique as for the RISCV processor. The pipeline structure increases the throughput by reducing the critical path delay and therefore it allows a higher operating frequency. But as described in the following section, the pipeline introduces also some problems, so it brings to a more complex datapath. [1]

2.1 RISC-V pipeline

The RISC-V architecture can be pipelined in five stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write Back (WB). The pipelined organization provides a better throughput, but the elaboration of more instructions at the same time creates the possibility for hazards to occur. In particular, three types of hazards are defined: structure, data and control hazards.

Structure Hazard This type of hazard is related to the conflicts due to multiple instructions using a certain resource at the same time. In RISC-V case, those hazards are resolved by having separated instruction and data memory to avoid conflicts when a load/store operation would impede the instruction fetch, and a register file in which it is possible to read and write at the same time from different ports. The remaining resources can't cause a conflict of this type.

Data Hazard These hazards are related to the data dependencies among instructions. In particular, data dependencies would impede an instruction to proceed if the instruction computing one of its operators is still being elaborated. This condition can be easily detected by comparing the source and destination registers of the operations and, as a first solution, possibly stall the instruction. Nonetheless, it is possible to use the **forwarding** technique to resolve the cases in which a data dependency is detected, but the needed data is already available inside the pipeling stages. In this way it is possible to avoid most of the stalls, therefore improving the performance of the processor.

Control Hazard This type of hazard is related to the branch operations, as they decide the next instruction to be fetched. In a pipelined architecture, this decision is taken after or while the next instruction should already being fetched, therefore some stall operations should be required each time a branch instruction occurs. This would drastically reduce the performance of the processor, thus different solutions have been proposed throughout the years. In the presented work, **branch prediction** is exploited. Moreover, some architectural expedients are used to anticipate the branch outcome evaluation and this requires an additional forwarding as discussed in section 2.1.3.

2.1.1 Branch evaluation anticipation

In order to reduce the penalties due to the jump, the operations needed for the branch evaluation are anticipated in the instruction decode stage. This entails the need for a comparing unit, besides the earlier availability of the operators. Moreover, the forwarding would be necessary also during the decode stage.

2.1.2 Forwarding

The forwarding technique (or bypassing) is used to remove a data hazard when the needed value of an instruction is available inside the pipeline stages. In particular, a **forwarding unit** is used to compare the source and destination registers among different instructions in order to detect a data dependency which is resolvable by forwarding. If this is the case, some control signals are sent to the proper multiplexer to route the correct data.

In the presented work, this technique is used for both execution stage and branch evaluation, which is placed in the instruction decode stage in order to anticipate as much as possible the branch outcome. It is worth noticing that the two designed forwarding units are identical and both necessary in their completeness in order to minimize the need for stalling. In fact, while the forwarding is obviously needed in the instruction decode stage for the branch instructions, there are some additional cases that are covered by the forwarding in the execution stage. In particular, the forwarding in the instruction decode stage does not cover the following cases:

- if the instruction in the decode stage has a data dependency with the immediately previous one, the needed data is still being computed, therefore it will only be available in the next cycle. In this case, a forwarding between memory stage and execution is needed;
- if the instruction in the decode stage has a data dependency with a load instruction currently in memory stage. In this case, it is still not possible to forward during this step, but it is possible to forward during the next cycle, when the data is actually needed.

Whenever a data dependency cannot be resolved by forwarding, a stall operation is necessary. This situation is detected by the **hazard detection unit**, which stalls the instruction fetch and decode stages by inserting a NOP in the execution stage. The conditions for which the data hazard is not removable are condensed in the following:

- A branch instruction has a data dependency with the immediately previous instruction. In this case, the required data is still being computed in the execution stage, while the branch outcome needs to be evaluated at the same time in the instruction decode stage. Therefore a NOP will be inserted between the two and the forwarding unit will provide the needed value in the next cycle.
- An instruction has a data dependency with the immediately previous load

instruction. Also in this case, the load instruction would provide the data in the same cycle as it would be needed by the following instruction. It is important to notice that the NOP is inserted when the load is still in execution stage, therefore the hazard detection unit has to perform the same operations as in the first case.

- A branch instruction has a data dependency with a load instruction which is currently in memory stage. Also in this case, a NOP needs to be inserted in order to forward the data loaded from the memory in the next cycle, while the branch instruction is stalled in instruction decode stage.

2.1.3 Branch prediction

The penalties brought by the branch instructions are strongly impacting on the performance of the processor, especially if it is considered the regularity with which branch instructions are present in a code. For this reason, many solutions have been provided. One of these is branch prediction, which is a mechanism that tries to guess whether the branch will be taken or not. This technique can be implemented based on two approaches: static and dynamic prediction. It requires some components which are identifiable as:

- the event selection: recognizes which instruction needs the prediction;
- the prediction indexing: a table which associates the event with a specific entry of the table;
- the prediction mechanism: based on the content of the table, it decides the prediction;
- the feedback mechanism: compare the prediction with the execution outcome in order to modify the content of the table and the prediction mechanism. This makes the prediction technique adaptive and may improve its efficiency to a specific code currently being executed.

Therefore, the branch prediction technique requires some hardware components grouped in:

- prediction structures: the Prediction Table (PT), a set of multiplexers that load in the Program Counter the predicted target address;
- a mispredict recovery mechanism: a register that stores the next PC in case of sequential execution, a mechanism that corrects the PT and provides the correct target address in case of branch taken.

In the presented work, the branch prediction is based on a 1-bit prediction, exploiting a Cache-like Prediction Table (PT). In particular, the Program Counter (PC) is used in such a way that a portion of its address is used as indexing and the remaining bits are used as TAG within the Cache-like PT. In addition, the PT contains the prediction bit and the target address in case the branch is predicted to be taken.

Whenever a TAG comparison generates a "hit", either the sequential address or the target address stored in the PT are routed in the PC, according to the prediction bit.

Once the branch outcome is evaluated, both the decision and, if required, the target addresses are compared. If they are identical, the prediction was correct and the execution can continue without any penalty. Otherwise, the fetched address was wrong, so the misprediction needs to be handled. This consists of flushing the IF/ID registers, load the correct address in the PC and proceed with the fetching operation. In this case, the penalty is just one cycle and both the decision and the target address are updated in the PT.

2.2 Implementation

The design of the RISC-V architecture is implemented in VHDL. The top-level entity instantiates each pipeline stage as `component`, which are designed as described below.

2.2.1 Instruction fetch stage

A new instruction needs to be fetched from the Instruction Memory during each cycle, possibly providing a branch prediction and routing the next address to be loaded in the Program Counter. These operations, together with the first pipeline registers referred as IF/ID, are included in the **fetch_stage** entity.

The fetch operation requires a 32-bit register called the Program Counter (PC), which stores the address needed to index in the Instruction Memory (IM) the instruction to be fetched. Moreover, the next address has to be provided as input to the PC. Several possibilities may occur to decide which this value will be. In general, it may proceed in sequential order, jump to a specific address called the Target Address (TA) or reset to a starting value.

In order to proceed in sequential order, it is sufficient to increment the PC value. As this represents a specific byte in memory and each instruction is 4-byte long, the PC must be incremented by 4.

The random order may occur for either a jump or a branch instruction. In the first case, the TA is computed in the instruction decode stage and it is fed to the PC through the appropriate multiplexers. In addition to these operations, a branch instruction may provoke the branch prediction to provide the TA, which is also routed accordingly.

If a reset occurs, the PC input is driven to a starting address. This is used to start the execution of the program from its initial address.

All these different cases are handled by a proper set of multiplexers, ordered in such a way to respect the priority of these events. For example, if in the same cycle, a jump is in instruction decode stage and a branch is in instruction fetch stage providing a certain prediction, the jump's TA is the correct value to be fed as input to the PC.

In order to implement the branch prediction, the instruction fetch stage must also be provided with a Cache-like Prediction Table (PT). In particular, the presented work exploits 4 bits (from 2 to 5) of the PC as indexing of the table. Comparing the TAG field within the PT with the remaining 26 bits of the PC, a hit signal is generated and used to validate the prediction itself. With this organization, all the bits of the PC are used with the only exception for the two least-significant bits, which are not useful to address an instruction. As a consequence, no aliasing among different branches is possible, even though the PT addresses only 16 locations.

A misprediction is reported by the signal `wrong_prediction` computed in the instruction decode stage. Two cases may occur: either it is necessary to load the TA or the sequential order has to be restored. In the first case, the correct TA is always been computed in the instruction decode stage, independently if it is already stored in the PT. Therefore, the TA provided is simply routed to the PC input. In the second case, the correct address is provided by a dedicated register which always samples the PC+4 value. In addition, the PT is updated, storing the TAG, TA and prediction based on the execution outcome.

The registers IF/ID are instantiated in this VHDL entity. These have to support both a reset, implemented as the load of a NOP instruction, and an enable signal which can be disabled in order to repeat the decode of an instruction when a NOP is inserted in the execution stage.

