Politecnico di Torino


Electronic Engineering

# LABORATORY 3

GROUP 14

*Authors:*
Carta Federica (267580)
Dimroci Enea (262947)
Pesaresi Danilo (262845)

*Prof:*
Guido Masera

14th November 2019

# Contents

# Chapter 1

# RISC V

## 1.1 Abstract

This report analyzes the different design steps to implement a RISC-V processor. This type of processor is thought as processor that manages simple operations, and consequently has a very limited Instruction Set.

Each stage of the processor are implemented in order to have the better trade-off in terms of complexity-speed.

In order to explain how the architecture is implemented, the work is divided in order to develop the RISC-V components separately and to merge them after the verification of each one. This division allows the team to assign different tasks in order to develop the project in a fast way. For this reason the report is organized dedicating a section for each main component.

After the first implementation the portability of the architecture is tested adding a new instruction to its Instruction Set.

Both architectures developed are analyzed in terms of power, area and routing.

## 1.2 Design of the architecture

As previously mentioned in the abstract, this type of processors manage simple operations. Specifically the Risc-V designed has an ISA restricted to the following instructions:

- arithmetic add, addi, auipc, lui;
- branches beq;
- loads lw;
- shift srai;
- andi, xor;
- compare slt;

- jump and link jal;
- stores sw.

The NOP operation is implemented by the instruction addi x0, x0, 0.

To implement the processor is initially considering the reference architecture in the Fig.1.1. Obviously a lot of changes are applied and described in the report but it is the starting point to the develop of the project.

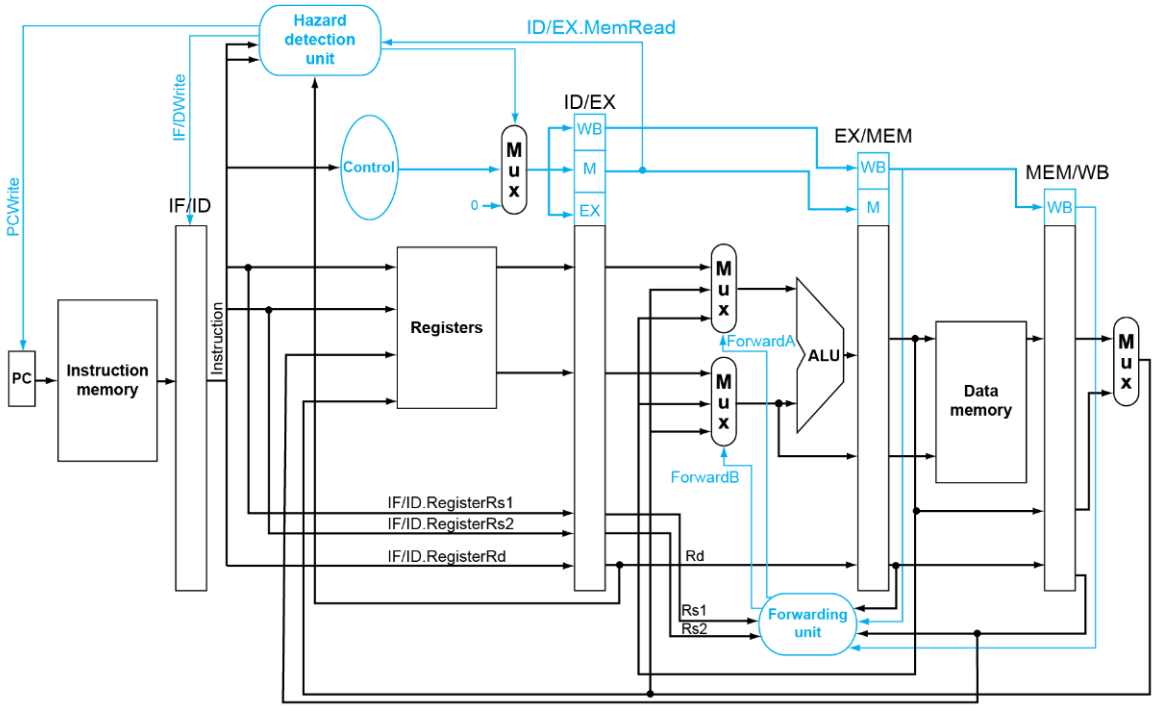


Figure 1.1: Reference architecture

The implemented architecture has 5 pipeline stages, to split the critical path between the instruction fetch and the end of its complete execution. The pipeline stages are the following:

- First stage: it's the instruction fetch stage (IF) in which it's chosen if the next address of the instruction has to be taken in a sequential way or if it's a jump address (in case of JAL or BEQ);
- Second stage: it's the decode stage (ID), in which the instructions are decoded and the extrapolated information are sent to the units they are needed to;
- Third stage: it's the execution stage, in which the arithmetical operation requested from the current instruction is performed;
- Fourth stage: it's the Memory stage (MEM) of interface with the data memory;
- Fifth stage: the Write Back stage (WB) that selects, depending on the instruction, the result source, in particular between the *Data Memory* and the *ALU*.

The implementation of these stages with their relative components and functionalities are described one by one in the following chapters.

### 1.2.1 Fetch stage

The architecture of this stage is composed by a 2 to 1 mux in charge of the selection of the sequential address or the jump address, to proceed in a sequential way or to select the jump address in case of the previous instruction was a beq or a jal.

To the sequential way it's used an incrementer. In particular would be necessary an adder able to do an increment of a factor 4, because one instruction is composed by 32 bits and an addressed cell of the instruction memory is of 1 byte. So the address done in input at the memory it's referred to the first byte location (using little endian convention) and the selected cell and the following three are given automatically in output. So it's thought that to spare some connections and HW, the input of the PC and so its computation into the architecture can be done without using the two LSBs, that are fixed at "00" value and the adder can be simplified with an incrementer designed by contraction, that increments of one the third LSB of the effective address.

The Program Counter register, also it contained in this stage, is for the previous explained reasons, a 30 bits register.

In this stage is necessary to insert the *Instruction Memory* that contains the code to execute.

### 1.2.2 Decode stage

In the decode phase, the instruction is sent in input to several blocks, then that communicate with each other:

- to the Control Block in order to generate the control signals associated with the current instruction;
- to the Hazard Detection Unit in order to recognize and manage possible dangerous situations;
- to the registers file that contains the temporary useful data to extract to perform the instruction;
- to the immediate generator block in order to reconstruct a possible constant value needed for the instruction.

A comparator in output of the registers file is placed with the aim to compute the equality condition in case of *BEQ* instruction. It can be formally considered as an execution part, but because of the result of the comparison decides the next instruction, it has to be done as soon as possible to avoid to lose time fetching the wrong instruction, and bringing to an instruction flush.

### Registers file

This block is a bank of 32 registers, of 32 bits. It is located into the Instruction Decode (ID) stage. The registers are sampled in the negative edge of the clock to not interfere with the timing imposed by the pipeline stages. The selection of the needed registers is done in two different ways, one for the writing and one for the reading:

- the reading is not driven by a control signal, but it's always enable, so the addressed registers are always read. This choice is done because almost all instructions need of the registers file reading, so adding this signal the cost of control would increase without effective savings.

  The selection is done by means of two 32-to-1 Mux, one for the first source register called by the instruction and one for the second source register if the fetched instruction requires it. These mux are implemented hierarchically using three 8-to-1 multiplexers and one of the size 4-to-1;

- In order to select the destination register to write, a decoder 5-to-32 is used, implemented with a hierarchical organization of littler decoders.

  This decoder is managed by the address of the destination and the *RegWrite* signal that enables it. As previously mentioned the output of the decoder is used in order to enable the writing of the correct register

### Hazard Detection Unit

The aim of this component is to avoid operation that causes data hazards, substituting a dangerous operation with one or more bubbles. This operation is done in the decode stage and takes into account 4 consecutive instructions, taking in input the new fetched instruction and the control signal relative of the previous three from the following stages. This unit avoid the detect hazards stalling the architecture, and so it's preferable to manage hazards in other way if it possible (for instance with the Forwarding Unit) to not have a loss of performance.

