Politecnico di Torino

III Facoltà di Ingegneria

Exercises and Homeworks for the course Integrated Systems Architecture

LAB 3: Design of a RISC-V-lite processor

Master degree in Electronic Engineering

Group number: 37

Costantino Taranto - 274492
Pasquale Santoro - 278329
Alberto Aimaro - 253196

February 21, 2021

Repository Link: `https://github.com/isa037/lab3`

Contents

1 Introduction **1**

1.1 About the Risc-V architecture . . . 1

1.2 Processing stages . . . 2

1.2.1 Instruction Fetch (IF) . . . 2

1.2.2 Instruction Decode (ID) . . . 2

1.2.3 Execution (EX) . . . 3

1.2.4 Memory Access (MEM) . . . 3

1.2.5 Writeback (WB) . . . 4

1.3 Lite ISA . . . 4

2 Risc-V: General implementation **6**

2.1 HW description . . . 6

2.1.1 Register file . . . 6

2.1.2 Alu . . . 7

2.1.3 Control Unit . . . 8

2.1.4 Branch value provider . . . 10

2.1.5 Program counter manager . . . 10

2.1.6 Write Back selector . . . 11

2.1.7 Data hazard management . . . 11

2.2 System Simulation . . . 15

2.2.1 Single Instructions Test . . . 15

2.2.2 Test with the assembly program . . . 21

2.3 Processor Logic Synthesis . . . 24

2.3.1 Elaboration Results . . . 26

2.3.2 Maximum clock frequency evaluation . . . 26

2.3.3 Area evaluation . . . 26

3 Risc-V: Advanced Architecture to face Control Hazards **27**

3.1 HW modification: Branch Taken . . . 27

3.1.1 Hazard Detection Unit for the branch delay reduction . . . 28

3.2 SW modification: Instructions Rescheduling . . . 29

3.3 System simulation . . . 29

3.4 Logic Synthesis . . . 31

3.5 Synthesized Netlist extraction and testing . . . 31

3.6 Place & Route . . . 32

3.6.1 Parasitics extraction . . . 33

3.6.2 Verification of the setup and hold requirements . . . 34

3.6.3 Connectivity and Design rules verification and Geometry . . . 34

3.6.4 Modelsim simulation and swithing activity recording 34
3.7 Power consumption estimation 35

4 Risc-V with ABS function 36
4.1 Hardware modifications 36
4.2 System Simulation 37
4.3 Logic Synthesis 37
4.4 Place&Route 39
4.4.1 Parasitics extraction 39
4.4.2 Verification of the setup and hold requirements 40
4.4.3 Connectivity and Design rules verification and Geometry 40
4.4.4 Modelsim simulation and switching activity recording 40
4.5 Power consumption estimation 40
4.6 RISC-V with ABS function, without Branch Handling 41

5 Comparison between different architectures 42
5.1 Speed and Area 42
5.2 Power consumption 43

6 Appendix 44
6.1 minv-rv.s 44
6.2 minv-rv-NOPs.s 45
6.3 minv-rv-rescheduling.s 46
6.4 minv-rv-abs-rescheduling.s 47
6.5 minv-rv-abs-NOPs.s 48
6.6 elaborate_results.txt 49

CHAPTER 1

Introduction

1.1 About the Risc-V architecture

The RISC-V is an open-ISA Reduced Instruction Set Computer developed at UC Berkeley, now managed by the RISC-V Foundation. It is provided with a register file made up by 32 registers of various width, ranging from 32 to 64 bits.
The aim of the laboratory is to design a *lite* version of the processor, named **RISC-V-lite**, with 5 pipeline stages and 32-bit width registers.
The RISC-V-lite block diagram is shown in figure 1.1.

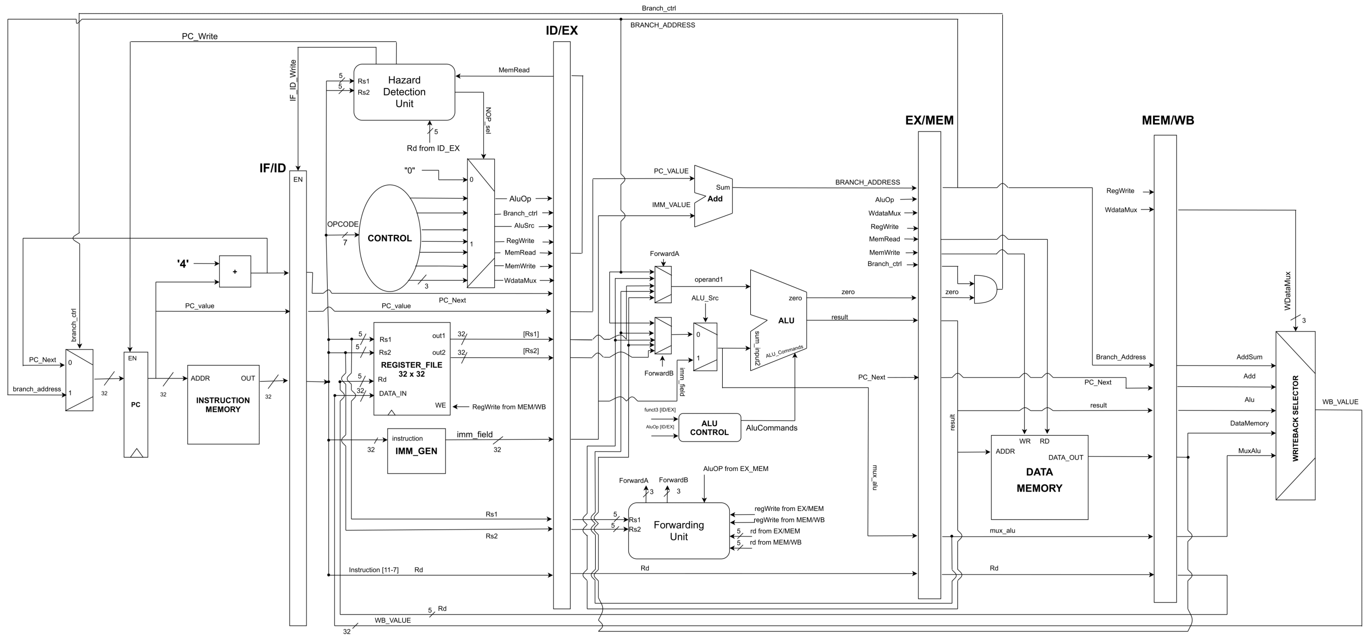


Figure 1.1: Datapath of the *Risc-V-lite*.

1.2 Processing stages

The 5 pipeline stages divide the instruction flow in 5 processing steps, each of them with a specific aim.

1.2.1 Instruction Fetch (IF)

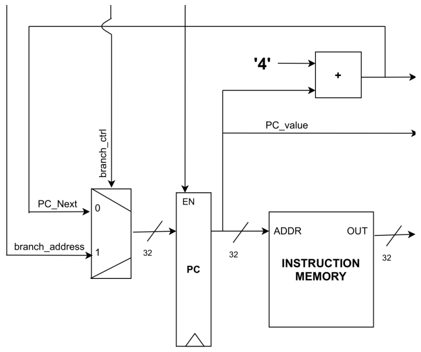


Figure 1.2: The **IF** stage.

It has the aim of loading the correct value for the next instruction in the *Program Counter*. A selection signal (*branch_ctrl*) allows to select between the sequential address (*PC_next*) and the jump address (*branch_address*). The PC selects the correct instruction from the Instruction Memory (IM), which is sampled by the first pipe register (IF/ID).

1.2.2 Instruction Decode (ID)

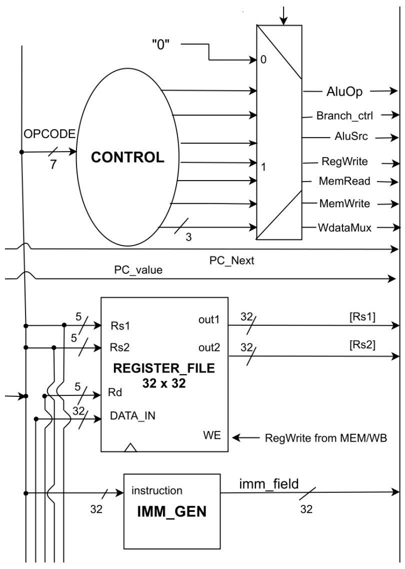


Figure 1.3: Part of the **ID** stage.

In this stage the instruction coming from the IM is used by the Control unit to select the correct control signals, by the Register File (RF) to read the correct value of the source registers and by the

IMM_GEN unit to generate the Immediate field depending on the instruction to be executed.

1.2.3 Execution (EX)

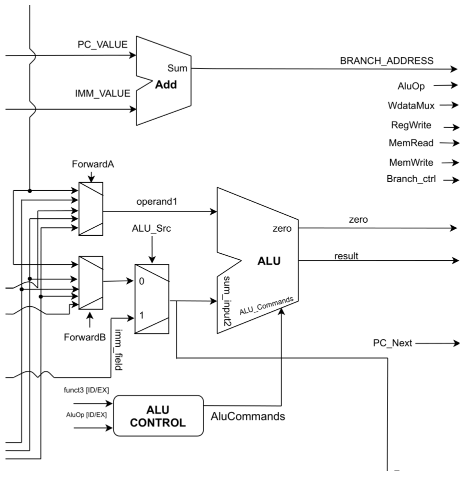


Figure 1.4: Part of the **EX** stage.

The main purpose of the EX stage is to Execute the instructions on the correct operands and to compute the jump address, if needed.

1.2.4 Memory Access (MEM)

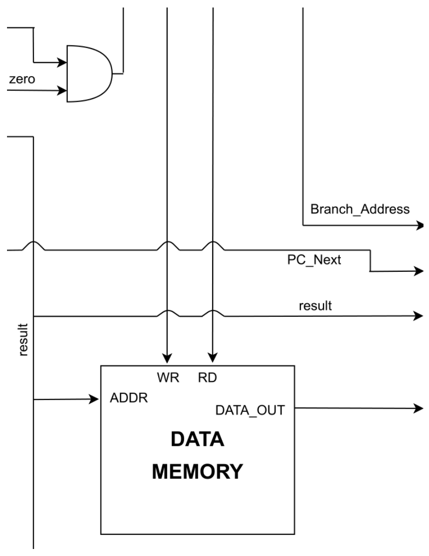


Figure 1.5: The **MEM** stage.

In the MEM stage the memory access can happen, the output of the Data Memory (DM) is sampled by the following register (MEM/WB). Here, the *branch_ctrl* signal is also computed, depending on the ALU result and the control signal coming from the Control Unit.

1.2.5 Writeback (WB)

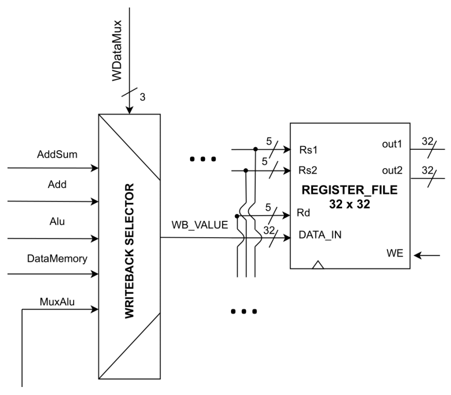


Figure 1.6: The **WB** stage.

The Writeback stage allows to save the the appropriate data on the destination register, in the Register File.
Notice that the connection between the MUX and the RF is highlited but it does not represent the actual reciprocal position of the components in the datapath.

1.3 Lite ISA

The word *lite* indicates that the processor can only execute a subset of the whole ISA of the RISC-V. The instructions are organized in six classes:

- R (Register)
- I (Immediate)
- S (Store)
- SB (Conditional Branch)
- U (Upper Immediate)
- UJ (Unconditional Jump)

Each particular class has an instruction format associated to it. In table 1.1 all the instruction formats on the 32-bit instruction word are reported.

31-25	24-20	19-15	14-12	12-7	6-0	Class
funct7	rs2	rs1	funct3	rd	opcode	R (Register)
imm[11:0]		rs1	funct3	rd	opcode	I (Immediate)
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S (Store)
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	SB (Conditional Branch)
imm[31:12]				rd	opcode	U (Upper Immediate)
imm[20|10:1|11|19:12]				rd	opcode	UJ (Unconditional Jump)

Table 1.1: Instruction Formats in the RISC-V

Particular attention must be given to the meaning of each field.

- **rs1,rs2,rd**

  The addresses of the two source registers and of the destination register, respectively (5 bits each).