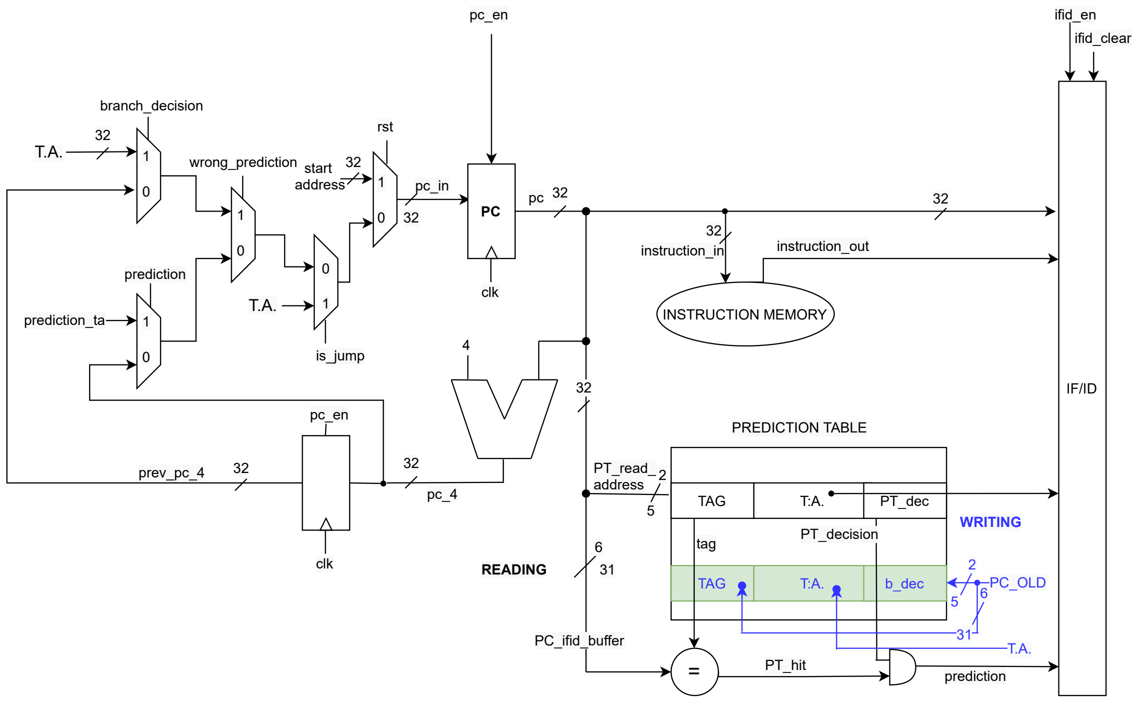


Figure 1: fetch stage

2.2.2 Instruction decode stage

After the fetch stage, the instruction moves in the instruction decode stage, where its operators should be provided. The structures needed in order to handle these operations are instantiated in the **decode_stage** entity. In particular, a Register File (RF) is implemented in order to organise all the internal registers of the RISC-V architecture, along with a Control unit which generates all the required control signals according to the elaborated instruction. In addition, an Immediate Generator is needed to compose the immediate value starting from the read instruction. Lastly, also the branch outcome has to be evaluated, comparing it with the branch prediction and generating the `wrong_prediction` signal.

The RF is implemented with an asynchronous reading in order to provide the data within the cycle and a synchronous writing.

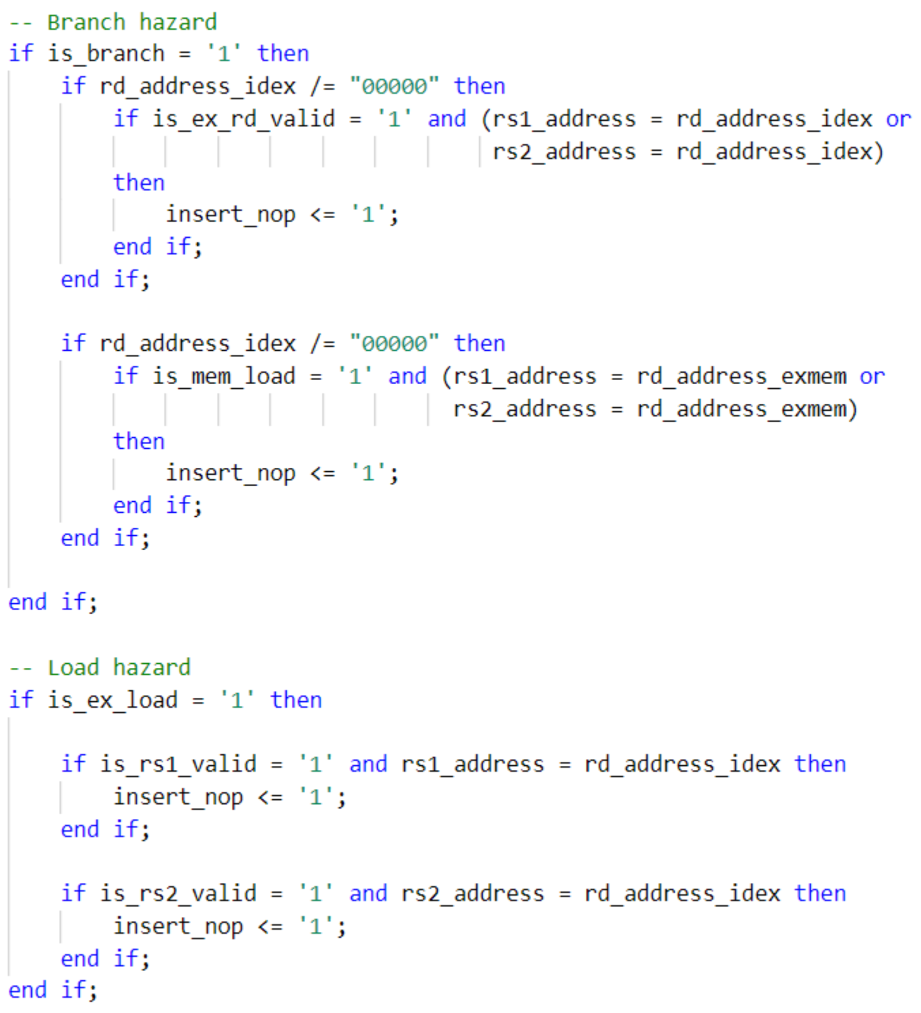
The control unit is implemented as a combinational logic, which receives as input the section of the instruction corresponding to the opcode. This input is sufficient to determine the control signals needed to implement the operation. These signals have to be delivered to all the elements in the processors, observing the correct timing in the pipeline.

The immediate generator has to possibly define the immediate value from the read instruction. Depending on their class, the immediate is organized in different fields of the instruction, therefore it is generated accordingly.

A forwarding unit is included in this stage in order to apply the forwarding to the branch evaluation. This will also be useful for a generic instruction, for example when one of the operators is written in the RF in the same cycle as it should be read. In order to enable its operation, the forwarding unit exploits some signals which declares whether a data in EX/MEM or MEM/WB registers is valid. The delayed signal RegWrite is suited for this operation since it is normally used for enabling the write operation in the RF during the WB stage. Whenever a data dependency is detected, the forwarding unit generates two selection signals for the two multiplexers, routing the available data.

On the other hand, if a data dependency is not resolvable by the forwarding, a NOP should be inserted in such a way that the forwarding will be able to handle the data dependency in the next cycles. This operation is provided from the hazard detection unit, which is included in the instruction decode stage as well. This unit must be able to detect the proper conditions for which the NOP must be inserted, exploiting some flag signals as shown in figure 2. Here it is possible to see that the three situations presented in section 2.1.2 are identified by the combination of some asserted flag signals corresponding to the fact that the instruction in decode is a branch, if the one in execution is a load and if the one in memory is a load.

The target address of both JAL and BEQ instructions is computed starting from the immediate and the value of the PC. For this operation, an adder is exploited.



```vhdl
-- Branch hazard
if is_branch = '1' then
    if rd_address_idex /= "00000" then
        if is_ex_rd_valid = '1' and (rs1_address = rd_address_idex or
                                     rs2_address = rd_address_idex)
        then
            insert_nop <= '1';
        end if;
    end if;

    if rd_address_idex /= "00000" then
        if is_mem_load = '1' and (rs1_address = rd_address_exmem or
                                  rs2_address = rd_address_exmem)
        then
            insert_nop <= '1';
        end if;
    end if;

end if;

-- Load hazard
if is_ex_load = '1' then

    if is_rs1_valid = '1' and rs1_address = rd_address_idex then
        insert_nop <= '1';
    end if;

    if is_rs2_valid = '1' and rs2_address = rd_address_idex then
        insert_nop <= '1';
    end if;
end if;
```

Figure 2: Extract from **hazard_detection_unit.vhd**

In order to evaluate the branch outcome, the BEQ instruction imposes to compare the two source registers indexed, and if they are equal, the branch has to be taken. For this purpose, a comparator is instantiated and its result is used as branch outcome.