The architecture has 2 critical cases that can't be managed from the other units and so require the Hazard Detection Unit intervention:

- Data hazard after the load operation: indeed with the load operation the "loaded" value is available only in the last stage and if the following instruction needed this data, forwarding could be done too late (one clock cycle), and so in this case a bubble is inserted;

- Data hazard in the branch case: if the data to be compared are modified by one of the previous instructions, because of the source registers are compared in the decode stage, the Forwarding Unit cannot act on them, and so a variable number of bubbles are inserted (from one to three, depending on in which stage the needed data is). This operation is done using the information contained in the *RegWrite* signal in the different stages. Indeed the hazard is generated

only if one or both of the two source registers correspond to the destination register of one of the previous 3 instructions and only if it is an instruction that write (and so change) the register of interest.

These 2 cases are simulated and the results are shown in the figure 1.2 (blue for the Branch case, with three bubbles, red for the load case, with one bubble).

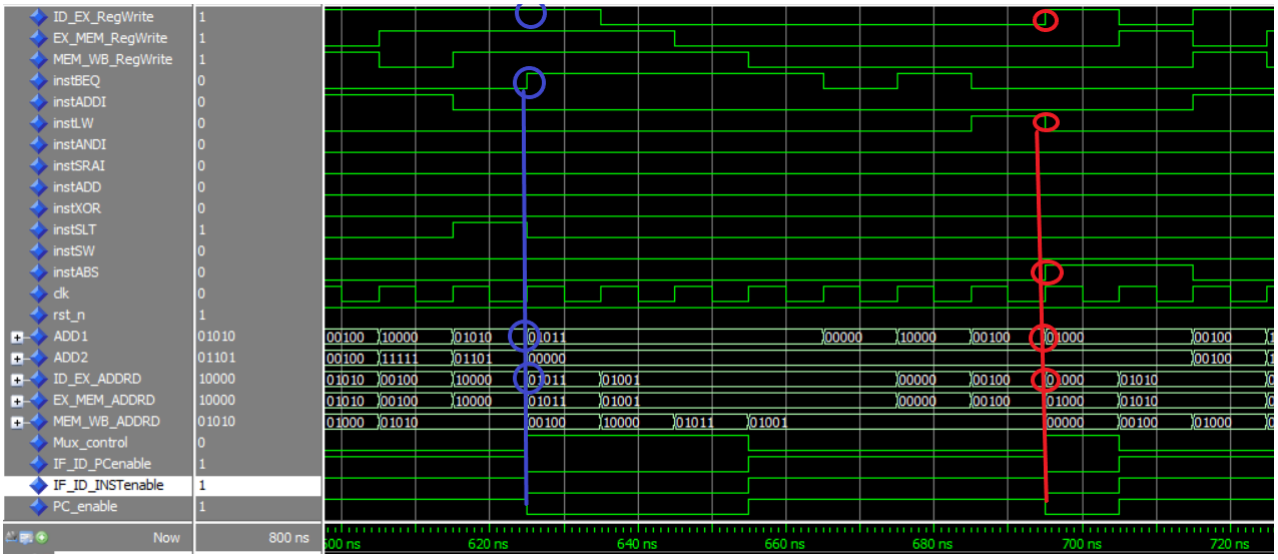


Figure 1.2: Hazard Detection Unit simulation

The hardware of this unit is made of the following components:

- Comparators: in order to compare the address of the source registers with the address of the destination register of the other stages;
- FFs: two of them("*enbobble2*", "*enbobble3*") in order to count the number of bubbles in the branch case, one of them ("*ffLW*") in order to delay and guarantee the correct timing of the instLD signal in the load case;
- Logic gates net: in order to generate the outputs taking into account all the cases previously described;

The RTL design of the block is shown in the figure 1.3

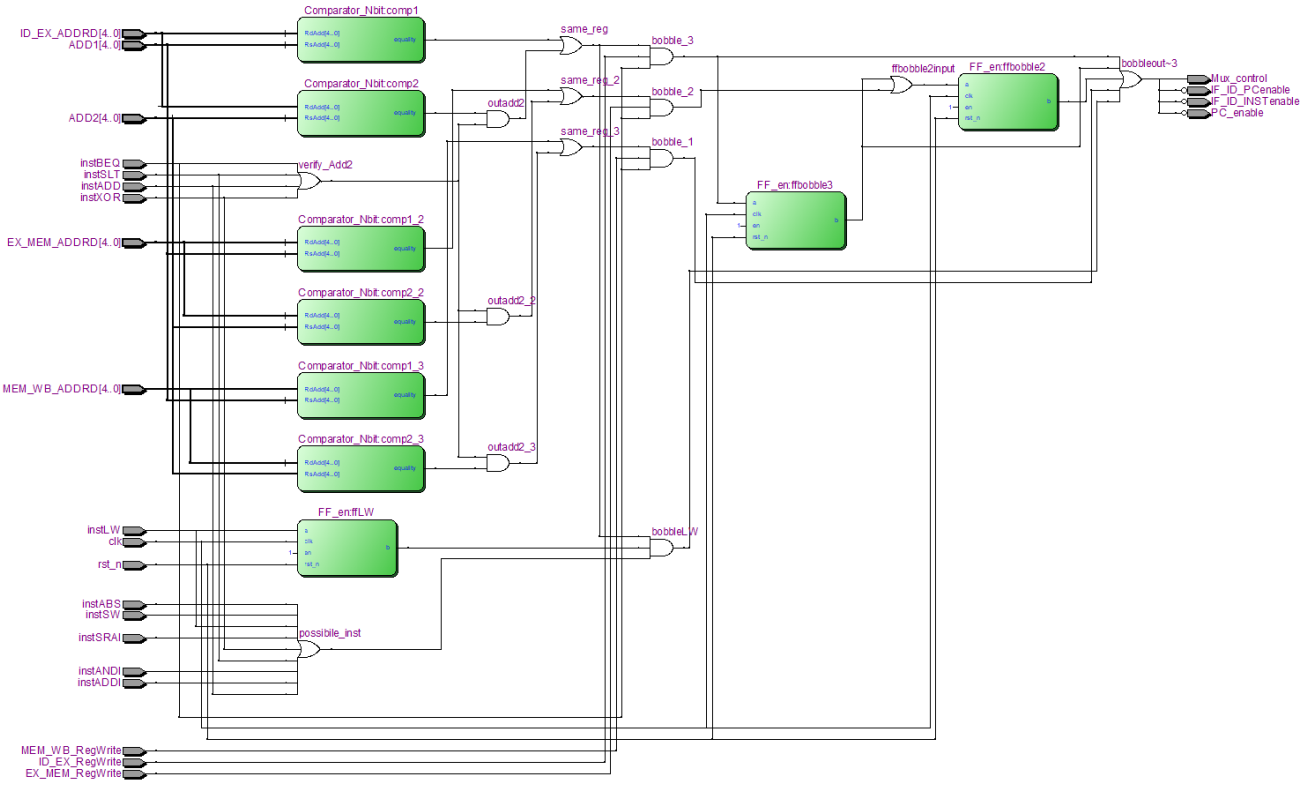


Figure 1.3: RTL viewer of the hazard detector block

### Control block

The Control block is a group of combinational circuits that decodes the instruction fetched, in order to recognize its type and to provide the control signals associated with it, in a way to proper drive the execution unit, the access to the memory and the write back in the registers file.

At this point is necessary to define the control signals generated taking into account the difference between the possible instructions. The control signals are:

- one signals for every instruction, called "<inst>OK" to highligth the instruction recognized.

  To recognize the instruction is used the *opcode* field. In some cases it can be sufficient to detect the instruction, otherwise, for the instruction that share the same opcode, also the fields *func3* and *func7* are necessary, (such as for the *XOR* and the *SLT* that differ only in the *func3* field).

  In order to verify the correct behavior of the type detector combinational circuit a dedicated *Testbench* is written and the circuit is simulated.