- **opcode, funct7, funct3**

  The codes which identify the particular operation, used by the control unit and other logic blocks to execute the correct operation stored in the IM (opcode: 7 bits, funct7: 7 bits, funct3: 3 bits).

- **imm**

  The immediate field, used for immediate operations (variable length depending on the instruction class).

The supported instructions are:

- **ADD**: Performs the addition between the contents stored in rs1 and rs2. The result is sotred in rd. (R-type)
- **ADDI**: Performs an addition between the content of rs1 with the immediate field (sign-extended on 32 bits). The result is stored in rd. (I-type)
- **AUIPC**: Performs an addition between the sign-extended immediate field and the current value of the PC. The result is stored in rd. (U-type)
- **LUI**: Stores the sign-extended immediate field in rd. (U-type)
- **BEQ**: Loads the branch address in the PC if the contents in rs1 and rs2 are equal. The branch address is computed from the immediate field and the current PC value. (SB-type)
- **LW**: Loads in rd a word in the data memory, at the address given by the sum of rs1 and the (sign-extended) immediate field. (I-type)
- **SRAI**: Shifts arithmetically the content of rs1 by an amount indicated by the immediate field. The result is stored in rd. (I-type)
- **ANDI**: Performs bitwise AND, between the sign-extended immediate field and rs1. The result is placed in rd. (I-type)
- **XOR**: The bitwise logical XOR between rs1 and rs2 is performed. The result is placed in rd. (R-type)
- **SLT** Compare the contents of rs1 and rs2 (considered as signed numbers). If the first is greater than the second, write "1" in rd, "0" otherwise. (R-type)
- **JAL**: Stores the next value of the PC in rd, then load in the PC the jump address. The jump address is computed from the immediate field and the current value of the PC. (UJ-type).
- **SW**: Stores in the Data Memory the content of rs2. The address of the DM for the store is computed from rs1 and the immediate field. (S-type)

CHAPTER 2

Risc-V: General implementation

In the following sections each component of the processor is described in detail. Information are given about their behavior and implementation.
Note: The contents discussed in this chapter are referred to the branch *risc-V-No_branch_handle* of the github repository.

2.1 HW description

2.1.1 Register file

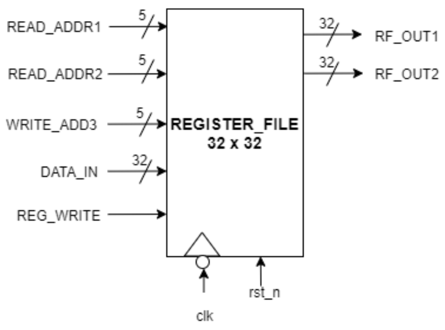


Figure 2.1: The Register File

The register file is a data structure used to manage the internal registers of the processor. It is made up by 32 register that store 32 bit data.

It accepts as INPUT 3 addresses (length: 5-bit), 2 used for read operation, 1 used for write operation. It has 2 outputs that provide the 32 bits operands stored in memory.

READ operation is asynchronous. The structure provide directly to the outputs the operands stored in input addresses.

WRITE operation is synchronous and happens on the Clock signal's falling edge. This property of the RF allows to perform writing and reading operations in the same clock cycle.
In figure 2.2, an example is shown: suppose that an instruction (named A) writes the result of an operation in a certain clock cycle (WB of instr. A, writing operation pointed by the blue arrow). The writing on the RF in the same clock cycle allows the ID cycle of instruction B to read the correct value from the same register, if needed (read operation pointed by the green arrow).

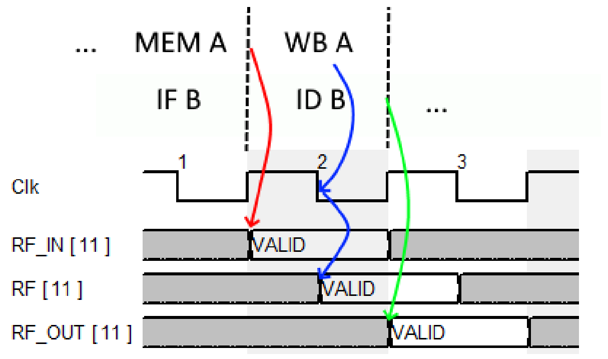


Figure 2.2: Data hazard solved by RF Transparency.

In the example, RF_IN[11] is the data at the input port of register R11; RF[11] is the content of register R11; RF_OUT[11] is the output of the ID/EX register (result of the ID opreation).

2.1.2 Alu

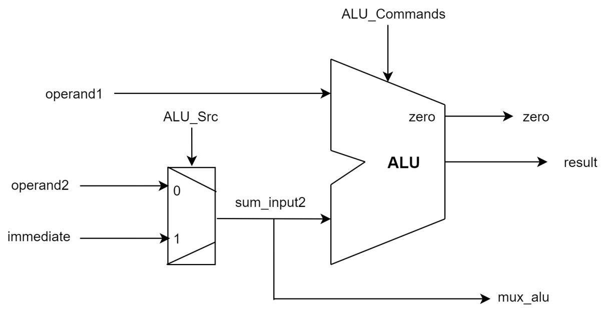


Figure 2.3: The Arithmetic and Logic Unit

The implemented Alu has as **input** the *2 operands* coming from the register file memory, the *immediate operand* required to execute immediate instructions and 2 control signals: *AluCommand* and *AluSrc*. The control signals are used to manage alu operations(AluCommand) and the input selection (AluSrc).

If AluSrc is '1', immediate operand is used to perform alu operations. The implemented AluCommands are:

- SUM: perform the sum of the two operands providing the result at the output;
- CONFRONTO_IF_EQUAL: verify if the two selected operands are equal and set the variable 'zero' to 1;
- SHIFT: perform the shift of the input operand 1 of the amount defined in operand 2;
- CONTRONTO_SLT: provide $(1)_{dec}$ as 32-bit output result if operand 1 $<$ operand 2, 0 otherwise. In this case the inputs are treated as Signed binary numbers;
- AND: provide as output (32 bit result) the bitwise AND of the input operands;
- XOR: provide as output (32 bit result) the bitwise XOR of the input operands;
- NOP: does not perform any operation.

The output values are a 32-bit result of the operations and a signal "zero" used to indicate if the input operands are equal or not.

2.1.3 Control Unit

The control unit is made up by 2 sections:

- CONTROL SECTION
- ALU CONTROL SECTION

The CONTROL SECTION is the part which generates the control signals for the whole datapath. The signals generation has been performed analysing the specification requirements of the ISA. In particular the signals generation depends only from the OP CODE field of the instruction.

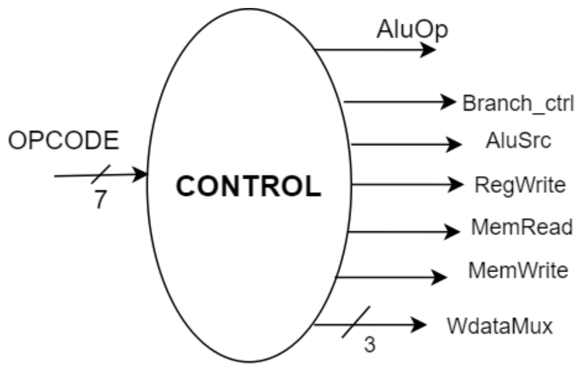


Figure 2.4: The *Control* unit

From the RISC V instruction specifications table 2.1has been pointed out.

INPUT	*OUTPUT*						
OP Code	**BRANCH**	**REGWRITE**	**ALUSrc**	**ALUOP**	**MEMWRITE**	**MemRead**	**WDataMux**
0110111	0	1	1	LUI	0	0	ALU
0010111	0	1	X	AUIPC	0	0	AddSum
1101111	1	1	1 (X)	JAL	0	0	Add
1100011	1	0	0	BRANCH	0	0	X
0000011	0	1	1	LOAD	0	1	Data Memory
0100011	0	0	1	STORE	1	0	X
0010011	0	1	1	OP_IMM	0	0	ALU
0110011	0	1	0	OP	0	0	ALU

Table 2.1: CU control signals depending on the OPCODE

This table reports for every instruction the HW operation that datapath has to perform:

- BRANCH: selection signal for jump. '1' value identify branch instructions
- REGWRITE: signal required to write a data in the register file. '1' value to perform the write
- ALUSrc: signal required to use immediate value as ALU operand. '1' value select the immediate value, '0' to use the value coming from registers
- ALUOP: operation to perform in ALU. The possible operations are:
  - LUI
  - AUIPC

- JAL
- BRANCH
- LOAD
- STORE
- OP_IMM
- OP

- MEMWRITE: signal required to write a data in data memory. '1' value to perform the write
- MemRead: signal required to read a data from data memory. '1' value to perform the write
- WDataMux: signal to select the correct write back data (see section "Write Back Selector")

The ALU CONTROL SECTION is the section that generates the control signals for the ALU.

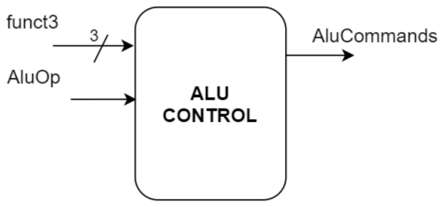


Figure 2.5: The Alu Controller

It takes as input the value of ALUOP and the values of funct3 from the input instruction. The 2 inputs are required because some instructions have the same opcode but different functions so it is required additional logic to select proper alu operations.

The selection is performed following the summary table:

ISA INSTRUCTION	*INPUT*		*OUTPUT*
	AluOp	**funct3**	**Alu Operation**
LUI	LUI	–	NOP
AUIPC	AUIPC	–	NOP
JAL	JAL	–	UNCONDITIONAL_JUMP
BRANCH	BRANCH	–	CONFRONTO_IF_EQUAL
LOAD	LOAD	–	SUM
STORE	STORE	–	SUM
ADDI	OP_IMM	000	SUM
ANDI		101	AND
SRAI		111	SHIFT
ADD	OP	000	SUM
SLT		010	CONFRONTO_SLT
XOR		100	XOR

Table 2.2: AluCommand depending on the AluOp and funct3 fields

2.1.4 Branch value provider

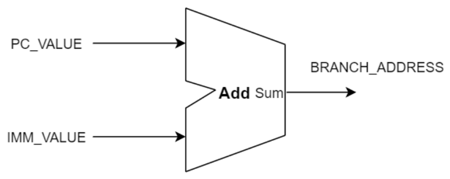


Figure 2.6: Branch Value Provider

This cell is a VHDL abstraction of an adder. Its name leads to its function to calculate the value of the branch address.

It receives as input the current instruction address stored in PC and the immediate value calculated from the branch instruction. It provides as output the sum of the inputs as required from Risc-V specifications (all inputs and outputs are on 32 bits, overflow is ignored).

2.1.5 Program counter manager

The program counter manager is the subsection that manages the next address selection.

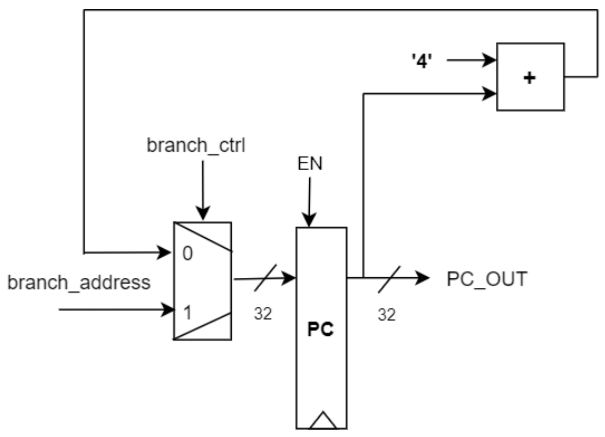


Figure 2.7: Program counter manager Block scheme

It takes as input the branch address and branch signal and provide as output the address of the next instruction, stored in the PC. The additional output PC+4 has been made available to implement JAL instruction.

Its structure is very simple since it is made up of a register, a +4 adder to perform sequential execution and a mux in order to select the jump address if required.

2.1.6 Write Back selector

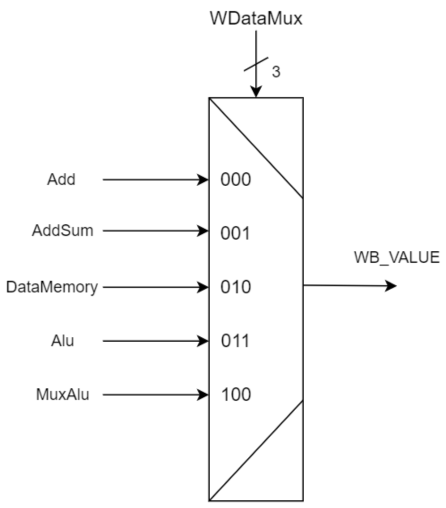


Figure 2.8: Writeback Selector

This section of the VHDL code implements the mux that manages the data to be stored in the proper register defined by the instruction. The mux is implemented to perform the following selections:

- ”100” : immediate value → required to implement LUI
- ”001” : PC+ (immediate x 2) → required to implement AUIPC instruction
- ”000” : PC+4 → required to implement JAL instruction
- ”010” : value from data memory → required to implement LW instruction
- ”011” : value from the Alu → required to implement the remaining instructions

The Control signal is WDataMux. It is properly generated by the Control Unit and registered in all the pipe stages.