Once both the branch outcome and the target address have been computed, these results have to be compared with the prediction that was performed when the BEQ was in the instruction fetch stage. This comparison is managed by the prediction validate unit and is needed in order to check whether the prediction was correct or not. If either the prediction or the target address was wrong, provided that it is actually needed, then the fetched instruction has to be flushed and the misprediction has to be handled. Therefore, the prediction validate unit is responsible for the generation of the `wrong_prediction` signal. A misprediction generates one cycle of penalty. On the other hand, if the prediction was correct, the execution can continue without any penalties.

As a JAL instruction is not predicted, it always requires the flush of the following instruction, implicating one cycle of penalty.

Also in this case, the entity **decode_stage** contains the output pipe registers ID/EX. In particular, these are provided of a `clear_idex` signal which is used to reset the registers when a NOP has to be inserted by the hazard detection unit, figure 3.

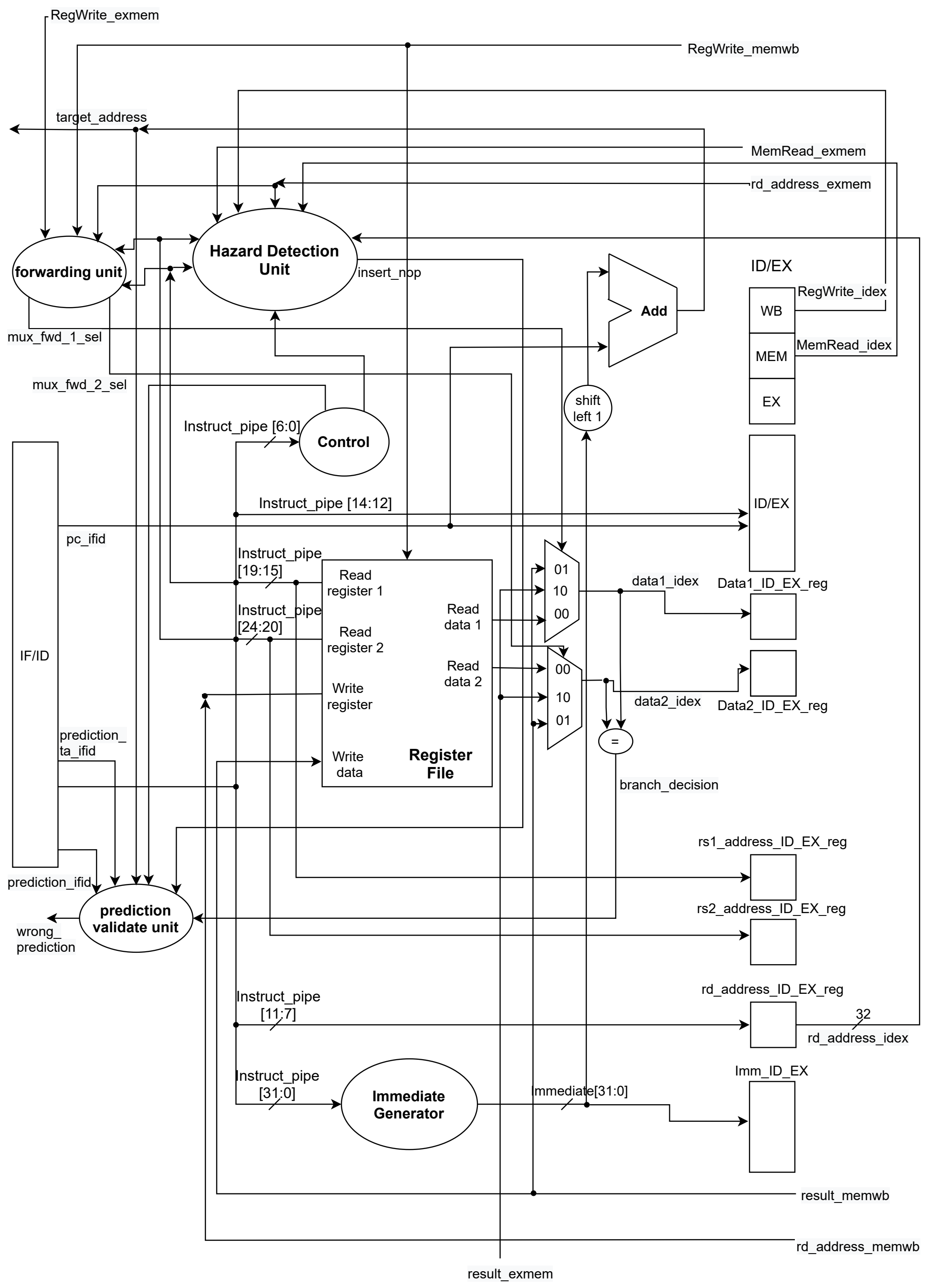


Figure 3: Decode stage

2.2.3 Execution stage

The execution stage is the one where the operations can be implemented. It contains different blocks like the ALU to do different procedures, the ALU Control used to define different instructions, multiple mux to manage the signals and the forwarding unit in order to minimize the need for stalling.

- ALU: block used to implement needed between two inputs of 32 bits. It is controlled by the 4-bits `ALU_ctr_input` signal that defines which kind of operation is done, in table 1.

ALU_ctr_input	Operation
0000	AND
0001	RIGHT SHIFT
0010	ADD
0011	XOR
0100	IS LESS THAN
others	ADD

Table 1: ALU opcode

- alu_control: it provides the control signal for the ALU in order to select different operations as a function of the signal `aluop` provided by the Control Unit, as shown in table 2.

aluop	funct3	Instruction	ALU_ctr_input
00	010	LOAD	0010
00	111	ANDI	0000
00	101	SHIFT	0001
00	010	ADDI	0010
10	000	ADD	0010
10	100	XOR	0011
10	010	SLT	0100
11	-	AUIPC, SW, LW	0010
others			0010

Table 2: ALU control table of truth

  Also, it is possible to have the default condition when `aluop`= 11, associated to the ADD instruction.

- mux1_alu_fwd: it is the first multiplexer 3-to-1 on the top of the scheme, controlled by the forwarding unit. It is used to choose among the data that arrives from the register ID/EX or the result obtained in the stage EX/MEM or MEM/WB.
- mux2_alu_fwd: it is the same as before, but defines the second input of the ALU based on another control signal from the forwarding unit.

- mux1_alu_pc: in this case it is possible to have a mux 2-to-1 which is used to decide between the data arrived from the PC or the output of the previous multiplexer. In this way, it is obtained the first input of the ALU.
- mux2_alu_imm: mux 2-to-1 where the control signal is used to decide if the second input of the ALU is the one arriving from the previous forwarding mux or the immediate value.
- `pc_next`: this signal represents the next sequential address of the PC, which is computed also in this stage by a dedicated adder.
- result_mux: this mux provide the final result, choosing among the ALU output, the immediate value or the signal `pc_next`.
- forwarding_unit: it is identical of the one in the decode stage, and in particular cover the cases not solved by the first one, as described in the section 2.1.2.

At the output, there are the pipeline registers EX/MEM, which are used to have the correct time scheduling of the control signals and to save the data before the memory stage. It is worth noticing that the data sampled from result_mux output is fed to the forwarding units.

2.2.4 Memory stage

In this stage, the data arrived from the previous register may be used in order to define the data to be written and the address to be provided to the data memory. Both the data coming from the execution stage and read from the data memory are sampled in the pipeline registers MEM/WB.

2.2.5 Write back stage

In this last stage the final multiplexer is implemented in order to choose if the signal wanted to go back to the register file comes from the data memory or the EX/MEM register.

2.3 Test of the processor

In order to test the processor a program *minv-rv.s* is exploited, this program takes an array v and computes the minimum absolute value present in the array.