  The result can be seen in the figure 1.4:

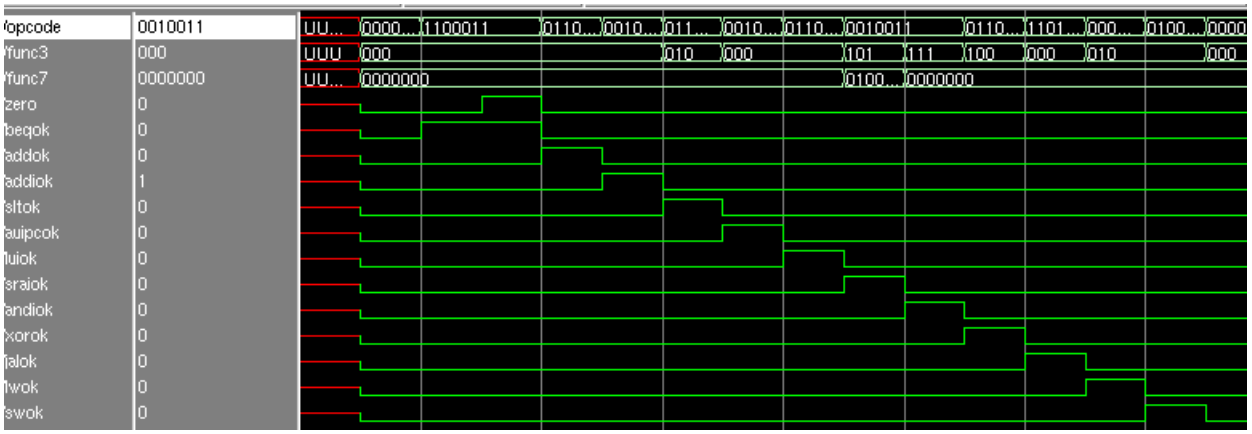


Figure 1.4: Control Unit simulation

- *ALU_SRC* is the selection signal for the mux that choices between *RS2* and the *Immediate* value generated by the Immediat Generator;
- *ALU_OPERATION*, that actives the suitable block of the ALU (Adder/Subtracor, XOR, SRAI or AND block) depending on the arithmetic operation to execute;
- *REG_WRITE* to able the writing in the registers file;
- *WRITE_MEM* and *READ_MEM* signals to manage the interface with the DATA_MEM, when occurs a STORE or a LOAD operation respectively;
- the mux selector to choice between the result of the ALU execution *ALU_RESULT* or the data extracted from the memory *OUT_DATA_MEM*.

It's described an example of the control signals generation in the case of an *ADD* instruction: the control block set at '1' the signal "ADDOK", and select by means of the *ALU_SRC* the *RS2* (no *Immediate* value is used). The *ALU_OPERATION* enables the *ADD* block of the ALU. The memory is not used in this case so the *WRITE_MEM* and *READ_MEM* signals are set to '0'. The write back mechanism is configured to select the *ALU_RESULT* instead of the *OUT_DATA_MEM* to write back the result of the ALU into the registers file, setting the *REG_WRITE* at '1'.

A figure that shows the different controls signals for the different instruction is provided below:

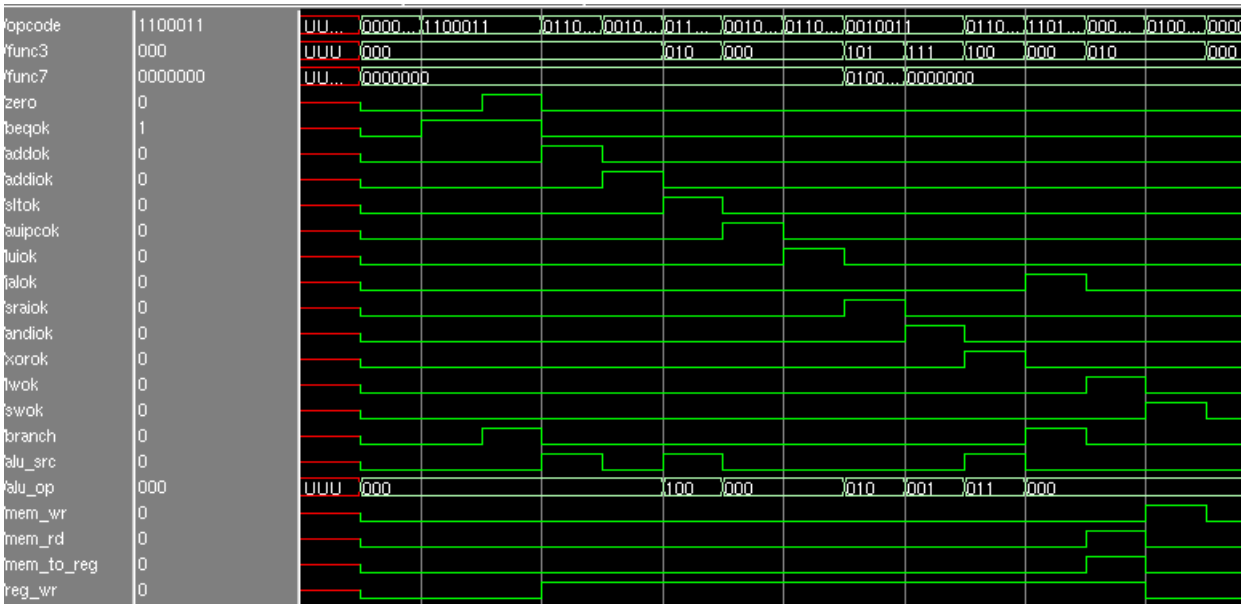


Figure 1.5: Control signals

The control signals are propagated into the processor following the timing dictated by the pipeline timing, so they are send to the pipe registers. A *MUX* is placed in the decode stage in order to ignore them and send to the ID/EX pipeline register all zeros, in case of the *Hazard Detection Unit* detect the needed to insert a bubble.

### Immediate generator

The immediate generator is a block belonging to the decode stage that takes in input the fetched instruction and starting from it recunstructs the immediate value associated with the operation, if there is one. Infact the immediate fields are different for every instruction type, differently distributed in the instruction code, and in some case neighter existent. The output immediate value is expanded to a 32-bits value, to be conform to the size of the data extracts from the registers file. Infact the instruction with an immediate value doesn't use both source register but only once of them.

The unit is divided in three different blocks:

- the **Format Detector block**, that has the aim of understand the format of the instruction in input to proper generate the immediate;
- the **Mux Selector block**, that have the purpose to properly generates the immediate value bit by bit (or by bit groups);
- an **adder/subtractor** used in the case the immediate value generated is the increment/decrement value of the current address to obtained the next address. In particular one input of this arithmetic unit is the current address value contained in the IF/ID register, and the other is the 32 bit immediate value shifted left of 1, to be compliant with the dynamic of the program counter (a

total possible shift of $2^{10}$, $2^5$ left and $2^5$ right in case of a long jump, because of the 64 address bit of the PC).

### Format Detector Block

It exploits the signal generated from the control unit in the same stage (decode stage), that in particular generates one control signal to identify every instruction, setting to '1' the signal associated with the recognized instruction when it's decoded. The Format Detector, taking in input these signals and by means of logic gates, gives in output the format of the instruction.

In particular the format with associated immediate value are reported below:

- R-type: immediate value not used;
- U-type: 20-bits immediate;
- I-type: 12-bits immediate;
- SB-type: 12-bits immediate;
- UJ-type: 20-bits immediate;
- S-type: 12-bits immediate;

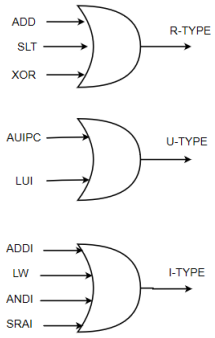


Figure 1.6: Format Detector

In the figure 1.6 it can be seen that R, U and I type, are identified by a logical gates net. Instead the SB, UJ, S format have a direct corrispondence with BEQ, JAL and SW instruction respectively, and so their connection is made by only one wire for each one.

### Mux Selector Block

It's implemented as a bank of multiplexers To implement a smart organization of the bit division the similarity of the different code fields are been evalueted, to properly group the generation.