2.1.7 Data hazard management

In order to recognize and manage in HW the data dependencies between instructions that have to be executed, two units are considered:

- Forwarding unit;
- Hazard detection unit.

Forwarding unit

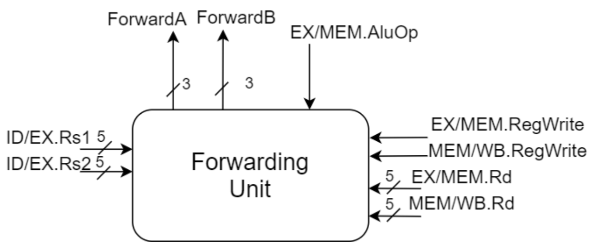


Figure 2.9: Forwarding Unit

Notice that in figure 2.9 and in the following discussion, the notation *Stage1/Stage2.Sig1* means that we are referring to the signal *Sig1* coming from the register between *Stage1* and *Stage2*.
The Forwarding Unit permits to recognize and solve the **read after write** (**RAW**) data dependencies.
It is possible to recognize them verifying two conditions:

- **EX/MEM.RegWrite = 1 or MEM/WB.RegWrite = 1**

  This means that the instruction in MEM stage or that in WB stage include a write operation on the register files;

- **EX/MEM.RegisterRd = ID/EX.RegisterRs1 or EX/MEM.RegisterRd = ID/EX.RegisterRs2 or MEM/WB.RegisterRd = ID/EX.RegisterRs1 or MEM/WB.RegisterRd = ID/EX.RegisterRs2**

  We can know if a data dependecy occurs verifying that one of the source registers of the instruction in the EX stage and the destination register of the instruction in the MEM stage or that in the WB stage are equal.

Instead, it is possible to solve them based on these 2 observations:

- even if the input data for the second instruction involved is required in the decode stage (ID) actually it is used in the execution stage (EX) by the ALU;
- even if the data produced by the first instruction is written in the memory in the write stage (WB) actually it is already available by the execution stage (EX) as output of the ALU.

Based on that, giving in input to ALU directly its output operand or the extracted value from the memory you can avoid the hazard condition.
In this way, five input operands must be considered as possible inputs for the ALU:

- operands from the RF;
- operand from the EX/MEM pipe register (the out of the ALU);
- operand from the MEM/WB pipe register (the out of the data memory)
- The immediate value from the previous instruction (used for the LUI instruction)
- The branch address computed from the previous instruction (used for the AUIPC instruction)

In order to properly choose which input must be provided to the ALU, a MUX for each ALU input must be considered with the selection signals given by the Forwarding unit. The values of the selection signals are shown in the following table.

Mux Control	*Source*	*Explanation*
Forward(A/B) = 000	ID/EX	The operand comes from the register file.
Forward(A/B) = 010	EX/MEM	The operand is forwarded from the prior ALU result.
Forward(A/B) = 001	MEM/WB	The operand is forwarded from the data memory or an earlier ALU result.
Forward(A/B) = 011	EX/MEM	The operand is the immediate field computed in the previous cycle.
Forward(A/B) = 100	EX/MEM	The operand is the branch address computed in the previous cycle.

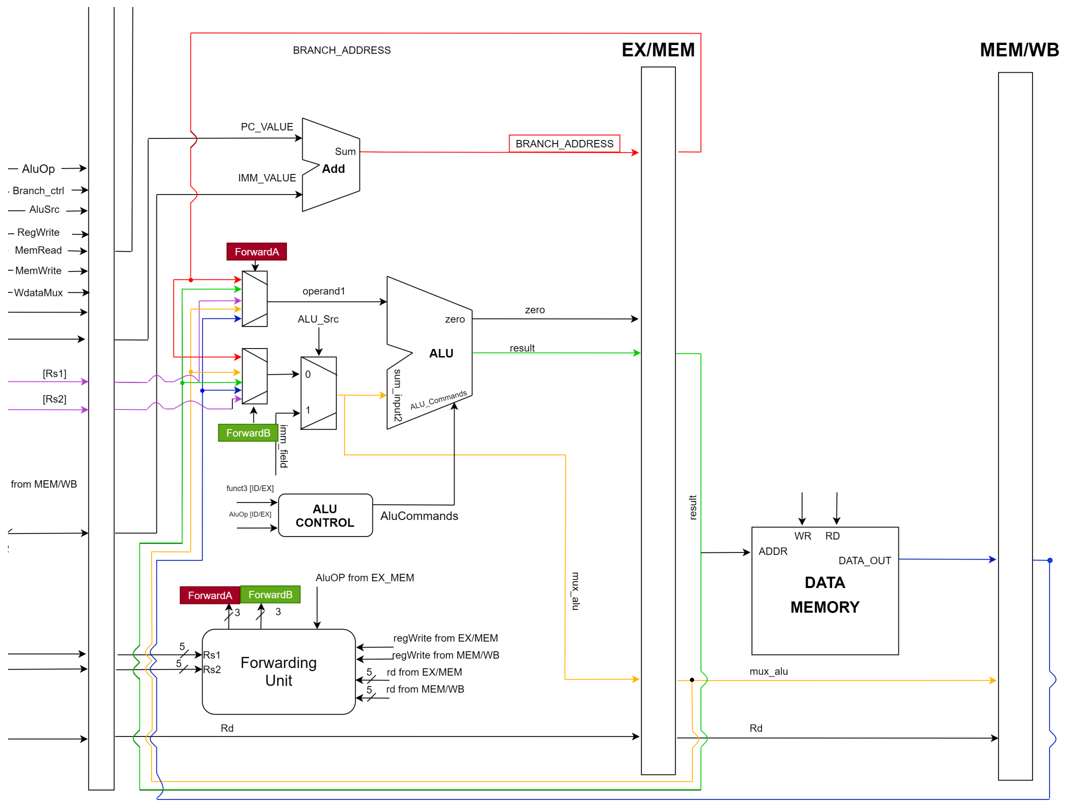


Figure 2.10: Forwarding Unit Functionality

Notice that in figure 2.10 only blocks related to the Forwarding Unit functionality are shown. This is made in order to make the connection between the blocks more readable.

Hazard detection unit

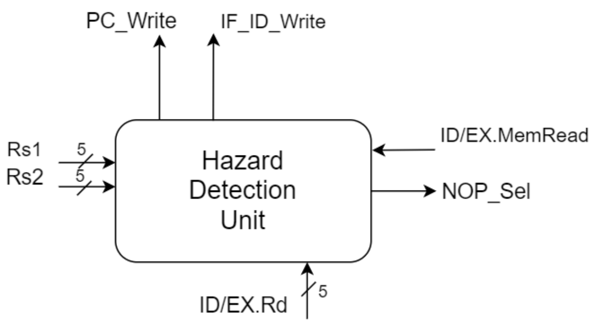


Figure 2.11: Hazard Detection Unit

The Forwarding unit does not permit to avoid **load-use** data hazard that can occurs when we have a load instruction to a certain destination register followed by an instruction that reads from the same register. When a load-use data hazard occurs, the Hazard Detection Unit implement these operations:

1. **Detect when an hazard is not fixed by Forwarding Unit**

   The load-use data hazard can be detected verifying 2 conditions:

- **ID/EX.MemRead = 1**
  when this control signal is assert we are sure that the instruction involved is a *load*;
- **ID/EX.RegisterRd = IF/ID.RegisterRs1**
  verifying this condition we are sure that the destination register of the *load* instruction is equal to the source register of the subsequent instruction;

2. **Insert a NOP by HW**

   For the already decoded instruction in order to avoid wrong execution, it is necessary make this instruction a NOP setting all the control signals to 0. This is done through the selection signal **NOP_sel**.

3. **Postpone the execution of the current instruction to the next cycle**

   To do that the writing of the Program Counter with the address of the next instruction to be executed and the writings on the IF/ID pipe stage are blocked. This is done by **negating** the control signals *PC_write* and *IF_ID_write*, which are the *Write Enable* signals of the PC and the first pipe register, respectively. In figure 2.12 the hazard detection unit inside the architecture is shown.

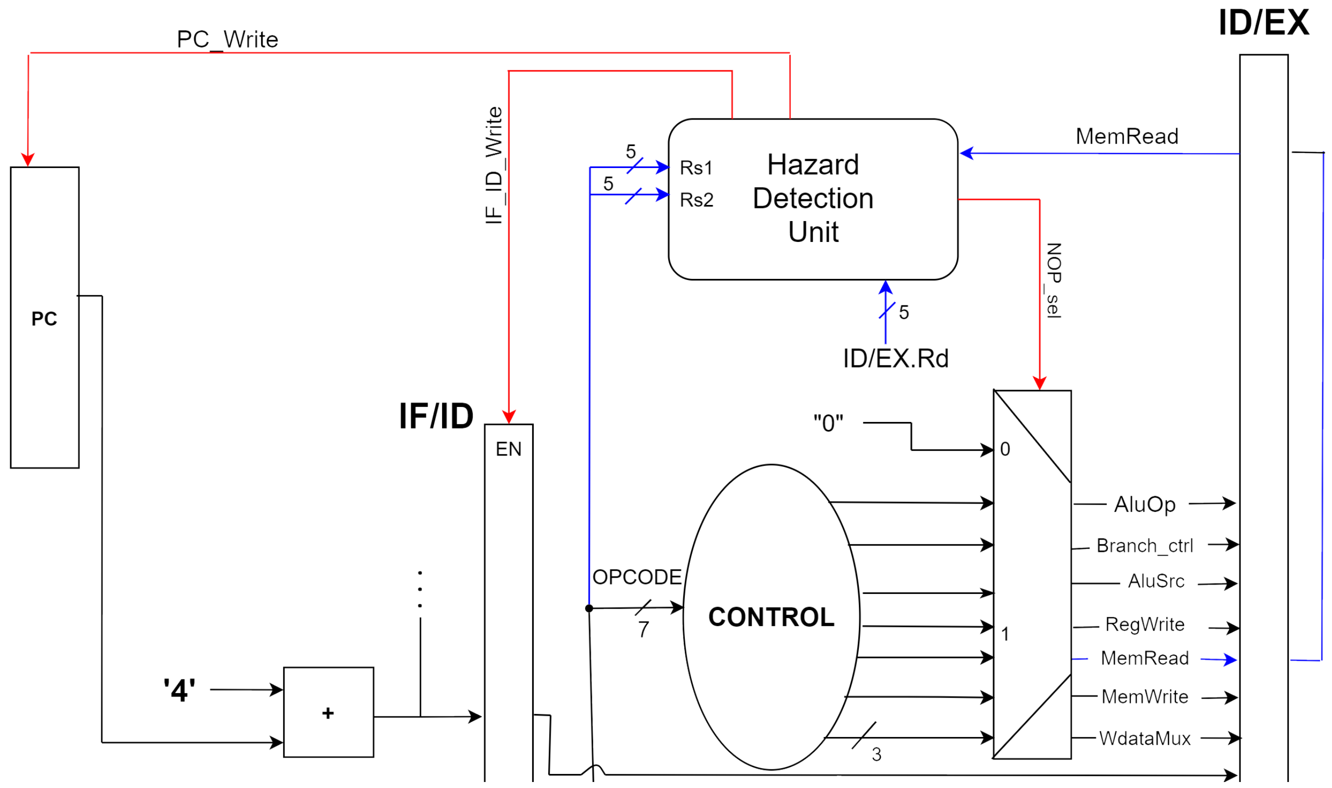


Figure 2.12: Hazard Detection Unit Functionality

Notice that in figure 2.12 only blocks and connections related to the Hazard detection Unit functionality are shown. This made in order to make the connection between the blocks more readable.

2.2 System Simulation

To ensure the correct behavior of all the instructions provided in the lite ISA, **each instruction** has been tested **individually**, before the overall test with the assembly program.
Note: All the testbench used are reported in the folder *instructions_test*.

2.2.1 Single Instructions Test

For this purpose, a specific testbench has been written for each instruction of the ISA.
Note: The content of this section can be found in the folder *instructions_test* inside the branch *risc-V-No_branch_handle*.