In particular, the vector v = [10 -47 22 -3 15 27 -4] is saved in the data memory, and the program is divided into 4 main parts:

- **start**: the number of elements of the array v is saved in the register x16, the address of v in the register x4, the address of the memory in which the minimum will be stored in x5 and the temporary minimum is initialized with the maximum positive number;

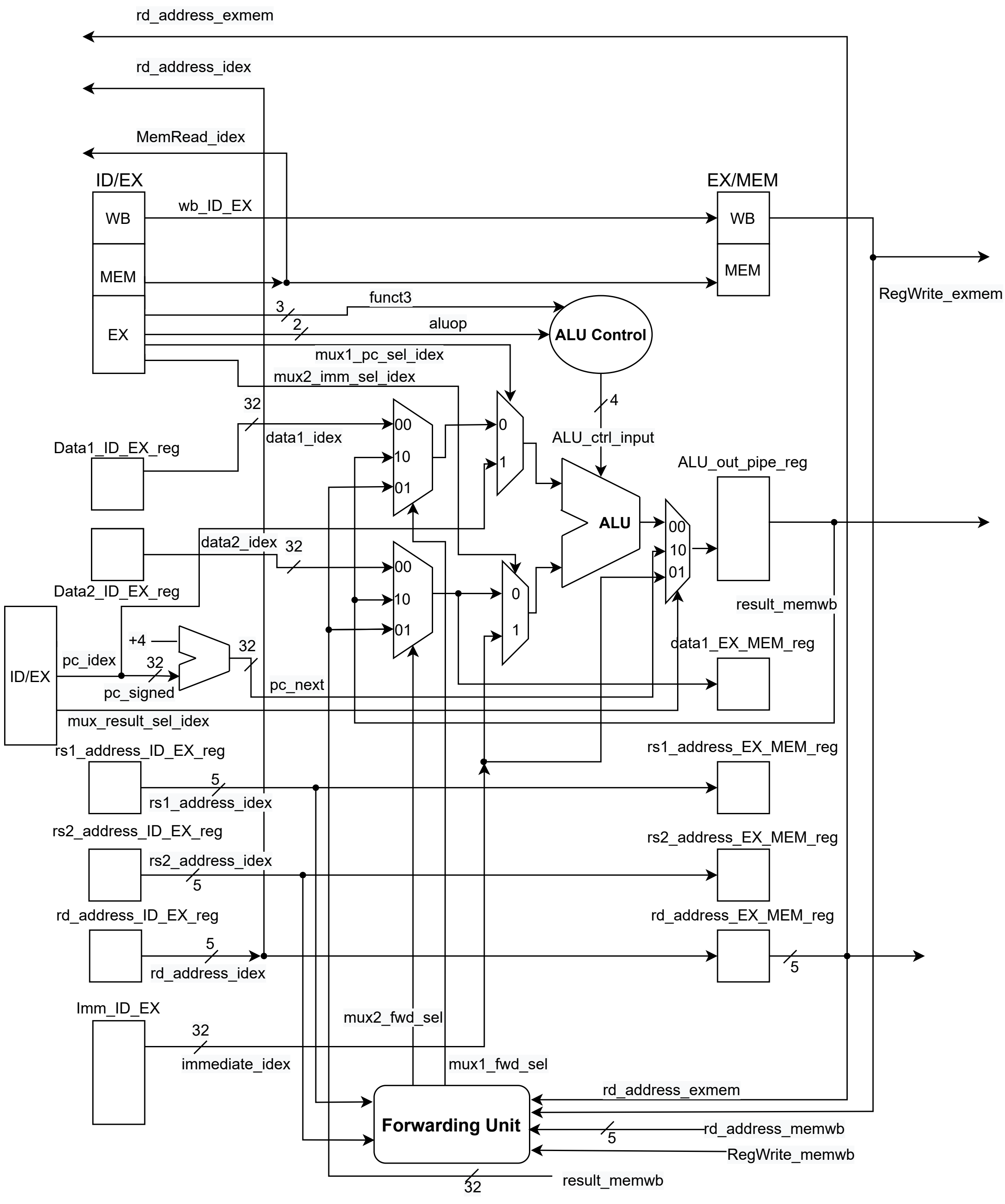


Figure 4: Execution stage

- **loop**: this *for* cycle examines the elements of v one by one, and it stops once all the elements are analyzed. First of all, the i-th element *v(i)* is taken from the memory and saved in the local register file, in particular in the register x8, with a load word instruction. Then, several instructions are used in order

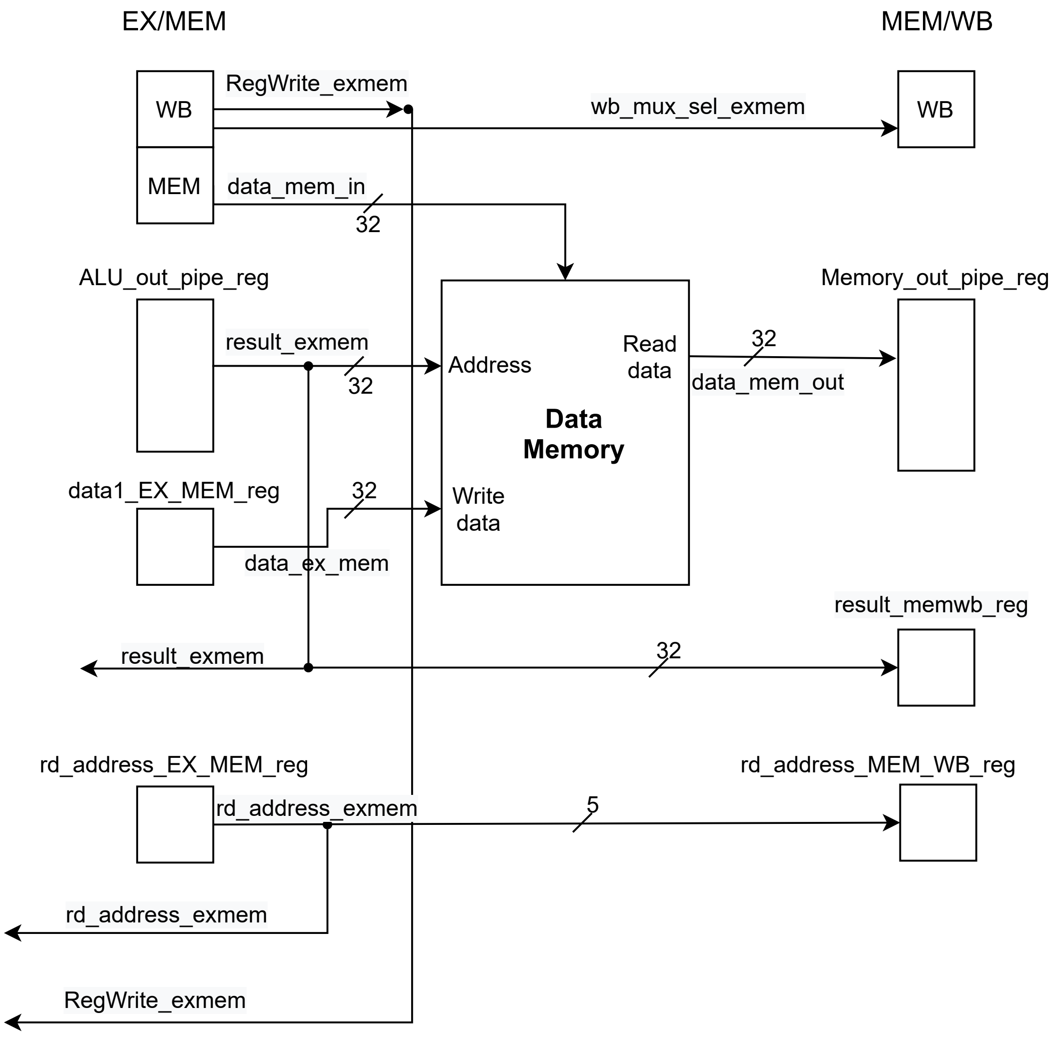


Figure 5: Memory stage

to compute the absolute value of *v(i)*. After that, this last value is compared with the actual value present in register x13 and possibly the content of x13 is updated. Finally, if all the elements have been analyzed, the program goes in the **done** part, otherwise the loop restarts;

- **done**: the final value present in the register x13 is saved in the memory, precisely at the location pointed by the content of register x5.
- **endc**: since it is not possible to stop the processor, two instructions, jal and addi, are used to implement an infinite loop.

In order to translate the ASM to machine language, a RARS simulator is used, in this way the values to be saved in the instruction memory and data memory are obtained. Now all the preliminary work to test the RISC-V processor is done. The simulation is performed by exploiting a top-level Verilog module test-bench. The VHDL entities used are:

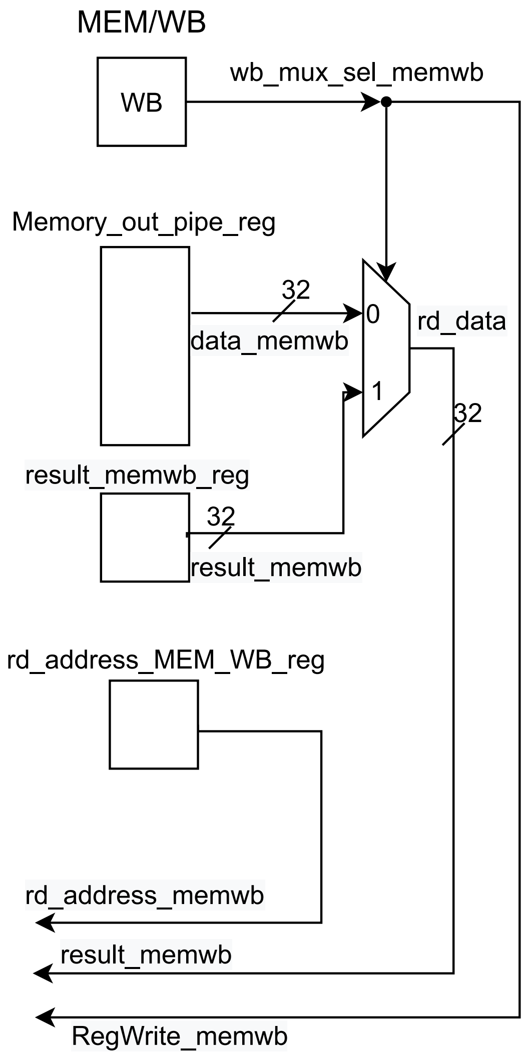


Figure 6: Write back stage

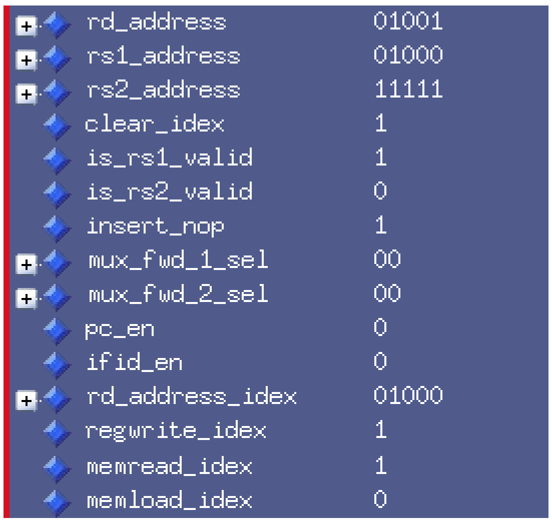
- **clk_gen.vhd**: it generates the clock and reset signals and provides the start address of the processor;
- **RISCV.vhd**, which is the description in VHDL of the processor;
- the entities **data_memory.vhd** and **instruction_memory.vhd** are not inserted in the VHDL entity of the RISC-V processor, but they are connected to it through the test-bench.

The data memory is implemented as a RAM, in which once the reset signal is asserted, the memory is initialized by reading a .txt file. On the other hand, the instruction memory is implemented as a ROM, where the content is defined as a constant. In both cases, the address ranges are defined with two parameters, `start_index` and `stop_index`, which have been decided according to the generated code.

2.3.1 Simulation

The processor has been tested by using ModelSim. It is possible to see in evidence some particular conditions, for example:

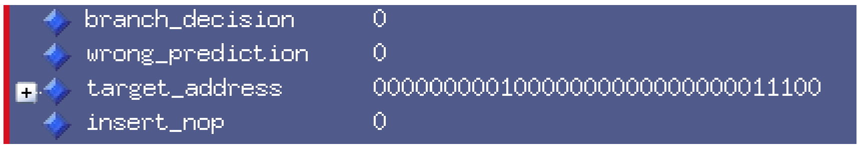
- When the LW instruction arrives in execution stage, SRAI is in decode, therefore it is possible to detect a data dependency as the addresses of source and destination registers are equal. This is one of the cases where the forwarding unit is not able to resolve the data dependency. This condition is identified by the hazard detection unit, which asserts the signals `insert_nop` and `clear_idex` in order to insert the NOP instruction as expected. It is also possible to see that `pc_en` and `ifid_en` are set to '0' in such a way to stall the instruction fetch and decode stages. This situation is shown in figure 7. In the following clock cycle, the forwarding unit in the instruction decode stage does not resolve the data dependency as the data required is not available yet. Nonetheless, the execution may proceed as in the next cycle the forwarding unit in execution stage will be able to provide the correct value, resolving the data hazard.



```
rd_address        01001
rs1_address       01000
rs2_address       11111
clear_idex        1
is_rs1_valid      1
is_rs2_valid      0
insert_nop        1
mux_fwd_1_sel     00
mux_fwd_2_sel     00
pc_en             0
ifid_en           0
rd_address_idex   01000
regwrite_idex     1
memread_idex      1
memload_idex      0
```

Figure 7: Data hazard detected and NOP insertion.

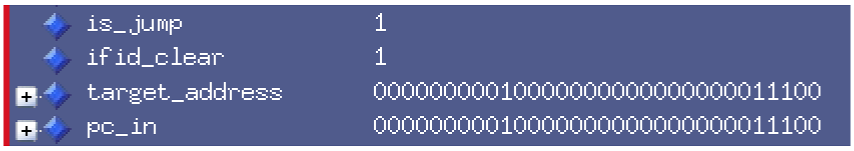
- It is interesting to analyze the situation when the second branch instruction is in fetch, namely when the instruction address corresponds to 0x00400040. At this point, since this is the first execution of this instruction, the TAG comparison of the Prediction Table sets the signal `pt_hit`=0. In the following clock cycle, analyzing the decode stage, it is possible to notice that `insert_nop`=1, which disables the prediction validation, forcing `wrong_prediction` to '0'. This is due to the fact that the branch outcome cannot be evaluated since there is a data dependency not solved. After another clock cycle, `insert_nop` becomes equal to '0' and the outcome of the branch can be evaluated. Since the branch is resulted to be untaken, `wrong_prediction` is still set to '0'. This situation is shown in figure 8.



```
branch_decision     0
wrong_prediction    0
target_address      00000000010000000000000000011100
insert_nop          0
```

Figure 8: BEQ untaken

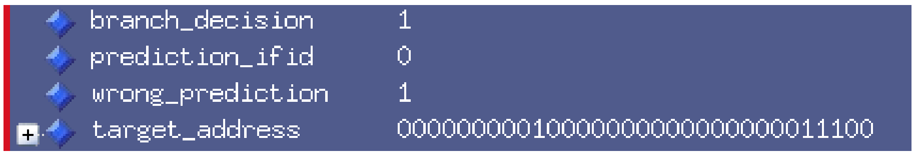
- When the instruction address is 0x00400048, the JAL is fetched. In the following clock cycle, the signal `is_jump` is asserted, therefore the instruction decode stage computes the `target_address`. This is used in the instruction fetch stage as input of the PC `pc_in` through the proper multiplexer. In addition, it is possible to see that the signal `ifid_clear` is asserted in order to flush the wrong instruction fetched due to the sequential execution of the program. Figure 9.



```
is_jump             1
ifid_clear          1
target_address      00000000010000000000000000011100
pc_in               00000000010000000000000000011100
```

Figure 9: JAL instruction

- At the second round of the loop, the BEQ is evaluated again, but this time the branch outcome results in a branch taken. Since the PT is still empty, the prediction bit is set to '0', causing the prediction validation unit to detect a wrong prediction, setting the signal `wrong_prediction`=1. This situation is shown in figure 10, where it is also possible to see the computed target address.

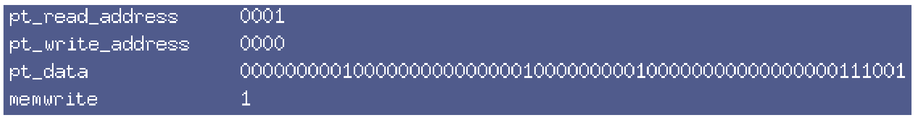


```
branch_decision     1
prediction_ifid     0
wrong_prediction    1
target_address      00000000010000000000000000011100
```

Figure 10: Wrong prediction

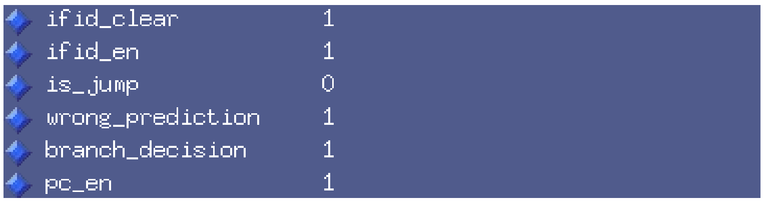
Since `wrong_prediction`=1, within the instruction fetch stage, the `target_address` is provided to the PC and the Prediction Table. In figure 11 the internal signals of the PT are shown. In addition to the write enable signal `MemWrite`=1, `PT_data` is the written data at the location of the prediction table corresponding to the BEQ instruction, composed as 26 bits of TAG, 32 bits of `target_address`, and 1 bit of prediction.