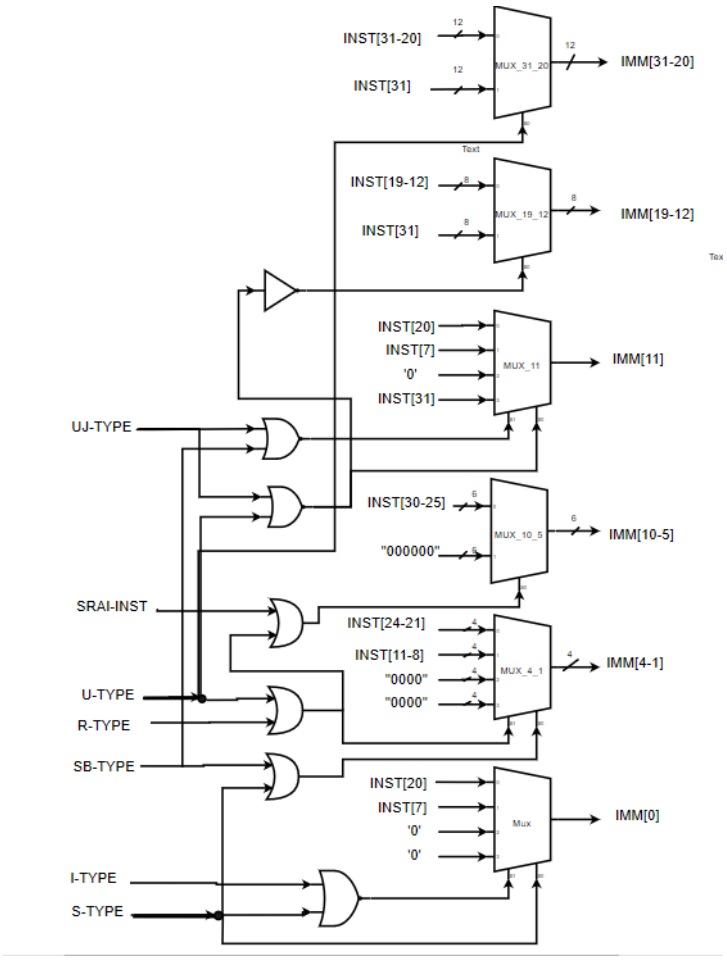


Figure 1.7: Mux Selector

### 1.2.3 Execution stage

The Execution Stage is the core of the architecture, because it's the stage in which the instruction is effectively executed. This is done by an ALU block. The operands are properly selected by two mux driven by the Forwarding Unit, that implements a data bypass to avoid stall when it's possible. A third mux is used at the input of the second Mux of the second operands to permits to select the immediate value when it's requested.

#### Forwarding Unit

The Forwarding Unit is a block of the execution stage used to avoid stalls when a previous value is needed and it's already computed but not yet written in the registers file component. It implements a control to bypass stages and make available the data to the new instruction.

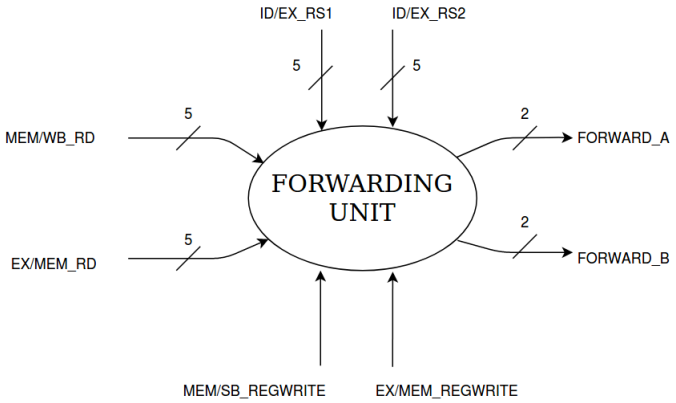


Figure 1.8: Forwarding Unit

The work of this unit is described in detail below.
The unit has as interface signals in input:

- the value of the source registers of the new operation to compute (operation located at the first pipe register, and so at the time of the fetch/decode);
- the destination registers of the two previous instructions (and so from the instructions placed in the registers of execute/memory and in the register of memory/write_back)
- the reg_write signals associated to the two previus instruction.

and as interface signals in output:

- Forwarding_A, that is the selection signal of the mux_A at the input of the ALU. Its aim is choose the first operand of the ALU between the value extracted by the registers file, the value forwarded by the instruction in the ex/mem stage or the value from the mem/wb stage;
- Forwarding_B, that is the selection signal of the mux_B at the input of the ALU. Its aim is choose the second operand of the ALU between the value in output at the mux that is or taken from the registers file or generated by the immediate generator, the value forwarded by the instruction in the ex/mem stage or the value forwarded from the mem/wb stage;

Now it's explained how the unit detects if the forwarding is possible.
The forwarding conditions to take into account are the following ones:

- the destination register to forwarding cannot be the reg0, in which is written the null value. To control this condition a comparator is used, that generates a signal *rd_equal_to_0 ='1'* if the condition is satisfied. This procedure is done for both destination registers of the instructions in the stage *ex/mem* and *mem/wb*.

- the value to forward has to be a value that brings change on registers file and so that writes on it. This condition is verified by means the *reg_write* value ='1' of the two instructions.

- obviously the stall to avoid is generated only if the new decoded instruction uses as one of the source registers or as both of them a registers that is the same of a destination register of the two previous instructions. Proceding with the normal operations without inserting a nop instruction a Read After Write error is generated. In this case the FU detect the equality of the registers and set the variable relative to the stage in which the equality is detect to highlight it.

Once every condition is computed and the proper control signal obtained, using the Karnaugh map in such away to optimize the occupied area, is built a structure of logical gate to detect the really need to forward, and in this case from which stage.

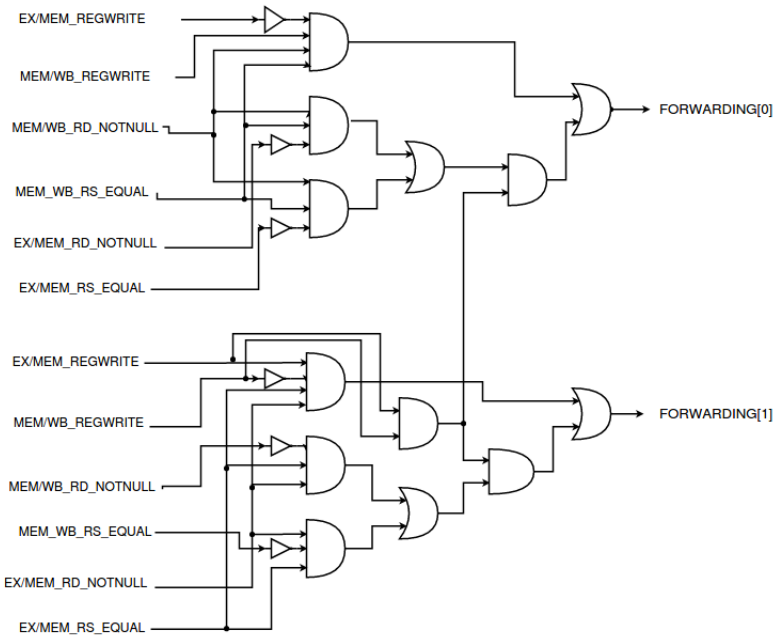


Figure 1.9: Gates Net

The wanted behaviour is tested with a proper testbench that simulates all cases of interest. The different selector signal generated depending on the address of the registers is shown in the Fig.1.10:

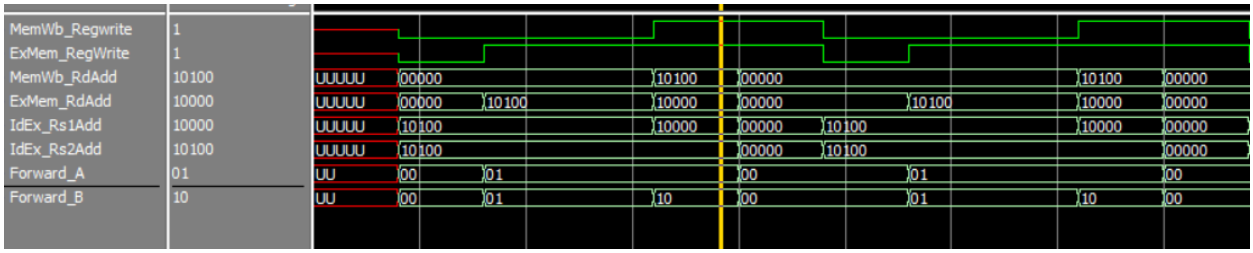


Figure 1.10: Forwarding Unit simulation

### ALU block

The ALU block is placed in the *Execution* Stage, and it performs the arithmetic computation required from the deocded instruction. The operands in input to the ALU are chosen through the selection mechanism implemented with the *Forwarding Unit* reported in detailed in the previous dedicated section.