Test LUI

To verify the correct functioning of this instruction, the following operation has been performed.

```
LUI x03, 0x00000005 #(Code: 0x000051B7)
```

The testbench used is *tb_LUI.vhd*, the result of the simulation is in figure 2.13.

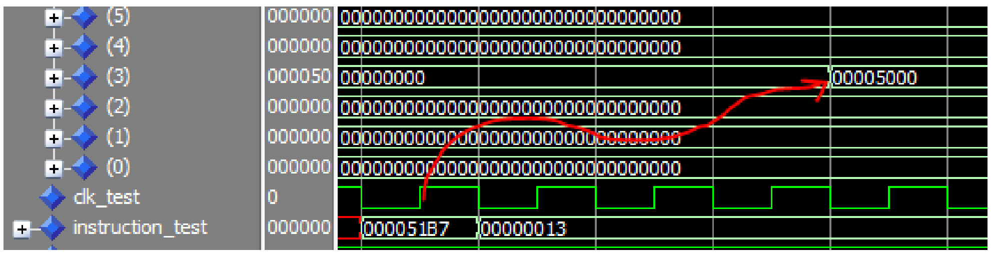


Figure 2.13: The correct value is written in R3 after 4 clock cycles, at the clk's falling edge.

In figure 2.13, the sign-extension performed by the imm-gen unit must be considered.
Even if the test involves only a unique test vector, it can be considered sufficient for our purposes (the same consideration holds for the other tests).

Test ADD

The following program has been executed in order to test this instruction.

```
LUI x05, 0x000D9D26 #(Code: 0xD9D262B7)
LUI x10, 0x00036B4D #(Code: 0x36B4D537)
ADD x11, x05, x10   #(Code: 0x00A285B3)
```

The result of the simulation (figure 2.14) shows a correct behavior of the system.

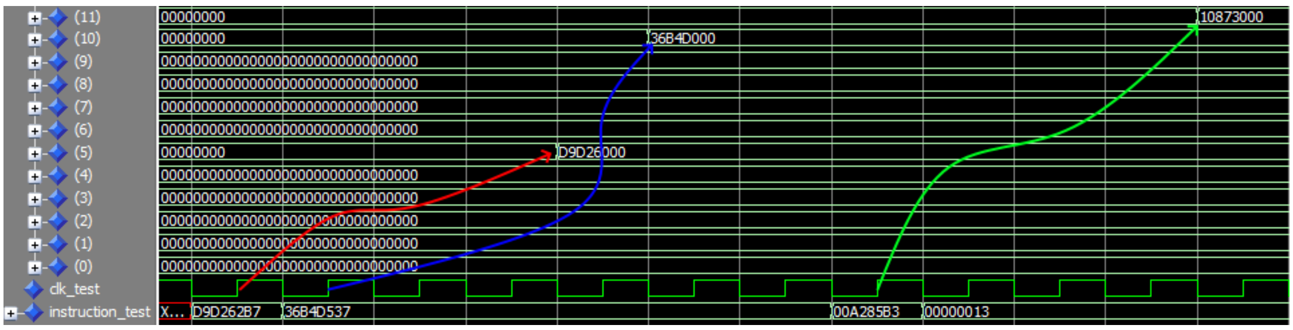


Figure 2.14: Result of *tb_ADD.vhd* simulation.

Notice that a bubble has been inserted in the instruction flow to avoid data hazards. The same bubbles will be inserted also in the other tests, because the hazard handling is not in our purpose yet.

Test ADDI

To test this instruction the following code has been executed.

```
LUI x08, 0x00036B4D       #(Code:0x36B4D437 )
ADDI x20, x08, 0xFFFFFD9D #(Code: 0xD9D40A13)
```

The result of the simulation is shown in figure 2.15.

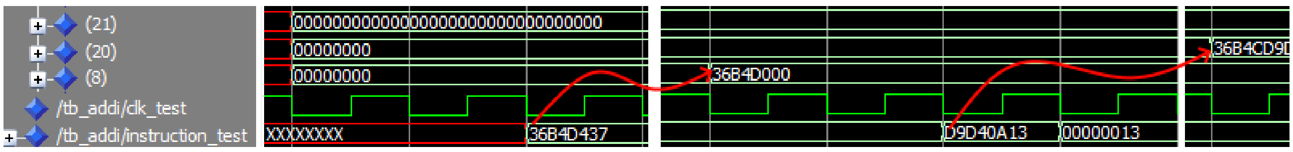


Figure 2.15: Result of *tb_ADDI.vhd* simulation.

Note: A blank space in the figure means that an unspecified amount of time has passed. This is done in order to fit better the image within the page.

Test ANDI and LW

In this case two instructions has been tested at once, using the following program.

```
LUI x14, 0x00010010       #Loads the base address for the load (Code: 0x10010737 )
LW R1, R14, 0x0000001C    #Loads from memory the data "0x0FAAF001" (Code: 0x01C72083)
ANDI x20, x01, 0x0000059D #(Code: 0xD9D40A13)
```

The result of the simulation is shown in figure 2.16.

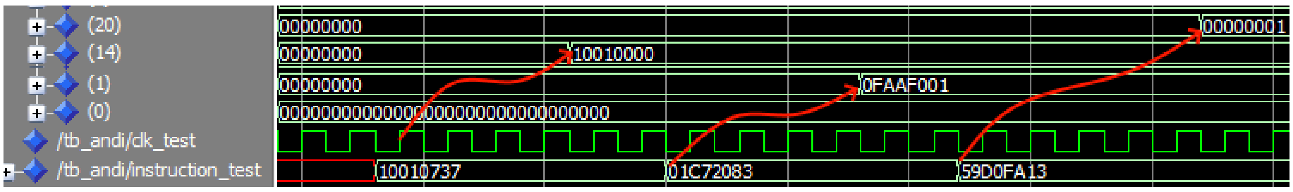


Figure 2.16: Result of *tb_ANDI.vhd* simulation.

Test AUIPC

To check the correct behavior here, a single instruction has been executed:

```
AUIPC x31, 0x1BA3A000 #(Code: 0x1BA3AF97)
```

After waiting the PC to be at value $(4194508)_{10}$, the value 0x1BA3A000 $= (463708160)_{10}$ is summed and the result is written in r31.

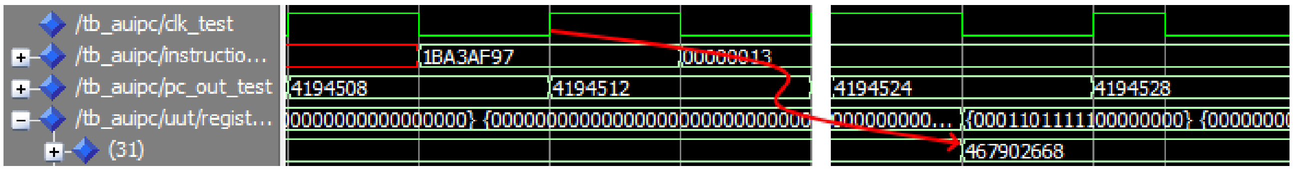


Figure 2.17: Result of *tb_AUIPC.vhd* simulation.

Since $4194508 + 463708160 = 467902668$, we can confirm the correctness of the result.

Test BEQ

To test this instruction the following code has been executed.

```
    LUI x3, 0x000D8A47
    LUI x5, 0x000D8A47
    BEQ x3, x5, 0x00000014 #offset=20 #(Code: 0x00328A63)
```

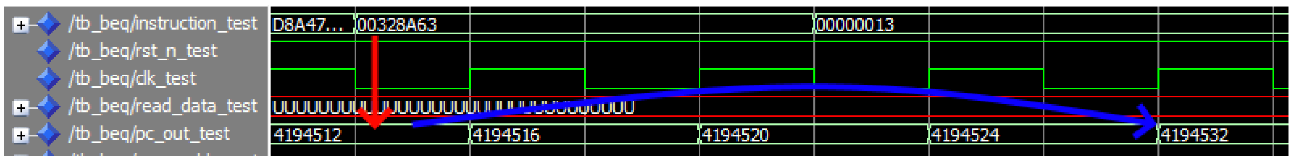


Figure 2.18: Result of *tb_BEQ.vhd* simulation.

The *BEQ* instruction is located at address $(4194512)_{10}$, because of the fact that $< rs1 > = < rs2 >$ the PC loads $4194512 + 20 = 4194532$. The branch has been taken correctly.

Test SLT

To test this instruction two different programs have been executed

1. *Do not set*

In this program the register r11 must not be set to 1 since the content of rs1 (r5) is greater than the content of rs2 (r10):

```
    LUI x10, 0xD9D26000 #Load -640524288  in R10 (Code: 0xD9D26637)
    LUI x05, 0x36B4D000 #Load  9178192392 in R5  (Code: 0x36B4D2B7)
    SLT x11, x5, x10    #Set <R11>=1 if <R5> < <R10> (Code: 00A2A5B3)
```

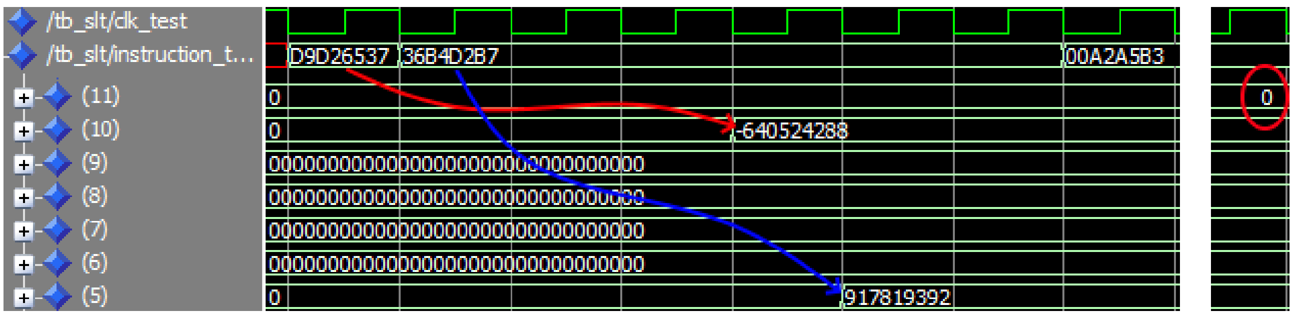


Figure 2.19: Result of *tb_SLT.vhd* when rs11 must NOT be set.

From figure 2.19, the simulation result confirm the correctness of the SLT when $rs1 > rs2$.

2. *Set*

In this program the register r11 must be set to 1 since the content of rs1 (r5) is less than the content of rs2 (r10):

```
LUI x05, 0xD9D26000 #Load -640524288  in R5 (Code: 0xD9D262B7)
LUI x10, 0x36B4D000 #Load  9178192392 in R10  (Code: 0x36B4D537)
SLT x11, x5, x10    #Set <R11>=1 if <R5> < <R10> (Code: 00A2A5B3)
```

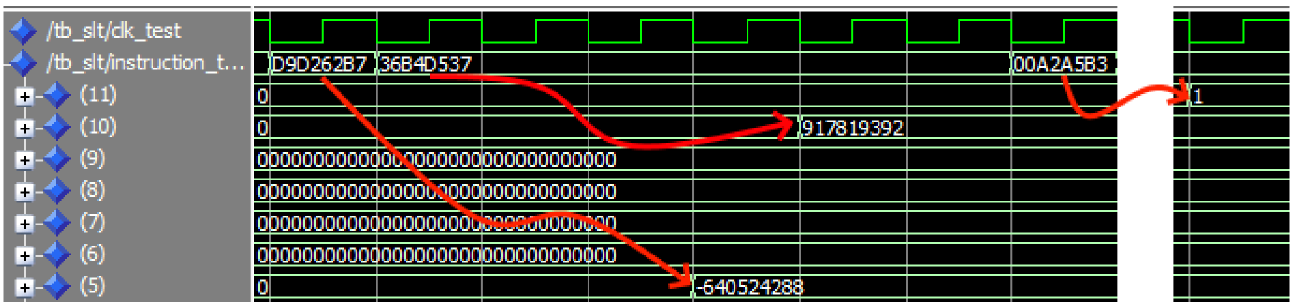


Figure 2.20: Result of *tb_SLT.vhd* when rs11 must be set.

From figure 2.19, the simulation result confirm the correctness of the SLT when $rs1 < rs2$. We can conclude that the SLT works correctly in both cases, for $rs1 > rs2$ or $rs1 < rs2$.