- Whenever the instruction fetched has to be flushed, the signal `ifid_clear`=1 in order to reset the register IF/ID. This entails the load of a NOP instruction, implemented as a ADDI x0, x0, 0x0. In figure 12 is presented the case of the flush of a fetched instruction as a result of a wrong prediction.



```
pt_read_address     0001
pt_write_address    0000
pt_data             00000000010000000000000001000000000100000000000000000111001
memwrite            1
```

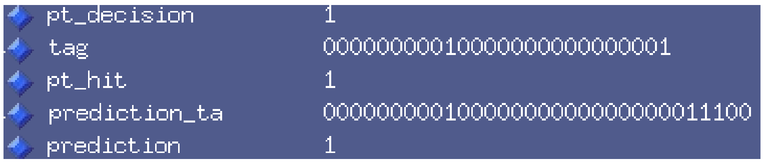
Figure 11: Prediction Table



```
ifid_clear          1
ifid_en             1
is_jump             0
wrong_prediction    1
branch_decision     1
pc_en               1
```

Figure 12: Flush

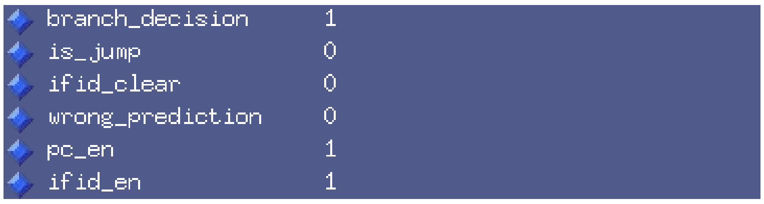
- After one entire loop, it is possible to see in evidence the case in which the previous BEQ is fetched, setting the signal `pt_hit`=1, which means that in the Prediction Table there is a match. The prediction is to have a branch taken since the bit `prediction`=1, so the Target Address stored in the PT is loaded in the PC. Figure 13.



```
pt_decision         1
tag                 00000000010000000000000001
pt_hit              1
prediction_ta       00000000010000000000000000011100
prediction          1
```

Figure 13: Branch prediction

After another clock cycle, the branch is in instruction decode stage, so `insert_nop` is assert because of the data hazard. Another clock cycle is needed to have valid branch outcome, which corresponds to a `branch_decision`=1, meaning that the prediction was correct. It is also possible to notice that the signal `wrong_prediction`=0 as this situation is shown figure 14.

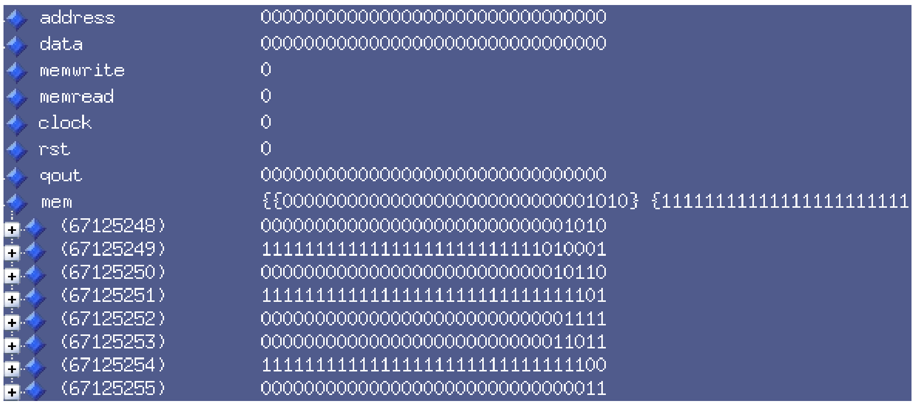


```
branch_decision     1
is_jump             0
ifid_clear          0
wrong_prediction    0
pc_en               1
ifid_en             1
```

Figure 14: Validation of branch prediction

- Finally the SW implicates that `MemLoad`=1, thus the final value is written at the proper address. In figure 15 it is possible to see the correct final minimum within the data memory, equal to 3.

  Other simulations were executed with different sets of numbers in the vector v, thus the correct behaviour of the processor is proved.



```
address        00000000000000000000000000000000
data           00000000000000000000000000000000
memwrite       0
memread        0
clock          0
rst            0
qout           00000000000000000000000000000000
mem            {{00000000000000000000000000001010} {111111111111111111111111
  (67125248)   00000000000000000000000000001010
  (67125249)   11111111111111111111111111010001
  (67125250)   00000000000000000000000000010110
  (67125251)   11111111111111111111111111111101
  (67125252)   00000000000000000000000000001111
  (67125253)   00000000000000000000000000011011
  (67125254)   11111111111111111111111111111100
  (67125255)   00000000000000000000000000000011
```

Figure 15: Final value

2.4 Synthesis

Once the correct behaviour of the processor was proved, it was synthesized with Synopsys Design Compiler using a customized script *riscv_syn.scr*. The synthesis was repeated till the slack became equal to 0. The critical path delay found was $T_{cp} = 3.32\, ns$, corresponding to the maximum clock frequency equal to $f_{max} = 301.2\, MHz$, while the total area was $A = 21998.47 \mu m^2$. The file *elaborate.txt* is examined in order to check that all the memory elements are flip-flops and no latches were synthesized.

Then, the clock period is set to $13.28\, ns$ corresponding to the clock frequency $f_M/4 = 75.3\, MHz$, in order to generate the information required to complete the design with the place and route. This last step is done using the file *write_risc.src*. First, the cells were ungrouped to flatten the hierarchy and the *Verilog* rules for the names of the internal signals were imposed. Then, it was possible to save the file *.sdf*, which contains the information about the internal delays. At this point, both the netlist in *Verilog* and the constraints to the input and output ports must be saved, producing *.v* and *.sdc* files.

Before going on with the power analysis, the generated netlist is simulated with ModelSim with the purpose to prove the correct behaviour even after the synthesis. In order to calculate the power consumption of the processor, it was necessary to extract the information about the switching activities of the nodes from Modelsim and evaluate them with Synopsys, so the two programs must be used together. More in detail the few steps are:

- launch Modelsim and record the switching activity;
- convert the file with the switching activity from *vcd* to *saif*;
- evaluate the power consumption with Synopsys Design Compiler.

The *script_activity.do* was implemented to do the first operation.

After the compilation of the *.vhd* and *.v* files, the delay's one *.sdf* is included, so Modelsim generates a *.vcd* file containing the switching activity information. This one has been converted into a *.saif* that Synopsys is able to read and to use in order to estimate the total power consumption of the circuit. It is necessary to read the netlist, the *.saif* file generated by the Modelsim simulation and to set the clock signal in the design. Finally, the *report_power* command is executed for the design at frequency $f_M/4$.

Power report at f/4

```
Cell Internal Power  =   1.5341 mW   (86%)
Net Switching Power  = 260.1564 uW   (14%)
                       ---------
Total Dynamic Power    =   1.7942 mW  (100%)
Cell Leakage Power     = 470.6623 uW

                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power
--------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000
memory             0.0000           0.0000              0.0000             0.0000
black_box          0.0000           0.0000              0.0000             0.0000
clock_network      0.0000           0.0000              0.0000             0.0000
register        1.4170e+03         23.4140          1.8959e+05         1.6300e+03
sequential         0.0000           0.0000              0.0000             0.0000
combinational    117.0070         236.7429          2.8108e+05           634.8237
--------------------------------------------------------------------------------
Total           1.5340e+03 uW     260.1569 uW     4.7066e+05 nW      2.2649e+03 uW
```

It is possible to see in evidence that the *Total Dynamic Power* $= 1.794\,mW$, due to the sum of the *Cell Internal Power* of all the elements in the circuit and the *Net Switching Power* due to the commutation of the signals. There is also an amount of *Cell Leakage Power* $= 470.662\,\mu W$. From this two contributions, the total power was derived as:

$$Total\ Power = Total\ Dynamic\ Power + Cell\ Leakage\ Power = 2.26\,mW.$$

2.5 Place and Route

After the synthesis of the design, the last step of implementation is the place and route at $f_M/4$. It can be performed with the Cadence Innovus tool. There are several steps that must be followed. The design of the RISC-V is imported after loading the files *design.globals*, customized for the submitted design, and *mmm_design.tcl*, which specifies the setup for the timing analysis. The required steps to complete the design are followed in the same way as in the first laboratory activity. Lastly, it is optimized in order to achieve the required timing constraints, figure 16.

Now it is possible to proceed analyzing the time behaviour performing a *Parasitics extraction* using technology and geometry information. Then the timing analysis is performed for both setup and hold times. The slack was always positive as noticeable in the files **RISCV_postRoute.slk** and **RISCV_postRoute_hold.slk**.

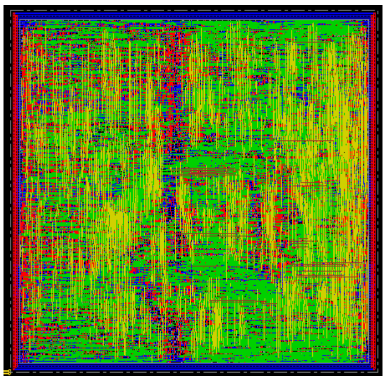


Figure 16: Post routing optimization

Differently, it would mean that the timing constraints are not respected. The next step is the design analysis and verification. Therefore, it was verified that both the connectivity and the geometry have no violations. At last, the area and the gate count data are computed using the *Gate Count* option in the Innovus tool.

The area $A = 21178.7\,\mu m^2$ is smaller than the previous one computed during the synthesis, due to a greater accuracy of this estimation after the place and route. Also, it is possible to estimate the number of gates = 26539 and the number of cells = 11261.