The architecture of the *ALU* has three main elements:

1. Decoder 2-to-4.
2. Computational Blocks
3. Mux 4-to-1.

The decoder selects the computational blocks necessary to the current instruction. It enables the inputs of the correct block and disables the input operands related to the other computational blocks. Freezing the inputs of the unused blocks avoid useless internal commutations. In this way only one computational block at a time is enabled. The computational blocks are four:

- Adder/Subtractor Block: to perform the instructions *ADD*, *ADDI*, *AUIPC*, *LUI*, *LW*, *SW*, *JAL* and *SLT*;
- And Block: to the *ANDI* instruction;
- Xor Block: to perform the *XOR*;
- Srai Block: to perform the *SRAI*.

The *MUX* is used to select the result from the correct combinational block used, so the selection can be done with the same signals in input to the decoder. A figure that shows the architecture is provided below:

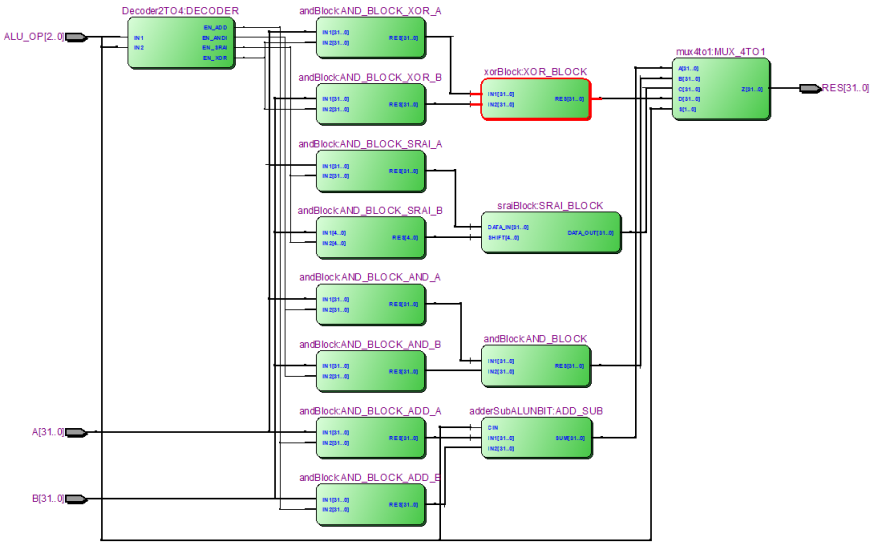


Figure 1.11: RTL-view of the ALU

In order to be sure that all stuff works correctly a *Testbench* is written and the simulation results are shown below:

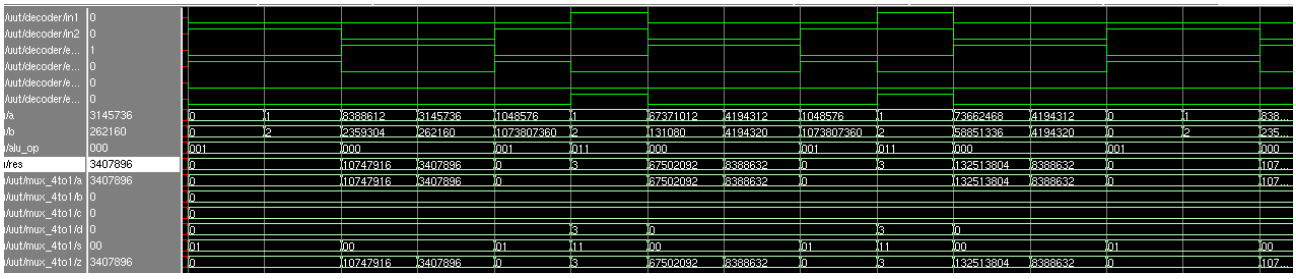


Figure 1.12: ALU simulation

As mentioned before, to obtain a low power optimization the idea is to freeze the input when they are not necessary to decrease the switching activity relative to the input and to the internal unused block of the *ALU*. This idea is implemented with an *ENABLE* signal for the *ALU*. In some case the instruction doesn't need the *ALU* at all, so in order to provide a low power optimization from a dynamic point of view the pipe registers before the ALU are disabled. In this way the ALU keep the previous data, and so no commutations occur inside its computational blocks. Obviously the control signals are not disable in order to follow the correct queue of operations. The Fig.1.13 shows the difference between the architecture of the ALU with and without *ENABLE*.

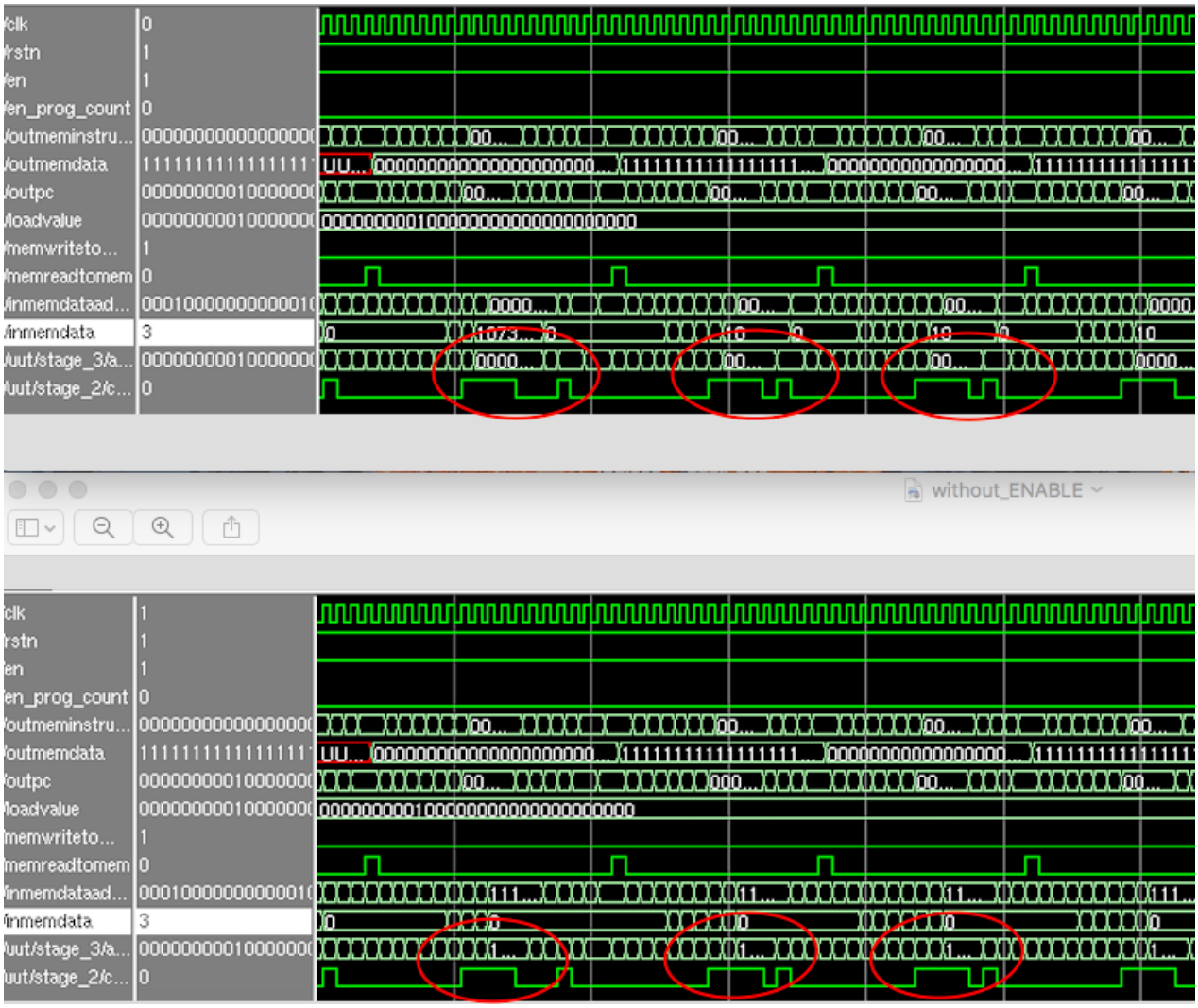


Figure 1.13: ALU switching with and without *ENABLE*

This implementation is thought convenient because also if the ALU is always used except for the *BEQ* instruction, that is completely executed in the second stage, it implements the *if* and the *for* statements, that occur on avarage every five or six instructions in a code. So it's reasonable to think that this choice can provide a substantial savings of the consumption against a negligible increase of the complexity.