Test SRAI

The testing here is done with this simple program:

```
LUI x05, 0x000CD244      #(Code: 0xCD2442B7)
SRAI x11, x5, 0x00000005 #(Code: 0x4052D593)
```

Actually, what is done is to load 11001101001001000100000000000000 in the register R5, then shift it 5 times towards the right to obtain 11111110011010010010001000000000. The shift is "arithmetic": the sign of the original number is preserved.
In figure 2.21 the result of the simulation is shown, the five zeroes in green are "removed" with the x5 shift.

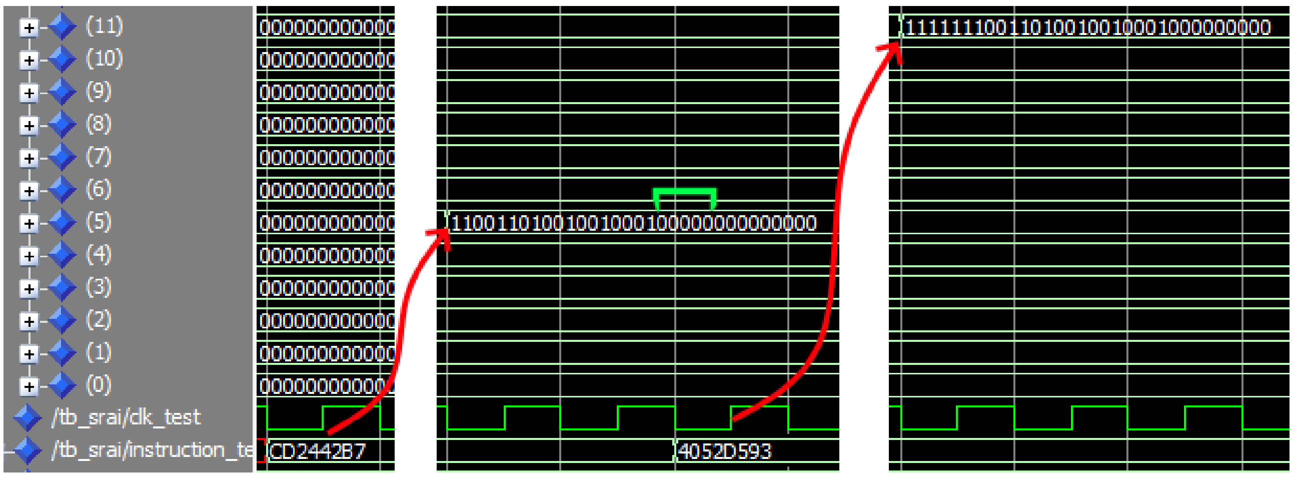


Figure 2.21: Result of *tb_SRAI.vhd* simulation.

Test SW

To test the storing of a word it's necessary to store the base address and the data to be written in two different registers. That is what is done in the following script.

```
LUI x03, 0x000D8A47      #Load the DATA in R3 (Code: 0xD8A471B7)
LUI x05, 0x00010010      #Load the BASE in R5 (Code: 0x100103B7)
SW x05, x03, 0x00000004 #Store the DATA (Code: 0x0032A223)
```

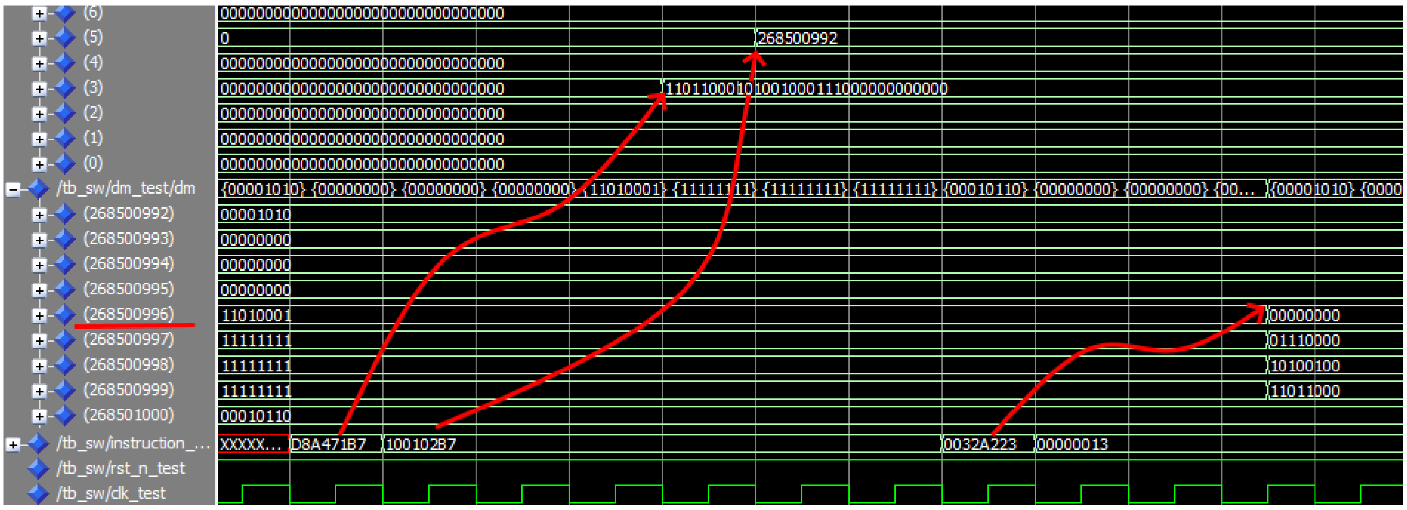


Figure 2.22: Result of *tb_SW.vhd* simulation.

The result of the simulation confirms the correct behavior of the instruction. In fact, the data has been stored at the address 268500992 + 4 = 268500996 which is a byte address. This means that the stored word spans from 268500996 to 268500999.

Test XOR

This is an operation on registers, so the operands are loaded in rs1 and rs2, then the XOR is performed. The script used is the following.

```
LUI x05, 0x000D9D26 #Load rs1 (Code: 0xD9D262B7)
LUI x10, 0x00036B4D #Load rs2 (Code: 0x36B4D537)
XOR x11, x5, x10    #(Code: 00A2C5B3)
```

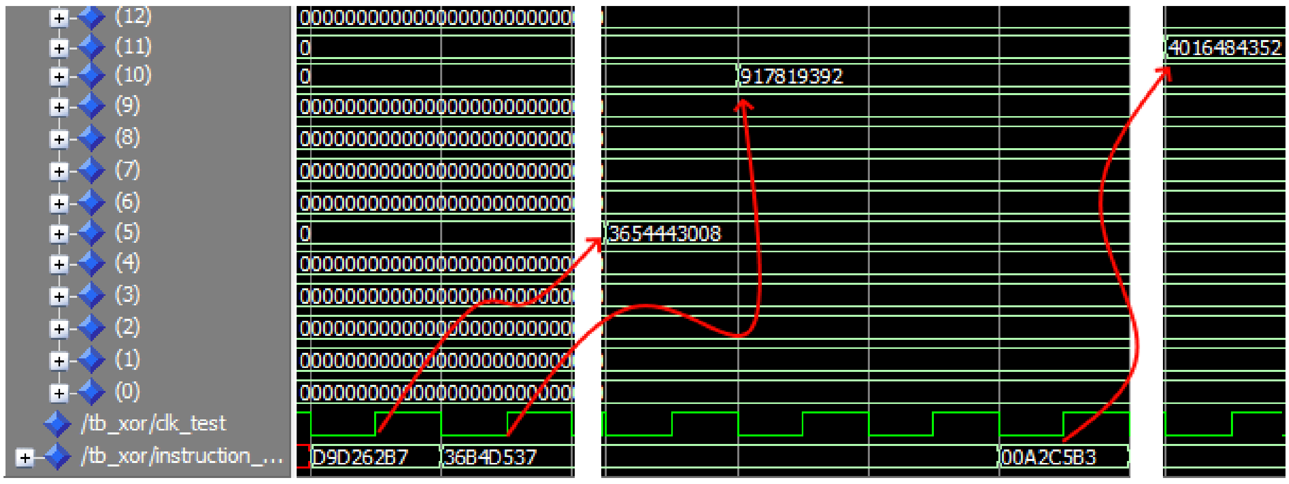


Figure 2.23: Result of *tb_XOR.vhd* simulation.

The operation performed can be expressed in base 2 as:

$$11011001110100100110000000000000 \bigoplus 00110110101101001101000000000000 = 11101111011001101011000000000000$$

Test JAL

To test the JAL instruction, a single instruction suffices:

```
JAL x06, 0xFFFFFFFC #Jump back of an instruction (PC=PC-4) Code: 0xFFDFF36F
```

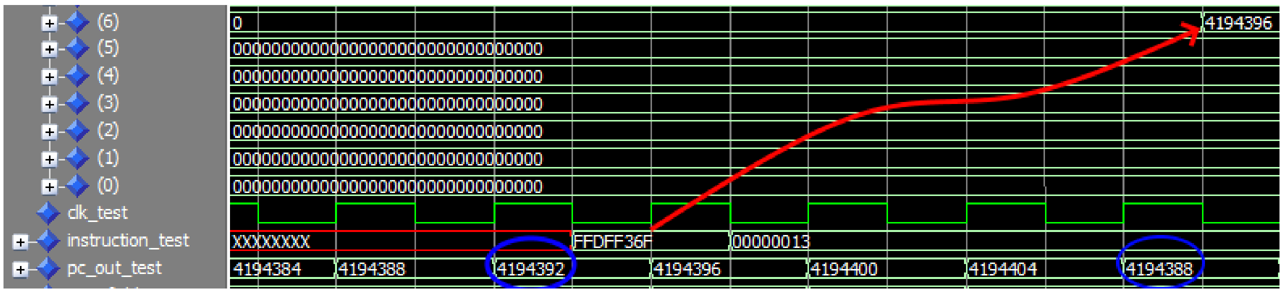


Figure 2.24: Result of *tb_JAL.vhd* simulation.

Notice that the JAL allows to store the value of the instruction which follows the address of the JAL ($4194392 + 4 = 4194396$). When the jump happens, the PC is brought back to instruction which preceeds the JAL ($4194392 - 4 = 4194388$, highlited in blue).

This concludes the first test phase. Even though the test vectors were not various, this step is useful to correct the great part of the errors. Instead of testing the architecture directly on a complex program, focusing on the single instruction is useful to find the bugs easily.

2.2.2 Test with the assembly program

The assembly program

To verify the correct behaviour of the processor, an assembly program is executed. This program takes an array v and computes $min\{|v[i]|\}$, the minimum of the absolute value, where $v[i]$ is the $i-th$ element in the array.
The source code *minv-rv.s* can be found in section 6.1.
This particular code contains some **control hazards**, that happen when we must take a jump or a branch. For example, in this portion of code the instructions are:

```
slt x11,x10,x13   # x11 = (x10 < x13) ? 1 : 0
beq x11,x0,loop   # next element
add x13,x10,x0    # update min
```

In the first instruction, register R11 is set to 1 if $< R10 > << R3 >$. If this is the case, the subsequent instruction must take the branch and go to *loop*. Due to the pipelining, the "add" instruction will be loaded even if we do not need it, causing the register R13 to be updated with the wrong data. Since the branch is taken after the **EX/MEM** register (where the *branch_ctrl* signal is computed), we need to insert 3 NOPs every time we find a jump instruction. The code which takes into account this limitation in shown in section 6.2 (*minv-rv_NOPs.s*).
Note: In this first version of the architecture, branches are not handled efficiently, in the next section the "Branch Taken" technique will solve this problem.