Finally, it is required to evaluate the power consumption post place and route doing similar steps to the ones of the procedure followed after the synthesis. Also in this case a simulation is done in order to prove the correct behaviour of the netlist but it is not necessary to translate in the *.saif* format, because Innovus is already able to communicate with ModelSim using *.vcd* files. Therefore, ModelSim can store the information related to the switching activity estimation of the circuit in the *vcd* directory. The script *script_activity_post_route.do* was exploited in order to perform the operations on ModelSim.

After the simulation, it is possible to go to the *innovus* directory and use the *Cadence Innovus* tool to do a power estimation using the switching activity evaluation saved in the *.vcd* file. Therefore, the *report_power_post_par.txt* is generated.

In evidence, there are different contributions of power: the $Total\ Dynamic\ Power = 2.724\,mW$, due to the sum of the *Total Internal Power* and the *Total Switching Power*, and the $Cell\ Leakage\ Power = 0.415\,mW$. From this two contributions, the total power was derived as:

$$Total\ Power = Total\ Internal\ Power + Total\ Switching\ Power +$$
$$+\ Total\ Leakage\ Power = 3.139\,mW$$

This value is greater than the *Total Power* estimated after the synthesis, because the evaluation after the place and route is more accurate and it also takes into account the power of the interconnections.

riscv_power_post_par.txt

```
Total Power
-----------------------------------------------------------------------------------------
Total Internal Power:       1.96838566     62.6889%
Total Switching Power:      0.75604479     24.0784%
Total Leakage Power:        0.41549397     13.2326%
Total Power:                3.13992443
-----------------------------------------------------------------------------------------
Group                          Internal   Switching     Leakage       Total  Percentage
                               Power      Power         Power         Power  (%)
-----------------------------------------------------------------------------------------
Sequential                        1.505      0.2185      0.1933       1.917       61.04
Combinational                    0.4636      0.5376      0.2222       1.223       38.96
-----------------------------------------------------------------------------------------
Total                             1.968       0.756      0.4155        3.14         100
```

3 Adding ABS instruction

A new instruction which performs the absolute value function is wanted to be added. The absolute value is an unary operation so only one source register and a destination register are needed. Therefore for this operation an I-type instruction can be used, filling the immediate field with zeros. The opcode chosen is "0 0001 11", which is not used by any other instructions.

3.1 Implementation

In order to execute this new instruction also the processor has to be modified, in particular there are two possibility:

- By inserting the new functional unit inside the ALU, in this case the following blocks should be modified:
  - Control unit, it should provide the signals for the new instruction such as the validation ones for the source registers, the RegWrite signal and the signals for the ALU control;
  - ALU control, now it should provide the opcode for the ALU in order to select the absolute function;
  - ALU, the new function should be added.
- By inserting the new functional unit as a special one outside the ALU, with this choice the following things should be done:
  - insert a new independent special unit with one input and one output;
  - change the mux3-to-1 at the out of the ALU into a mux4to1;
  - modify the Control Unit in order to drive correctly the mux4-to-1 described right before.

These two solutions are almost equivalent, but since the absolute value is an unary function, only one input, the second approach is preferred.

3.2 Testing

After the modifications, in order to prove the correct behaviour of the new processor, the original code used to test the standard RISC-V should be changed. In particular the instructions used to perform the absolute value can be substituted by an unique instruction.

In the previous case, there are 4 instructions that performs the absolute function, which are the instruction number 10(SRAI), 11(XOR), 12(ANDI), 13(ADD). This four can be substituted by abs x10, x8, which performs the absolute value on the content of the register x8 and saves the result in x10. But this is not enough: also the target addresses of the instruction auipc, beq, jump should be updated. These modifications can be done by hand or more efficiently using the free RARS simulator to translate into machine language the following instructions:

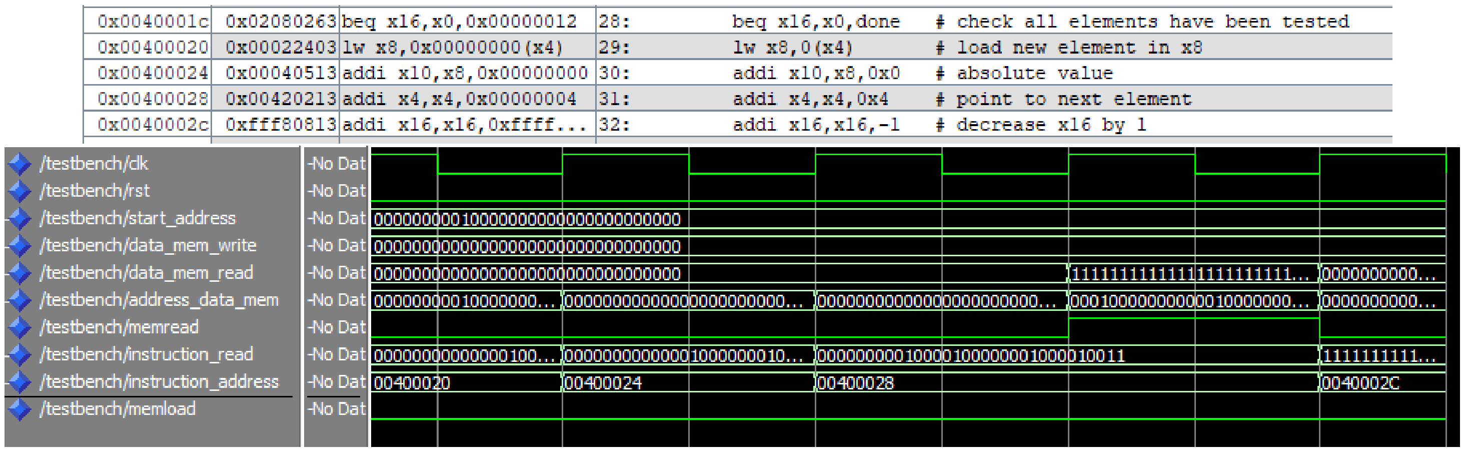
```
loop:
beq x16,x0,done   # check all elements have been tested
lw x8,0(x4)       # load new element in x8
addi x10,x8,0x0   # absolute value
addi x4,x4,0x4   # point to next element
addi x16,x16,-1   # decrease x16 by 1
slt x11,x10,x13   # x11 = (x10 < x13) ? 1 : 0
beq x11,x0,loop   # next element
add x13,x10,x0    # update min
jal loop          # next element
```

Where the instruction addi x10,x8,0x0 is then transformed into absolute value by simply changing the opcode from 0010011 to 0000111.

Finally the new instructions are saved in the instruction memory of the processor and a simulation on ModelSim is run. The vector to be analyzed is the same of the standard RISC-V processor: [10 -47 22 -3 15 27 -4].

The abs function corresponds to the instruction address 0x00400024. In figure 17 the instructions of the second cycle are shown, this cycle reads -47. It is possible to see that when the abs instruction is in the decode stage a stall is inserted since the previous instruction is a load word instruction and there is data dependency between them. Then when a new instruction is fetched, the abs instruction is in execute stage.

In figure 18 it is possible to see that the input A of the ALU corresponds to -47, and the input B is 0, the result is simply the sum of the two. But the output of the ALU is not chosen to be stored in the register, since the select signal coming from the control unit of the previous stage is "11". Therefore as *ex_result* the *abs_result*

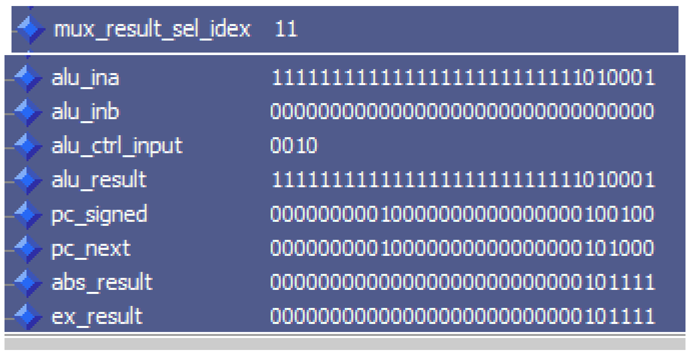


```
0x0040001c 0x02080263 beq x16,x0,0x00000012    28:  beq x16,x0,done    # check all elements have been tested
0x00400020 0x00022403 lw x8,0x00000000(x4)     29:  lw x8,0(x4)        # load new element in x8
0x00400024 0x00040513 addi x10,x8,0x00000000   30:  addi x10,x8,0x0    # absolute value
0x00400028 0x00420213 addi x4,x4,0x00000004    31:  addi x4,x4,0x4     # point to next element
0x0040002c 0xfff80813 addi x16,x16,0xffff...   32:  addi x16,x16,-1    # decrease x16 by 1
```

```
/testbench/clk                  -No Dat
/testbench/rst                  -No Dat
/testbench/start_address        -No Dat  00000000010000000000000000000000
/testbench/data_mem_write       -No Dat  00000000000000000000000000000000
/testbench/data_mem_read        -No Dat  00000000000000000000000000000000  11111111111111111111111111...  0000000000...
/testbench/address_data_mem     -No Dat  000000000100000000...  00000000000000000000000000...  00000000000000000000000000...  00010000000000010000000...  0000000000...
/testbench/memread              -No Dat
/testbench/instruction_read     -No Dat  0000000000000000100...  000000000000010000000010...  00000000010000100000001000010011  1111111111...
/testbench/instruction_address  -No Dat  00400020  00400024  00400028  0040002C
/testbench/memload              -No Dat
```