### 1.2.4 Memory stage

In this stage there are no functional units. But it's necessary to insert the *Data Memory*. So it's basically composed by interfacing signal to communicate with it. In particular it's necessary to provide the ADDRESS to point and access to the correct data cell. The memory also needed to the control signal WRITE_MEM and READ_MEM to enable the writing and the reading of the data when it's requested.

In output it's necessary to provide both the eventually data extracted from the data memory, and the output result of the ALU that bypass the memory, used when an instruction doesn't involve it.

### 1.2.5 Write Back stage

The Write back stage is directly connected with the Decode stage because of its strictly connection with the registers file. It consists in a mux 2-to-1 with the *ALU_result* and the *Out_Data_MEM* as input. The output is connected as *Data Input* for the registers file. Based on the instruction, the MUX will select the correct data.

## 1.3 Simulation and Synthesis of the architecture

To test the overall architecture is assigned an assembly code struct, in order to verify every possible instruction and the different type of hazarsds.

The code can be seen as an algorithm to detect the minimum value of a vector contained in the data memory.

So the verification of the architecture requires that at the end of the chain of instructions the minimum value of the vector given in the code and uploaded at the beginning in the data memory is recognised.

To use this assembly code is necessary to translate it into machine language. The obtained binary code is been uploaded into the *Instruction Memory*. The memory is thought as a memory which already contains the code to implement and so the test code it's directly written in the vhdl file of this dedicated *Instruction Memory*.

At this point the architecture is simulated on Modelsim. The written *Testbench* drives only three signals: the *CLOCK*, the *RESET* and the *ENABLE*, which the only aim to access to the *Instruction Memory* to fetch the code instructions.

The *Instruction Memory* and the *Data Memory* are not part of the architecture, but they are designed only for the simulation stage. The instuction and the data are fetched on the falling edge of the clock as in the registers file to not interfere with the timing.

To run the code the reset value of the *Program Counter* is not addressed to zero, but to the address in which is contained the first instruction in the *Instruction Memory*.

### 1.3.1 Modelsim Simulation

In the simulation only the most relevant signal are shown in order to highlight that the overall architecture works properly.

The simulation is shown below:

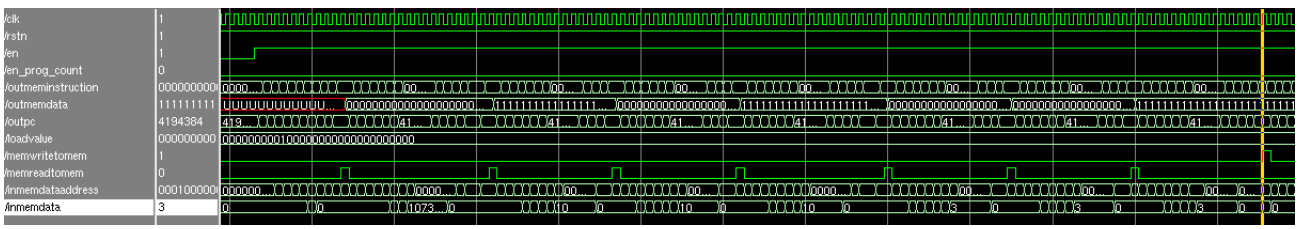


Figure 1.14: Risc-V simulation

It's verified step by step that every instructions work properly. This can also be seen clearly by the final value written into the *Data Memory* that is, as expected the minimum absolute value of the analysed data vector.

This makes possible to assert that the implemented architecture satisfy its required function.

### 1.3.2 Synopsys Synthesis

Verified the correct behaviour, its possible to perform the synthesis of the architecture.

In order to do that it's necessary import the *Cell Lybrary* library with the insertion of the file *.synopsys_dc.setup* in the synthesis directory. It also configures the *Design Vision* workspace.

The procedure of the synthesis has as first step the analysys of all the VHDL files belonged to the project and then the application of the constraints. The main constraints are the clock period, its uncertainty and the maximum delay for both input and output signals. For the compile phase is chosen to use the command *compile_ultra*. This keyword enables an optimized synthesis in order to easily achieve the required constraints. The *Clock Period* is set to 2.0 ns so the *Clock Frequency* is 500 MHz. The timing and the area reports of the synthesis are shown in Fig.1.15 and in Fig.1.16 respectively.

```
clock MY_CLK (rise edge)                                  2.00       2.00
clock network delay (ideal)                               0.00       2.00
clock uncertainty                                        -0.07       1.93
Stage_1/IF_ID_instruction/b_reg[0]/CK (DFFR_X1)           0.00       1.93 r
library setup time                                       -0.04       1.89
data required time                                                   1.89
--------------------------------------------------------------------------
data required time                                                   1.89
data arrival time                                                   -1.88
--------------------------------------------------------------------------
slack (MET)                                                          0.01
```

Figure 1.15: Report Timing

```
Number of ports:                          195
Number of nets:                          6774
Number of cells:                         6496
Number of combinational cells:           5112
Number of sequential cells:              1382
Number of macros/black boxes:               0
Number of buf/inv:                        444
Number of references:                      41

Combinational area:               6557.964030
Buf/Inv area:                      272.118000
Noncombinational area:            7385.756227
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                 13943.720257
Total area:                 undefined
1
```

Figure 1.16: Report Area

In order to perform the power estimation of the architecture, the NETLIST is

converted in Verilog and it's generated the *sdf files* that contains the informations about the delays of the netlist.

It's so possible to launch the simulation with *Modelsim*.

Before starting it the file *Value-Charge-Dump* (*vcd file)*, which containes the informations about the interconnections, is generated and saved.

At this point it's possible to run the simulation. The results are compliant with the VHDL design. So the synopsis shell is launched, and the *vcd file* is converted into *saif file* to make it readable by Synopsis. The power estimation is performed and the resultant report is shown in the Fig.1.17.

```
Design        Wire Load Model            Library
------------------------------------------------
risc_v                5K_hvratio_1_1    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  = 704.0732 uW   (72%)
  Net Switching Power  = 280.6323 uW   (28%)
                         ---------
Total Dynamic Power    = 984.7055 uW  (100%)

Cell Leakage Power     = 236.8207 uW

                 Internal         Switching           Leakage            Total                          Cell
Power Group      Power            Power               Power              Power   (   %    )  Attrs    Count
-----------------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)              0
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)              0
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)              0
clock_network      2.8771         113.8630            459.3028           117.1994  (   9.59%)              1
register         630.0798          15.2404          1.1075e+05           756.0734  (  61.90%)           1382
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)              0
combinational     71.1144         151.5287          1.2561e+05           348.2519  (  28.51%)           5111
-----------------------------------------------------------------------------------------------------------
Total            704.0712 uW      280.6321 uW      2.3682e+05 nW      1.2215e+03 uW
1
```

Figure 1.17: Report Power

### 1.3.3 Innovus Place and Route

The Cadence Innovus tool is used to perform the Place and Route. Firstly the design is set as the Top Level Design and the Timing Library setup is done. So it's possible to start with the structuring floorplanning, the area of the cell ensemble is determined and the tool is able to know how many rows are needed for the design. Nextly the PowerRings are added on the layout and the PowerRouting is performed. At this point it's possible to place the standard cells, an optimization related with the timing constraints is done and the physical cells are placed.

At this point a routing algorithm is executed at a very low level in order to make possible the final connection with all the metal layers.