Source code compilation and data structure

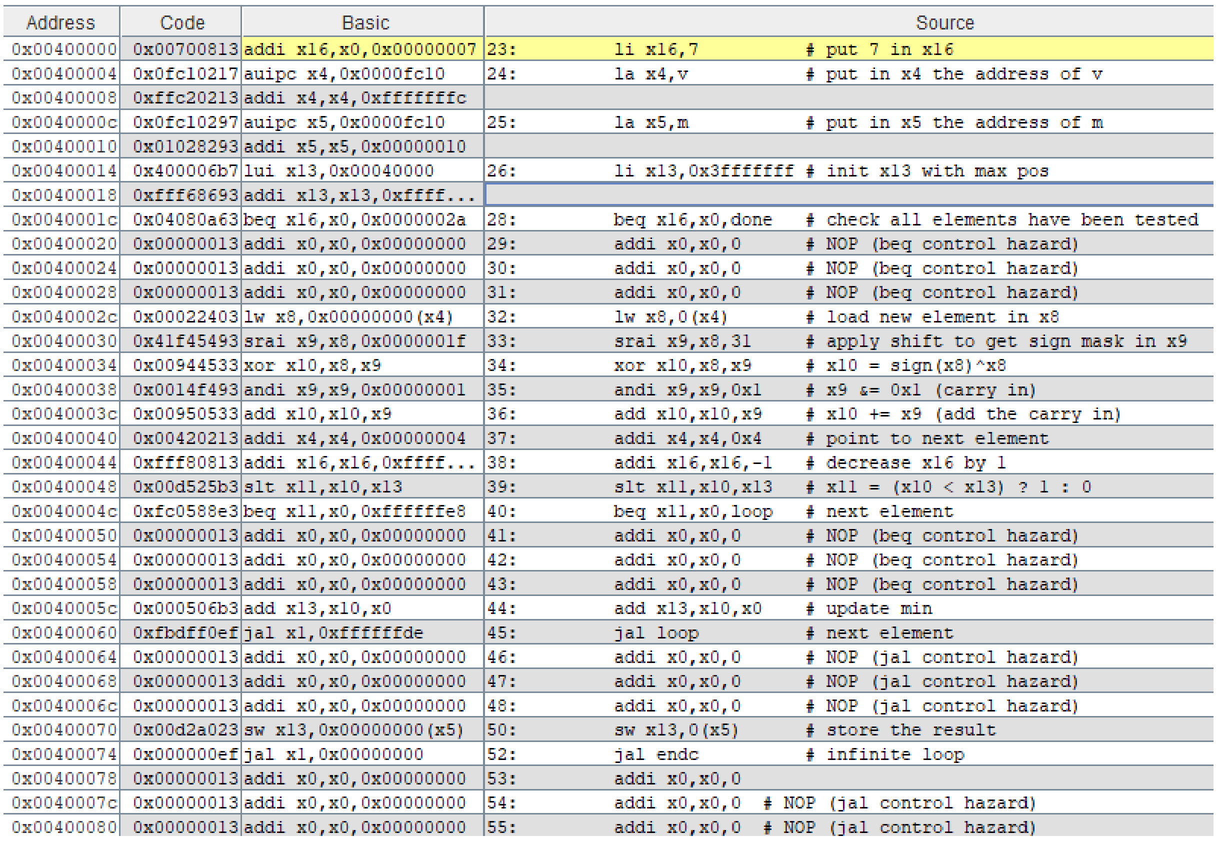
To feed the processor with the assembly code, it's necessary to use a compiler to translate the mnemonic instructions into hex values. This operation is done by the RARS simulator. In figure 2.25 the result of the compilation is shown.
Particular attention must be given to the data structure and the meaning of each register. Tables 2.3 and 2.4 show the purpose of each register in the program and the structure of the data memory.

Register	Purpose
x0	Contains the constant 0x00000000
x4	Contains the address of the current value of v that is checked
x5	Contains the address of m (the minimum)
x8 to x9	Contain temporary variables useful for the abs computing
x10	Constains the absolute value of the current v element
x11	Is a flag which tells if the current element is the minimum
x13	Contains the current value of the minimum
x16	Contains the number of element remained for checking

Table 2.3: Purpose of each register in the program

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	m
Value (DEC)	10	-47	22	-3	15	27	-4	0
Address (DEC)	268500992	268500996	268501000	268501004	268501008	268501012	268501016	268501020

Table 2.4: Data memory structure expected from the instruction flow



Address	Code	Basic	Source
0x00400000	0x00700813	addi x16,x0,0x00000007	23: li x16,7 # put 7 in x16
0x00400004	0x0fc10217	auipc x4,0x0000fc10	24: la x4,v # put in x4 the address of v
0x00400008	0xffc20213	addi x4,x4,0xfffffffc	
0x0040000c	0x0fc10297	auipc x5,0x0000fc10	25: la x5,m # put in x5 the address of m
0x00400010	0x01028293	addi x5,x5,0x00000010	
0x00400014	0x400006b7	lui x13,0x00040000	26: li x13,0x3fffffff # init x13 with max pos
0x00400018	0xfff68693	addi x13,x13,0xffff...	
0x0040001c	0x04080a63	beq x16,x0,0x0000002a	28: beq x16,x0,done # check all elements have been tested
0x00400020	0x00000013	addi x0,x0,0x00000000	29: addi x0,x0,0 # NOP (beq control hazard)
0x00400024	0x00000013	addi x0,x0,0x00000000	30: addi x0,x0,0 # NOP (beq control hazard)
0x00400028	0x00000013	addi x0,x0,0x00000000	31: addi x0,x0,0 # NOP (beq control hazard)
0x0040002c	0x00022403	lw x8,0x00000000(x4)	32: lw x8,0(x4) # load new element in x8
0x00400030	0x41f45493	srai x9,x8,0x0000001f	33: srai x9,x8,31 # apply shift to get sign mask in x9
0x00400034	0x00944533	xor x10,x8,x9	34: xor x10,x8,x9 # x10 = sign(x8)^x8
0x00400038	0x0014f493	andi x9,x9,0x00000001	35: andi x9,x9,0x1 # x9 &= 0x1 (carry in)
0x0040003c	0x00950533	add x10,x10,x9	36: add x10,x10,x9 # x10 += x9 (add the carry in)
0x00400040	0x00420213	addi x4,x4,0x00000004	37: addi x4,x4,0x4 # point to next element
0x00400044	0xfff80813	addi x16,x16,0xffff...	38: addi x16,x16,-1 # decrease x16 by 1
0x00400048	0x00d525b3	slt x11,x10,x13	39: slt x11,x10,x13 # x11 = (x10 < x13) ? 1 : 0
0x0040004c	0xfc0588e3	beq x11,x0,0xffffffe8	40: beq x11,x0,loop # next element
0x00400050	0x00000013	addi x0,x0,0x00000000	41: addi x0,x0,0 # NOP (beq control hazard)
0x00400054	0x00000013	addi x0,x0,0x00000000	42: addi x0,x0,0 # NOP (beq control hazard)
0x00400058	0x00000013	addi x0,x0,0x00000000	43: addi x0,x0,0 # NOP (beq control hazard)
0x0040005c	0x000506b3	add x13,x10,x0	44: add x13,x10,x0 # update min
0x00400060	0xfbdff0ef	jal x1,0xffffffde	45: jal loop # next element
0x00400064	0x00000013	addi x0,x0,0x00000000	46: addi x0,x0,0 # NOP (jal control hazard)
0x00400068	0x00000013	addi x0,x0,0x00000000	47: addi x0,x0,0 # NOP (jal control hazard)
0x0040006c	0x00000013	addi x0,x0,0x00000000	48: addi x0,x0,0 # NOP (jal control hazard)
0x00400070	0x00d2a023	sw x13,0x00000000(x5)	50: sw x13,0(x5) # store the result
0x00400074	0x000000ef	jal x1,0x00000000	52: jal endc # infinite loop
0x00400078	0x00000013	addi x0,x0,0x00000000	53: addi x0,x0,0
0x0040007c	0x00000013	addi x0,x0,0x00000000	54: addi x0,x0,0 # NOP (jal control hazard)
0x00400080	0x00000013	addi x0,x0,0x00000000	55: addi x0,x0,0 # NOP (jal control hazard)

Figure 2.25: Results of the *minv-rv_NOPs.s* compilation by the RARS simulator

Test bench

The Risc-V functionality is tested using the testbench *tb_one.vhd* shown in fig. 2.26.

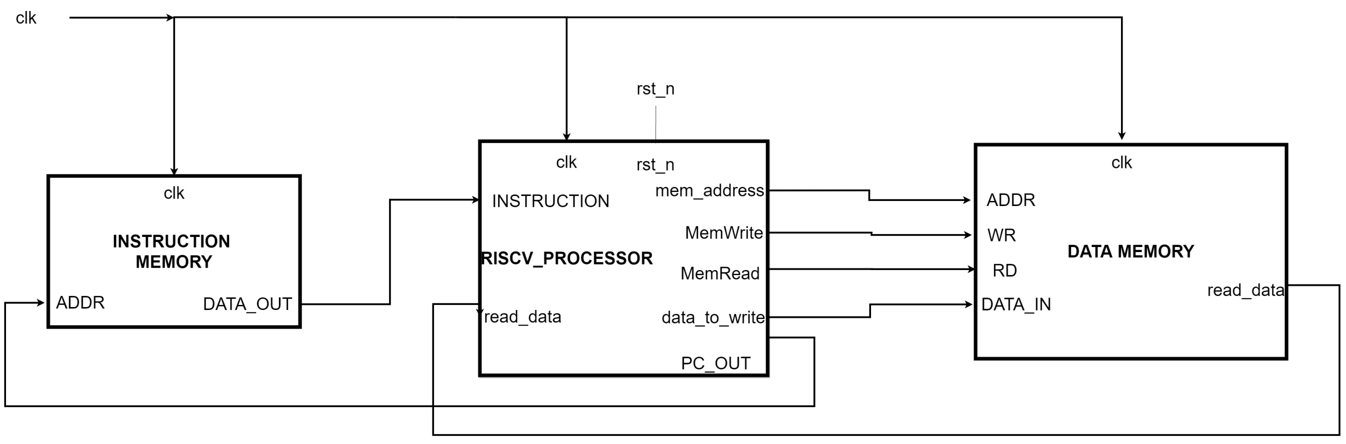


Figure 2.26: Test bench

The processor is feeded with the program compiled by the RARS and the evolution of the system is observed using the software *Modelsim*.

Data memory

According to table 2.4, a data memory has been prepared for the simulation.

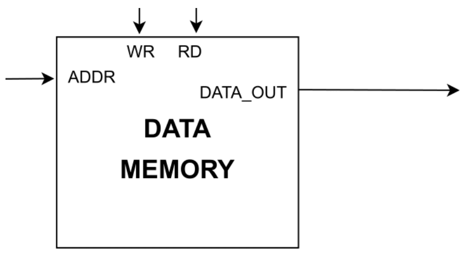


Figure 2.27: Data Memory

The memory is provided with a write and a read enable (WR and RD), writing is synchornous while reading is asynchronous.

Instruction Memory

According to the instructions provided by the RARS build of the code (fig. 2.25), the instruction memory has been prepared for the simulation.

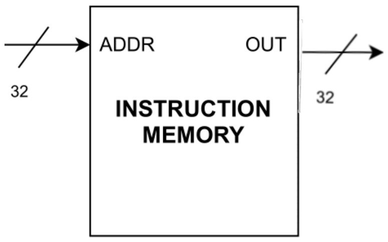


Figure 2.28: Data Memory

Note: Both the the Data Memory and the Instruction Memory will not be synthesized. They are included only in the testbench.

Simulation Results

The correctness of the results have been checked directly by analyzing the behaviour of the registers and the memory. In figures 2.29 and 2.30 some particular aspects are pointed out.

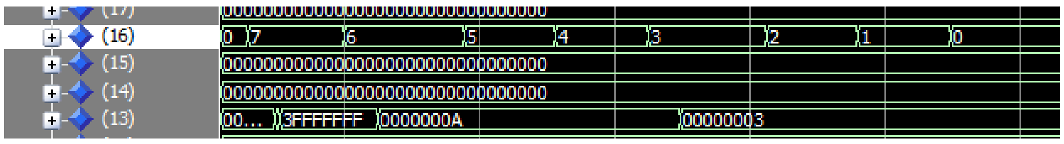


Figure 2.29: The register R16 is decremented while R13 is updated

From figure 2.29 we can notice that R16 and R13 behave as expected, considering their purpose (table 2.3).

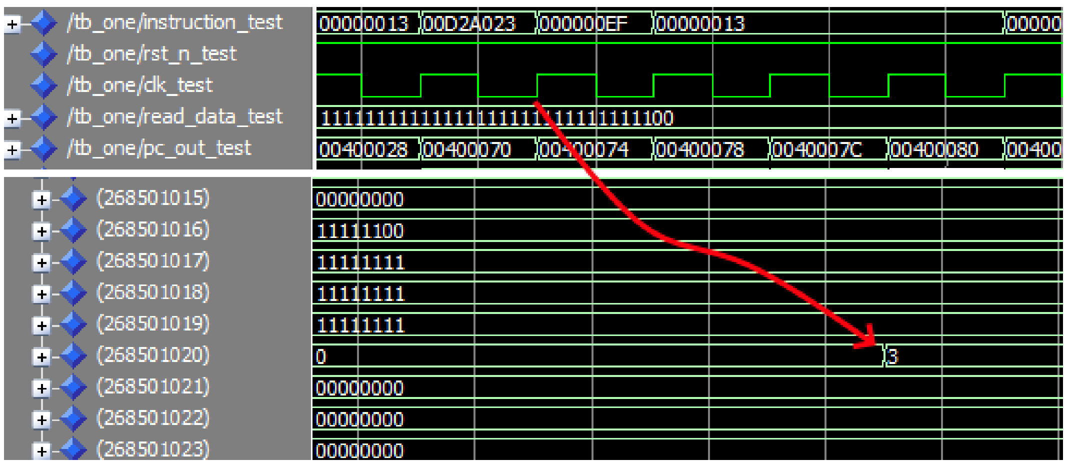


Figure 2.30: The value of the minimum (3) is written in the DM at the correct address

In figure 2.30 is showed that the correct value of the minimum $(3)_{10}$ is written in the data memory at the address of m (268501020).