Figure 17: Instruction address of the absolute function

is chosen, which corresponds to 47.



```
mux_result_sel_idex  11
alu_ina              11111111111111111111111111010001
alu_inb              00000000000000000000000000000000
alu_ctrl_input       0010
alu_result           11111111111111111111111111010001
pc_signed            00000000010000000000000000100100
pc_next              00000000010000000000000000101000
abs_result           00000000000000000000000000101111
ex_result            00000000000000000000000000101111
```

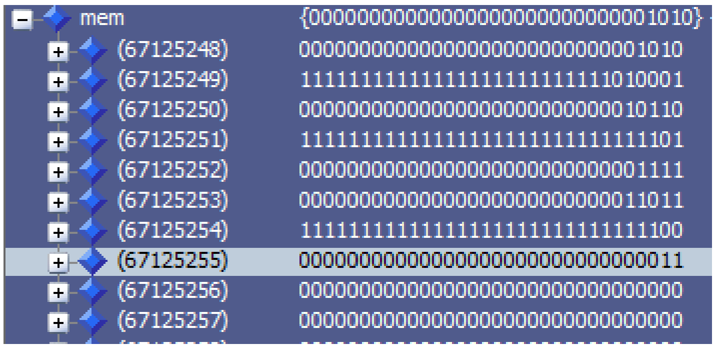
Figure 18: Signals in the execution stage

Since no other instructions were modified, the program was run to the end, that is 865 ns taking as clock period 10 ns, and the final result stored in the 8-th position of the memory is controlled. It is possible to see in figure 19 that it is exactly 3 as expected.

In this way the correct behaviour of the modified processor is proved.

3.3 Synthesis

Finally it is possible to proceed with the synthesis by using Synopsys Design Compiler. For this purpose a customized script *riscv_syn_abs.scr* is exploited. The synthesis was repeated till the slack becomes equal to 0. The critical path delay found was $T_{cp} = 3.32\,ns$, corresponding to the maximum clock frequency equal to $301.2\,MHz$, while the total area is $A = 22200.9\mu m^2$, greater than the previous one due to the introduce of a new block, *abs_unit.vhd*, used to implement the abs operation. The file *elaborate.txt* is examined in order to check that all the memory elements are flip-flops and no latches were synthesized.



```
mem             {00000000000000000000000000001010}
(67125248)      00000000000000000000000000001010
(67125249)      11111111111111111111111111010001
(67125250)      00000000000000000000000000010110
(67125251)      11111111111111111111111111111101
(67125252)      00000000000000000000000000001111
(67125253)      00000000000000000000000000011011
(67125254)      11111111111111111111111111111100
(67125255)      00000000000000000000000000000011
(67125256)      00000000000000000000000000000000
(67125257)      00000000000000000000000000000000
```

Figure 19: Values stored in the memory

So, the clock period is set to 13.28 ns corresponding to the clock frequency $f_M/4 = 75.3\,MHz$. Finally, it is necessary to save the data required to do a power analysis by using the file *write_risc_abs.src*. First, the cells were ungrouped to flatten the hierarchy and the *Verilog* rules for the names of the internal signals were imposed. Then, save the files *.sdf*, *.v* and *.sdc*.

As for the standard processor, a simulation is done in order to see the correct behaviour of the netlist and the same steps are repeated for the power analysis. So the estimation is performed using the command *report_power* at the frequency $f_M/4$.

Power report at f/4

```
Cell Internal Power  =   1.5217 mW   (87%)
Net Switching Power  = 236.1751 uW   (13%)
                       ---------
Total Dynamic Power    =   1.7579 mW  (100%)
Cell Leakage Power     = 475.3683 uW

                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power
--------------------------------------------------------------------------------
io_pad             0.0000           0.0000             0.0000             0.0000
memory             0.0000           0.0000             0.0000             0.0000
black_box          0.0000           0.0000             0.0000             0.0000
clock_network      0.0000           0.0000             0.0000             0.0000
register         1.4145e+03        19.3244          1.8955e+05         1.6234e+03
sequential         0.0000           0.0000             0.0000             0.0000
combinational    107.1802         216.8509          2.8582e+05          609.8459
---------------------------------------------------------------------------------
Total            1.5217e+03 uW    236.1753 uW      4.7537e+05 nW      2.2332e+03 uW
```

It is possible to see in evidence that the *Total Dynamic Power* = 1.758 mW, due to the sum of the *Cell Internal Power* and the *Net Switching Power*. There is also an amount of *Cell Leakage Power* = 475.368 μW. From this two contributions, the total power was derived as:

$$Total\ Power = Total\ Dynamic\ Power + Cell\ Leakage\ Power = 2.23\,mW.$$

As expected, the total power with the abs instruction is lower than the value obtained before, since the new structure implements less number of instructions.

3.4 Place and Route

After the synthesis of the design, the last step of implementation is the place and route at $f_M/4$. There are several steps that should be followed. It is imported the design of the RISCV_abs, loading the customized files *design.globals*, and *mmm_design.tcl*. There are also repeated all passages like the first laboratory to complete the design. It is also optimized in order to achieve the required timing constraints using *Post-Route*, figure 20.

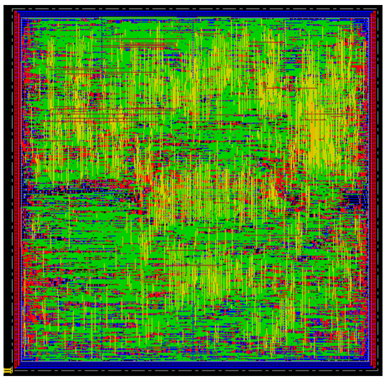


Figure 20: Post routing optimization

Now it is possible to proceed analyzing the time behaviour, using the parasitic resistance and capacitance values for each metal wire, which are evaluated performing a *Parasitics extraction*. Then the timing analysis is performed for both setup and hold times. The slack was always positive as noticeable in the files **RISCV_abs_postRoute.slk** and **RISCV_abs_postRoute_hold.slk**. The next step is the design analysis and verification. Therefore, it was verified that both the connectivity and the geometry have no violations. Finally, the area and the gate count data are computed using the *Gate Count* option in the Innovus tool.

The area $A = 21364.6\,\mu m^2$ is smaller than the previous one computed during the synthesis, due to a greater accuracy of this estimation after the place and route. Also it is possible to estimate number of gates equal to 26772 and the number of cells equal to 11447. But, obviously this RISCV_abs is greater than the area of the RISC-V because we need one more block to do the additional operation.

The last step is the power estimation after the place and route. It is possible to proceed with the same method of the previous implementation, before that also in this case a simulation is performed in order to prove the correctness of the netlist. The report below shows the results obtained from the power estimation.

In evidence, there are different contributions of power: the $Total\ Dynamic\ Power = 2.664\,mW$, due to the sum of the *Total Internal Power* and the *Total Switching*

riscv_abs_power_post_par.txt

```
Total Power
-----------------------------------------------------------------------------------------
Total Internal Power:        1.92536818     62.3904%
Total Switching Power:       0.73934925     23.9582%
Total Leakage Power:         0.42128153     13.6514%
Total Power:                 3.08599896
-----------------------------------------------------------------------------------------
Group                           Internal   Switching     Leakage       Total  Percentage
                                Power      Power         Power         Power  (%)
-----------------------------------------------------------------------------------------
Sequential                         1.464      0.2075      0.1934       1.865       60.43
Combinational                     0.4616      0.5318      0.2279       1.221       39.57
-----------------------------------------------------------------------------------------
Total                              1.925      0.7393      0.4213       3.086         100
```

Power, and the *Cell Leakage Power* $= 0.421\,mW$.

From this two contributions, the total power was derived as:

$$\begin{aligned} \textit{Total Power} = &\textit{Total Internal Power} + \textit{Total Switching Power} + \\ &+ \textit{Total Leakage Power} = 3.086\,mW \end{aligned}$$

This value is greater than the *Total Power* estimated after the synthesis, because the evaluation after the place and route is more accurate and it also takes into account the power of the interconnections. The *Total Power* for processor with the abs instruction is lower than the one without, due to the lower number of instructions executed.

References

[1] David A.Patterson and John L.Hennessy. *Computer organization and design, RISC-V edition*. Morgan Kaufmann, 2018.