The final layout is reported below:

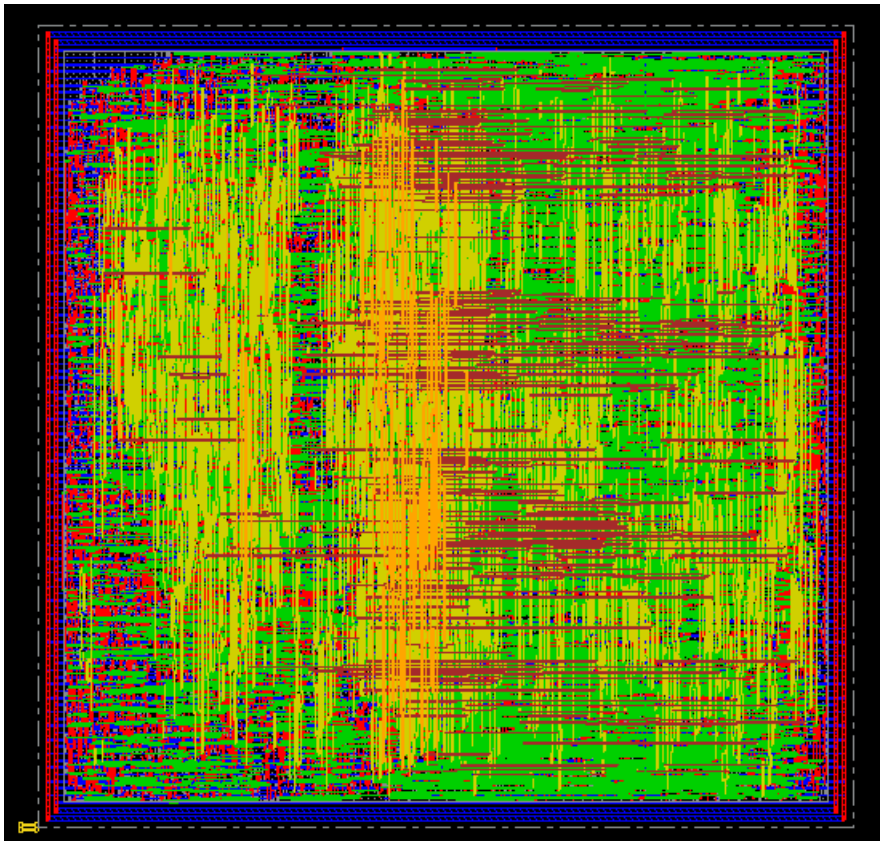

Figure 1.18: PostNano Route

Finally another Timing Optimization is applied to be comply with the constraint. So at this point is needed to extract the parasitic terms of the design, in particular the resistances and the capacitances related to the metal wires.

At the end is done the verification of the connectivity and the geometry. The final area of the circuit is reported in the Fig.1.19

```
Gate area 0.7980 um^2
Level 0 Module risc_v
Gates =  17746
Cells =  6669
Area  =  14161.8 um^2
```

Figure 1.19: Final Area

As a final step is required to write the sdf file in order to make possible the Power estimation. In order to do this is launched Modelsim, so a simulation is performed for the design and the vcd file is written. In this way Innovus can report the switching activity of the circuit. The final results are shown below:

```
Design: risc_v

Liberty Libraries used:
        MyAnView: /software/dk/nangate45/liberty/NangateOpenCellLibrary_typical_ecsm_nowlm.lib

Power Domain used:

Power View : MyAnView

User-Defined Activity : N.A.

Switching Activity File used:
        ../vcd/risc_v_inn.vcd
                Vcd Window used(Start Time, Stop Time):(-276688, -276688)
              Vcd Scale Factor: 1
              Design annotation coverage: 0/6947 = 0%

Hierarchical Global Activity: N.A.

Global Activity: N.A.

Sequential Element Activity: N.A.

Primary Input Activity: 0.200000

Default icg ratio: N.A.

Global Comb ClockGate Ratio: N.A.

Power Units = 1mW

Time Units = 1e-09 secs

Temperature = 25

Total cells : 6669
Internal Power : 5.695
Switching Power : 3.336
Total Power : 9.309
Leakage Power : 0.2783
Total Capacitance : 3.85e-11 F
```

Figure 1.20: Report Power

## 1.4 Addition of a new instruction to the ISA

One the operation of the RISC-V is ascertained, a new operation is required to be add to its instruction set. In particular it's about the computation of the absolute value.

This addition requires a set of operation:

- the choice of the instruction encoding: it's chosen the encoding that in the standard RISC-V is associated with the instruction *SUB*. This because the needed is an instruction that not exploits the immediate value, and gives a field for the destination register. The second source register field is not used also is available (set to the fixed value x0).

- the adaptation of the ALU block to implement the absolute value operation. In particular this is done with a modification of the XOR block that becomes able to do the two different operation, in such a way to keep using of the 4 outputs decoder and avoid HW addition. To control the right use of XOR block, two mux are necessary, one at the input and one at the output;

- the addition on the Control Block of the verification of the ABS instruction, to generates proper controls signal, as the ALU source and ALU operation;

- the modification of the Hazard Detection Unit in order to deal with the ABS instruction as an other computational instruction that may need some bubbles before the execution of it.
- the modification of the Forwarding Unit in order to bypass the source register of the ABS instruction if modified by previous instruction and not yet written on the registers file, in such a way to avoid as possible the isertion of bubbles.
- the written of a proper testing code, in particular it's taken the given code to test the initial processor, and its modified substituting the series of the 4 different available instructions that implement indirectly the absolute value computation, with the single instruction dedicated implemented. From this procedure it can been clearer that implement the absolute value as a dedicated instruction is improved in terms of timing performance, because the number of instructions to compute it it's restricted from 4 to only one (less clock cycles and less possible data hazards)
- the update of the address of instructions in the instruction memory to correct redirect the destination of JAL and BEQ.

## 1.5 Simulation and Synthesis of the new architecture

After the steps previously explained the architecture can perform the new instruction.

So the new test code is written into the *Instruction Memory* and the simulation of the architecture is done with *Modelsim*.

### 1.5.1 Modelsim Simulation

The simulation result is printed below:

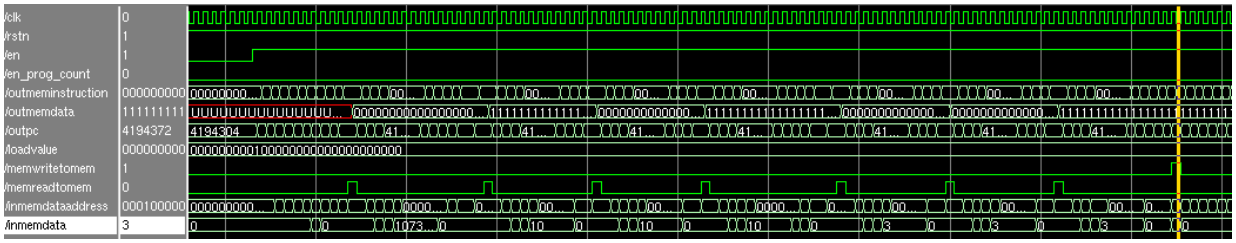


Figure 1.21: Risc_V with ABS simulation

Also in this case it's shown in the Fig.1.21 that the final value written into the memory is correct, so the architecture is confirm as properly implemented.

It's can also be seen the performance improvement: to perform the same task as before, thanks to the shorter code to execute, the total number of clock cycles are 135, against the previous 155, showing a decrease of the latency of the **13%**.

### 1.5.2 Synopsis Synthesis

As previously done is possible to procede with the synthesis of the circuit. The constraints related to this new architecture are the same as before, with a consequent *Clock Frequency* equal to 500 MHz. Also the *Power Estimation* is done at this step. The three resultant report about the area, timing and power consumption are visionable in the Fig.1.22, Fig1.23 and Fig1.24 respectively.

```
Number of ports:                          195
Number of nets:                          6764
Number of cells:                         6363
Number of combinational cells:           4979
Number of sequential cells:              1382
Number of macros/black boxes:               0
Number of buf/inv:                        450
Number of references:                      42

Combinational area:               6385.596019
Buf/Inv area:                      268.926000
Noncombinational area:            7385.490226
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                 13771.086245
Total area:                 undefined
1
```

Figure 1.22: RiscV ABS Report Ares

```
clock MY_CLK (rise edge)                                  2.00       2.00
clock network delay (ideal)                               0.00       2.00
clock uncertainty                                        -0.07       1.93
Stage_3/ExMemRegAluRes/b_reg[28]/CK (DFFRS_X1)            0.00       1.93 r
library setup time                                       -0.04       1.89
data required time                                                   1.89
--------------------------------------------------------------------------
data required time                                                   1.89
data arrival time                                                   -1.89
--------------------------------------------------------------------------
slack (MET)                                                          0.00
```