2.3 Processor Logic Synthesis

Synopsis software is used to synthesize the circuit starting from the VHDL design. First of all, a synthesis is done with the clock set to 0 in order to evaluate the minimum clock period from the negative slack pointed out in the simulation results.

```
#!/bin/bash
cd /home/isa37/git/lab3/syn/
rm -r work
mkdir work
mkdir analysis_results
source /software/scripts/init_synopsys_64.18
#lancia synopsis
dc_shell-xg-t
#*******Reading VHDL source files**************
analyze -f vhdl -lib WORK ../src/riscv_pkg.vhd
analyze -f vhdl -lib WORK ../src/REGN.vhd
analyze -f vhdl -lib WORK ../src/IMM_GEN.vhd
analyze -f vhdl -lib WORK ../src/ALU.vhd
analyze -f vhdl -lib WORK ../src/ALU_CONTROL.vhd
analyze -f vhdl -lib WORK ../src/BRANCH_VALUE_PROVIDER.vhd
analyze -f vhdl -lib WORK ../src/CONTROL.vhd
analyze -f vhdl -lib WORK ../src/REGISTER_FILE.vhd
analyze -f vhdl -lib WORK ../src/PROGRAM_COUNTER_MANAGER.vhd
analyze -f vhdl -lib WORK ../src/WRITE_BACK_SELECTOR.vhd
analyze -f vhdl -lib WORK ../src/PIPE1_REG.vhd
analyze -f vhdl -lib WORK ../src/PIPE2_REG.vhd
analyze -f vhdl -lib WORK ../src/PIPE3_REG.vhd
analyze -f vhdl -lib WORK ../src/PIPE4_REG.vhd
analyze -f vhdl -lib WORK ../src/RISCV_PROCESSOR.vhd
```

```
#Before completing the reading of source we set one parameter to preserve rtl names in the netlist to e
set power_preserve_rtl_hier_names true
#Launch elaborate command to load the components
#elaborate <top entity name> -arch <architecture name> -lib WORK > ./elaborate.txt
elaborate  RISCV_PROCESSOR -arch rtl -lib WORK > ./elaborate_results.txt
#uniquify #optional command to addres to only 1 specific architecture
link

#*******  Applying constraints   ***************
#create 100 Mhz clock
create_clock -name MY_CLK -period 0 clk
set_dont_touch_network MY_CLK

#jitter simulation
set_clock_uncertainty 0.07 [get_clocks MY_CLK]

#input/output delay
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] clk]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]

#set output load (buffer x4 used)
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set_load $OLOAD [all_outputs]

#flatten the hierarchy
ungroup -all -flatten

#*********    Start the syntesis    *************
compile >  ./analysis_results/compilation_results.txt

#*********    Save the results      *************
report_timing  >  ./analysis_results/timing_results.txt
report_area    >  ./analysis_results/area_results.txt
report_resources > ./analysis_results/resource_report.txt
```

2.3.1 Elaboration Results

The output of the elaboration of the HDL system has been logged on a file (*elaborate_results.txt*), **no errors** or warnings were found. Moreover, the absence of latches has been verified, **all** the memory elements are **flip-flops**. An extract of the log file is reported in section 6.6.

2.3.2 Maximum clock frequency evaluation

From the negative slack pointed out in the simulation results, it is possible derive the **minimum clock period:** $3.08ns$. Then, the synthesis is done setting the clock period to $4 * T_{min}$.
In report 2.1 an extract of the timing results report is shown.

Report 2.1: Extract of the file *timing_results.txt*

```
Startpoint: PIPE3/Q_reg[RD][1]
            (rising edge-triggered flip-flop clocked by MY_CLK)
Endpoint: PIPE3/Q_reg[ALU_RESULT][31]
          (rising edge-triggered flip-flop clocked by MY_CLK)
Path Group: MY_CLK
Path Type: max

Des/Clust/Port     Wire Load Model      Library
------------------------------------------------
RISCV_PROCESSOR    5K_hvratio_1_1       NangateOpenCellLibrary

Point                                              Incr       Path
-----------------------------------------------------------------------
clock MY_CLK (rise edge)                           0.00       0.00
clock network delay (ideal)                        0.00       0.00
PIPE3/Q_reg[RD][1]/CK (DFFR_X1)                    0.00       0.00 r

[...]

PIPE3/Q_reg[ALU_RESULT][31]/CK (DFFR_X1)           0.00       3.01 r
library setup time                                -0.04       2.97
data required time                                            2.97
-----------------------------------------------------------------------
data required time                                            2.97
data arrival time                                            -2.97
-----------------------------------------------------------------------
slack (MET)                                                   0.01

1
```

2.3.3 Area evaluation

According to Synopsis evaluation, the synthesized processor occupies an **area** of $15298.99\mu m^2$.

Note: For this first version of the processor, no netlist has been produced. The netlist extraction and simulation is done for the definitive version of the processor, in chapter 3.

CHAPTER 3

Risc-V: Advanced Architecture to face Control Hazards

As already mentioned in section 2.2.2, the code contains control hazards every time a jump is needed. In the previous architecture, the **control hazard** forced us to introduce three NOP instructions after each branch instruction, the result is a waste in terms of execution time.

In this **Advanced Architecture**, HW and SW techniques are used to face the control hazard problem and avoid the waste in execution time.
Note: The contents discussed in this chapter are referred to the branch *risc-V-Branch_handle* of the github repository.

3.1 HW modification: Branch Taken

From hardware point of view, the **Branch taken** technique is implemented. It is based on the following modifications:

- **Anticipate the evaluation of branch outcome**

  The comparison between the content of the two registers coming from the RF is anticipated to the ID stage. In this way, when the instruction is a BEQ, it is possible to evaluate the branch outcome in the ID stage.

- **Anticipate the calculation of the target address**

  Since the immediate field and PC are already available in stage ID we can anticipate the target address calculation of one clock cycle.

The architecture with the **Branch Taken** technique is shown in fig. 3.2. The modifications with respect to the previous architecture are highlighted. With this new architecture it is possible to **reduce the branch delay** from three two just one clock cycle. Since the aim is to remove the software NOPs, the stall cycle is inserted via hardware by the Hazard detection unit.

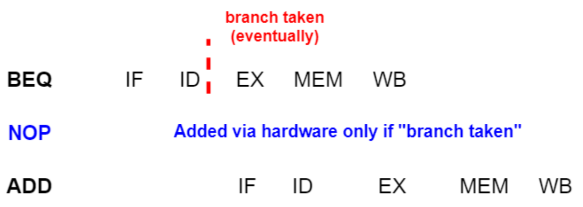

branch taken (eventually)
BEQ IF ID EX MEM WB
NOP Added via hardware only if "branch taken"
ADD IF ID EX MEM WB

Figure 3.1: Example of how the "branch taken" technique works

The line in red indicates the instant when the branch can be taken. If this is the case, the HW NOP forbids the loading of the ADD instruction, avoiding the computation of a wrong result.

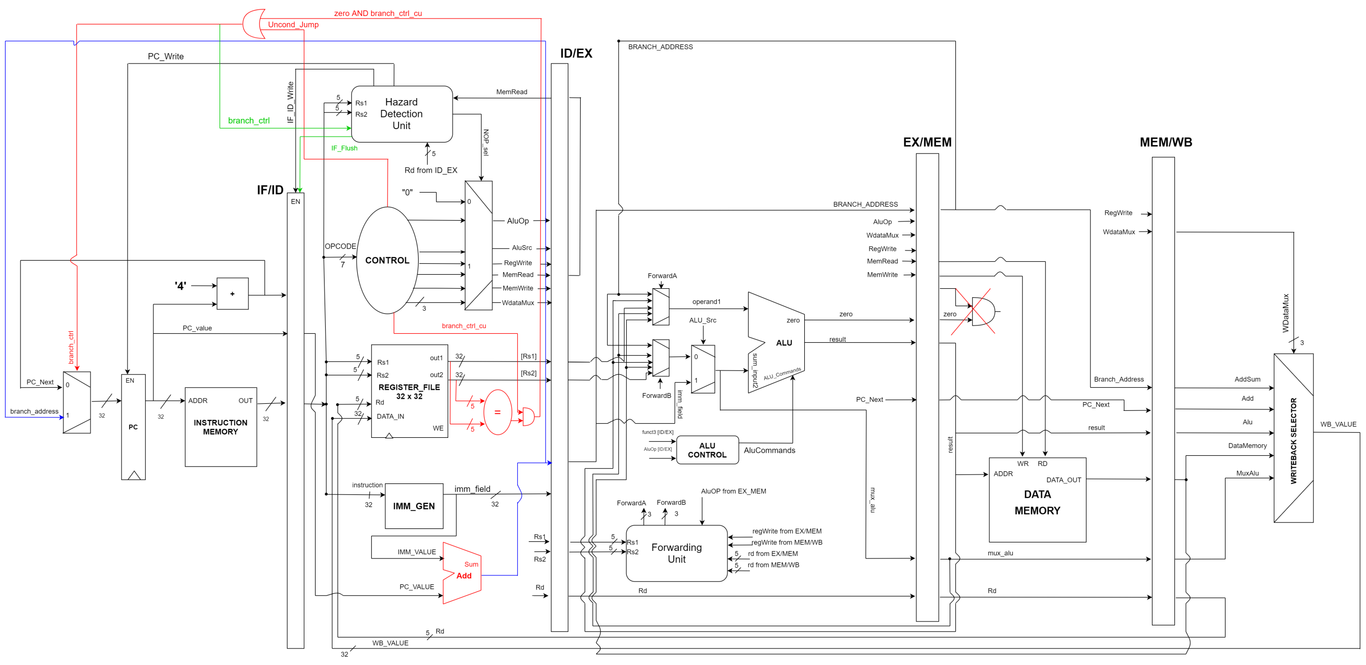


Figure 3.2: Risc-V processor architecture with the "Branch Taken" technique

3.1.1 Hazard Detection Unit for the branch delay reduction

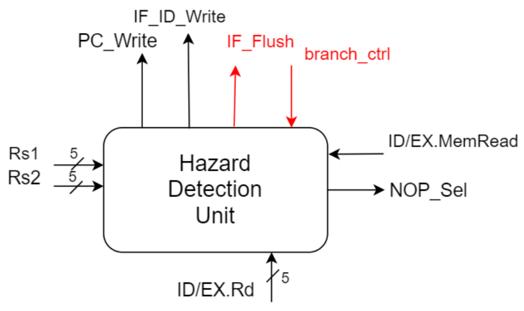


Figure 3.3: The Hazard Detection Unit for the Branch delay reduction

In figure 3.3 the new signals with respect to the previous architecture are highlighted. The **branch_ctrl** signal is used by this unit to understand when a branch must be taken. If that's the case, it provides

the insertion of the NOP **negating** the *IF_Flush* signal, which is a synchronous reset of the IF/ID stage. In this way, the next instruction will not be loaded and decoded. To propagate the NOP through the other stages, the *NOP_Sel* signal is set to zero.
With the use of the branch_ctrl signal, there is a penalty in terms of time only if a branch is **actually taken**, this avoids the pointless penalty of untaken branches.

3.2 SW modification: Instructions Rescheduling

The anticipation of the branch calculation from the MEM stage to the ID stage has a drawback, that is the ineffectiveness of the forwarding unit when the hazard involves a branch instruction.
To understand this concept, consider the following sequence of instructions in the original code (*minv-rv.s*).

```
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
```

The last two instructions generate a data hazard, since the result of the *SLT* is written in x11 and it's then needed by the *BEQ*. This hazard would normally be resolved by the Forwarding unit, but in this case the comparison between x11 and x0 is done in the ID stage. This is why the forwarding unit is useless in this case.Figure 3.4 shows this problem graphically.