Figure 1.23: RiscV ABS Report timing

```
Design        Wire Load Model            Library
------------------------------------------------
risc_v_abs             5K_hvratio_1_1    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  = 701.9758 uW   (71%)
  Net Switching Power  = 292.9227 uW   (29%)
                         ---------
Total Dynamic Power    = 994.8985 uW  (100%)

Cell Leakage Power     = 233.9847 uW

                 Internal         Switching           Leakage            Total                          Cell
Power Group      Power            Power               Power              Power   (   %    )  Attrs  Count
-----------------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)            0
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)            0
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)            0
clock_network      2.8771         113.8630            459.3028           117.1994  (   9.54%)            1
register         630.6490          15.5539          1.1076e+05           756.9645  (  61.60%)         1382
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)            0
combinational     68.4486         163.5054          1.2277e+05           354.7196  (  28.87%)         4978
-----------------------------------------------------------------------------------------------------------
Total            701.9747 uW      292.9223 uW      2.3398e+05 nW      1.2289e+03 uW
1
```

Figure 1.24: RiscV ABS Report Power

### 1.5.3 Innovus Place and Route

The steps done to execute the phase of place and route are exactly the same as the previous implementation. The final layout, the gate count and the power estimation are illustrated in the following figures:

```
Gate area 0.7980 um^2
Level 0 Module risc_v_abs
Gates =   17602
Cells =   6520
Area  =   14046.4 um^2
```

Figure 1.25: RiscV ABS Area Innovus

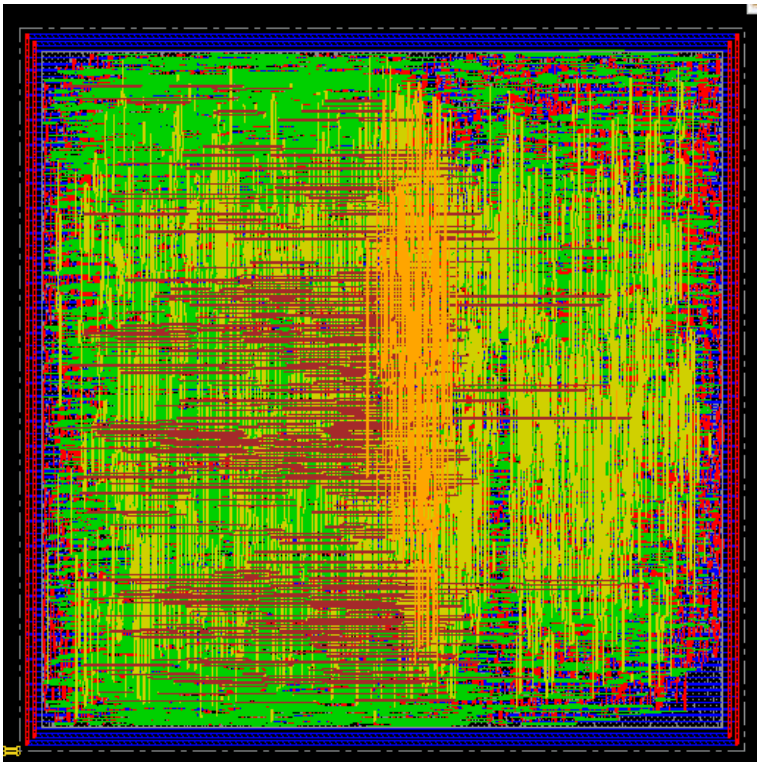

Figure 1.26: RiscABS PostNano Route

```
Design: risc_v_abs

Liberty Libraries used:
        MyAnView: /software/dk/nangate45/liberty/NangateOpenCellLibrary_typical_ecsm_nowlm.lib

Power Domain used:

Power View : MyAnView

User-Defined Activity : N.A.

Switching Activity File used:
        ../vcd/risc_v_abs_inn.vcd
                Vcd Window used(Start Time, Stop Time):(1586.27, 1586.27)
              Vcd Scale Factor: 1
              Design annotation coverage: 0/6921 = 0%

Hierarchical Global Activity: N.A.

Global Activity: N.A.

Sequential Element Activity: N.A.

Primary Input Activity: 0.200000

Default icg ratio: N.A.

Global Comb ClockGate Ratio: N.A.

Power Units = 1mW

Time Units = 1e-09 secs

Temperature = 25

Total cells : 6520
Internal Power : 5.779
Switching Power : 3.15
Total Power : 9.205
Leakage Power : 0.2755
Total Capacitance : 3.752e-11 F
```

Figure 1.27: RiscV ABS Power Innovus

## 1.6 Possible Improvements

At the end of this report it's considered important to list several ideas evaluate during the design. These solutions are shown but not implemented because they were considered not optimal in the specific case of the implemented architecture, but they could be useful in the case of a more general purpose processor or with a different Instruction Set.

1. *BRANCH PREDICTOR*: To manage the branch instuction in the processor, a *branch predictor* can be implemented to fetch dynamically the instruction before the computation of the branch condition. This is done in order to minimize the *pipe stall* resulting from time needed for the computation of the condition, especially in the case the operands are affected by data dependencies. The *branch predictor* can be implemented in different ways. For obtain an high general purpose level can be used different implementation of branch predictors for different dedicated application and a metapredictor to select time at a time the more reliable predictor depending on the wanted application.

   In the architecture developed, since the decision is taken in the second stage is not useful a dynamic branch prediction. A static branch prediction is implemented, infact the fetch after the branch instruction is done as the branch condition is false (sequential way) and only at the moment the condition is evaluated as true the previous wrongly fetched instruction is flushed and the fetch redone for the branch address. In the case of no branch there are no timing penalty. The case of penalty is so not substantially to decide to increase the complexity of the architecture implementing a predictor.

2. CACHE MEMORIES FOR DATA AND INSTRUCTIONS: in order to speed up the architecture from the memory response point of view is possible to use *cache memory* instead of RAM. Providing the *Mapping algorithm*, the *Replacement algorithm* and the *Write strategy* to handle the *Cache memory* is possible to change the critical path from the *Memory* stage to the *ALU* stage, and in the *execution* stage is possible to better exploit some improvements.

   In the treated case the structures of the memories is external to the architecture of the processor, they are only implemented to the test phase.

3. *REDUCE LATENCY*: For the *load* and the *store* instruction is possible to see that the *Data memory* cover an entire stage because of the need to determine the address memory through the ALU. But, if we think to determine the address memory with the use of the *adder-subtractor* of the second stage, is possible to make the memory access one stage before. The same goes for the *Write back stage*, so the *Instruction latency* would becomes of four clock cycles. This leads to a simpler *Forwarding unit*, *Hazard detection unit* and *Control unit* that have to deal with one stage less. So also the complexity would decrease.

In this way of thinking we are going to use only one between the $ALU$ and the *Data memory* for all the instructions, so it's possible to freeze one of them when is not used as we have done for our specific RISC-V in order to have less switching activity.

This is kept as an idea, but it's not implemented to be faithful to the traditional 5 stages of the Risc-V.

## 1.7 Conclusion

In the following table are collected all the results for the two different implemented architecture, in order to make the comparison between them more evident.

| Final comparison | | Power | | | | AREA |
|---|---|---|---|---|---|---|
| | | Internal | Switching | Total dyn. | Leakage | |
| Synopsys | Init Arch. | 704.1 $\mu W$ | 280.6 $\mu W$ | 981.7 $\mu W$ | 236.0 $nW$ | 13943 $\mu m^2$ |
| | Arch. abs | 702.0 $\mu W$ | 292.9 $\mu W$ | 994.9 $\mu W$ | 234.0 $nW$ | 13771 $\mu m^2$ |
| Innovus | Init Arch. | 5.70 $mW$ | 3.34 $mW$ | 9.31 $mW$ | 278 $\mu W$ | 14162 $\mu m^2$ |
| | Arch. abs | 5.78 $mW$ | 3.15 $mW$ | 9.21 $mW$ | 276 $\mu W$ | 14046 $\mu m^2$ |

The results are very similar for the two implemented architectures . These results can be attributable by the fact that, because the simulation are based on heuristic algorithms, the little variation of the result from simulation to simulation, makes the differences not evident. But in principle, starting from the consideration that the power analysis has to be evaluated after the place and route to be consistent (Innovus is related to Synopsys by a factor of about 8), and so based on the results obtained by Innovus, it can be seen a little improvement with the RISC-V ABS, due to the fact that the smaller number of instructions to execute, and so the consequently lower number of commutations.

In conclusion, it can be said that the addition of one instruction to the ISA doesn't involve in a relevant difference in terms of area and performance of the architecture. But obviously considering a further and significant enlargement of the ISA, the instructions addition, also if as in the treated case reduces the latency thanks to the possibility to compress the code, would give a significant increasing in terms of area for HW addition, and in terms of power due from the additive control signals.