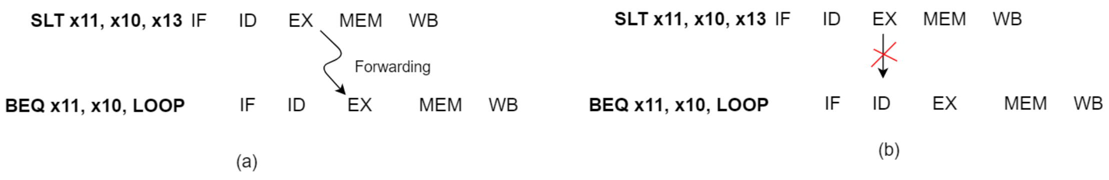

SLT x11, x10, x13 IF ID EX MEM WB
Forwarding
BEQ x11, x10, LOOP IF ID EX MEM WB
(a)
SLT x11, x10, x13 IF ID EX MEM WB
BEQ x11, x10, LOOP IF ID EX MEM WB
(b)

Figure 3.4: In picture (a), the branch taken technique is not applied: forwarding is possible. This is not the same with the anticipation of the branch computation, which needs the operand in the ID stage: forwarding is impossible.

A solution to this problem is **rescheduling**. This technique allows to separate in time the *SLT* and the *BEQ* exploiting instructions that can be executed in a different cycle without changing the semantic of the algorithm. In our case, we exploit the first two instructions (*addi*): their purpose is to prepare a counter and a memory pointer for the next iteration, so it's not a problem to move them next to the *SLT* until they're executed before the *BEQ*. The rescheduled code without hazards is the following one.

```
slt x11,x10,x13   # x11 = (x10 < x13) ? 1 : 0
addi x4,x4,0x4    # point to next element
addi x16,x16,-1   # decrease x16 by 1
beq x11,x0,loop   # next element
```

The BEQ's ID stage is now contemporary to the SLT's WB stage: the correctness of the result is granted by the RF transparency.
The code which includes the rescheduling is shown in section 6.3 (**min-rv_rescheduling.s**).

3.3 System simulation

The Risc-V functionality is tested using the testbench *tb_one.vhd* shown in fig.3.5.

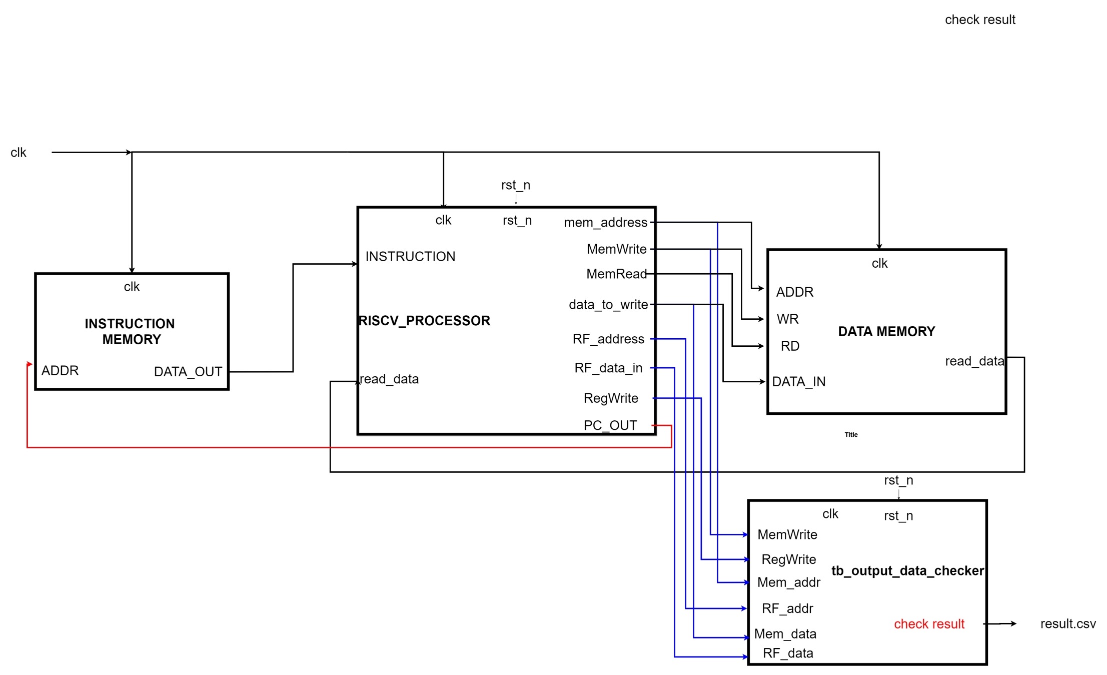


Figure 3.5: Test bench

The processor is feeded with the program compiled by the RARS and the evolution of the system is observed using the software *Modelsim*. The code tested in this case is the one showed in section 6.3. Notice that the NOP instructions have been removed, allowing for a smaller execution time with the same clock with the same clock period.
The correctness of the results have been checked, in this case, by analyzing the file *results.csv* in which are written the writing operations done on RF/memories. The file is filled by the *tb_output_data_checker* that, for each executed instruction, writes on this file the type of memory and the address in which the evaluated data is written and its value.

Report 3.1: *results.csv*

```
Mem Type, Address, Data
RF: 16, 7
RF: 4, 268500996
RF: 4, 268500992
RF: 5, 268501004
RF: 5, 268501020
RF: 13, 1073741824
RF: 13, 1073741823
RF: 8, 10
RF: 9, 0
RF: 10, 10
RF: 9, 0
RF: 10, 10
RF: 11, 1
RF: 4, 268500996
RF: 16, 6
RF: 13, 10
RF: 8, -47
RF: 9, -1
RF: 10, 46
RF: 9, 1
RF: 10, 47
RF: 11, 0
RF: 4, 268501000
RF: 16, 5
RF: 8, 22
RF: 9, 0
```

```
RF: 10, 22
RF: 9, 0
RF: 10, 22
RF: 11, 0
RF: 4, 268501004
RF: 16, 4
RF: 8, -3
RF: 9, -1
RF: 10, 2
RF: 9, 1
RF: 10, 3
RF: 11, 1
RF: 4, 268501008
RF: 16, 3
RF: 13, 3
RF: 8, 15
RF: 9, 0
RF: 10, 15
RF: 9, 0
RF: 10, 15
RF: 11, 0
RF: 4, 268501012
RF: 16, 2
RF: 8, 27
RF: 9, 0
RF: 10, 27
RF: 9, 0
RF: 10, 27
RF: 11, 0
RF: 4, 268501016
RF: 16, 1
RF: 8, -4
RF: 9, -1
RF: 10, 3
RF: 9, 1
RF: 10, 4
RF: 11, 0
RF: 4, 268501020
RF: 16, 0
DM: 268501020, 3
```

3.4 Logic Synthesis

Repeating the same step seen in chap.2.3, the synthesis of the architecture in which we have introduced the **Branch delay reduction** has these features:

- **Min Clock Period** 3.05 ns
- **Area** 15338.09 μm^2

Thus, we obtain the same result in terms of minimum clock period while the area is slightly increased. The increase of area is due to comparator of the addresses and the adder necessary to evaluate the branch address. Using this technique we can reduce stalls due to control hazard, so it permits to increase performance.

3.5 Synthesized Netlist extraction and testing

The netlist synthesized by Synopsis has been extracted using the following commands:

```
#We have to export the netlist in verilog. So that we impose verilog rules for the names of the interna
change_names -hierarchy -rules verilog
#We also save a file describing the delay of the netlist:
write_sdf ../netlist/myRiscv.sdf
#We can now save the netlist in verilog:
write -f verilog -hierarchy -output ../netlist/myRiscv.v
#and the constraints to the input and output ports in a standard format:
write_sdc ../netlist/myRiscv.sdc
```

Then, it has been tested using the same testbench shown in section 3.3 (and figure 3.5). The results are the same as shown in Report 3.1, so we can conclude that the synthesized netlist behaves correctly as our RTL description.

3.6 Place & Route

The Cadence's *Innovus* software is used to perform place and route operation of the system together with some files produced in the previous step. At first, a setup information file is imported into *Innovus*: it contains information about the paths based on the netlist synthesized by the design compiler (the *.v* file) and the constraints to the input and output ports (the *.sdc* file). This information are followed by commands that set the Standard Cell Library together with some fixed delays and the VDD and GND net name:

```
set IN_DIR "../netlist"
set TopLevelDesign "RISCV_PROCESSOR"
set in_verilog_filename "${IN_DIR}/myRiscv.v"
set in_sdc_filename "${IN_DIR}/myRiscv.sdc"

set LIB_DIR /software/dk/nangate45/liberty
set MyTimingLib ${LIB_DIR}/NangateOpenCellLibrary_typical_ecsm_nowlm.lib

set LEF_DIR /software/dk/nangate45/lef
set LEF_list [list ${LEF_DIR}/NangateOpenCellLibrary.lef]

set init_design_netlisttype "verilog"
set init_design_settop 1
set init_top_cell $TopLevelDesign
set init_verilog $in_verilog_filename

set init_lef_file "${LEF_list}"

set aspect_ratio 1.0
set target_row_utilization 0.6

set CustomDelayLimit 1000
set CustomNetDelay 1000.0ps
set CustomNetLoad 0.5pf
set CustomInputTranDelay 0.1ps

set MycapTable $LEF_DIR/captables/NCSU_FreePDK_45nm.capTbl
set init_gnd_net {VSS}
set init_pwr_net {VDD}
```

After the design import a series of step is followed, all of them reported below.

- **Floorplan structuring**
- **Power Rings Insertion**
- **Standard cell power routing**
- **Placement**
- **Post Clock-Tree-Synthesis (CTS) optimization**
- **Filler Placement**
- **Routing**
- **Post routing optimization**

The final result is shown by fig.3.6

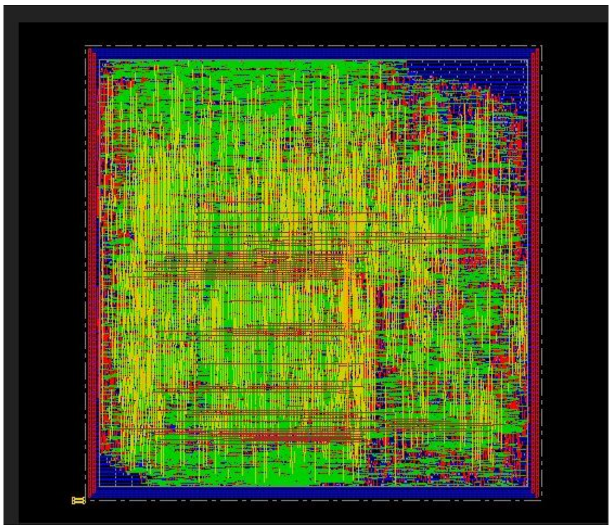


Figure 3.6: Layout

3.6.1 Parasitics extraction

Here resistance and capacitance parasitic values are extracted for each wire. This information will be used by Innovus to analyze in more accurate way the time constraints.The extracted values of the RC capacitance is shown in the following:

```
RC Corner Indexes            0
Capacitance Scaling Factor   : 1.00000
Coupling Cap. Scaling Factor : 1.00000
Resistance Scaling Factor    : 1.00000
Clock Cap. Scaling Factor    : 1.00000
Clock Res. Scaling Factor    : 1.00000
Shrink Factor                : 1.00000
```

```
Initializing multi-corner capacitance tables ...
Initializing multi-corner resistance tables ...
Checking LVS Completed (CPU Time= 0:00:00.1  MEM= 1120.4M)
Extracted 10.0015% (CPU Time= 0:00:00.4  MEM= 1189.4M)
Extracted 20.0019% (CPU Time= 0:00:00.5  MEM= 1189.4M)
Extracted 30.0013% (CPU Time= 0:00:00.5  MEM= 1189.4M)
Extracted 40.0017% (CPU Time= 0:00:00.6  MEM= 1189.4M)
Extracted 50.0022% (CPU Time= 0:00:00.7  MEM= 1189.4M)
Extracted 60.0015% (CPU Time= 0:00:00.8  MEM= 1189.4M)
Extracted 70.0019% (CPU Time= 0:00:01.0  MEM= 1189.4M)
Extracted 80.0013% (CPU Time= 0:00:01.2  MEM= 1189.4M)
Extracted 90.0017% (CPU Time= 0:00:01.4  MEM= 1189.4M)
Extracted 100% (CPU Time= 0:00:01.9  MEM= 1193.4M)
Number of Extracted Resistors     : 141946
Number of Extracted Ground Cap.   : 148780
Number of Extracted Coupling Cap. : 245932
```

3.6.2 Verification of the setup and hold requirements

It is based on verify the respect of the timing constraints. The clock period has been set by the design compiler in the previous part as $f_M/4$ (where f_M is the maximum operating clock frequency achievable by the system). After running the Timing Analysis and generate the reports, it has been verified that **the setup and hold requirements are met**: every slack computed **is positive**.

3.6.3 Connectivity and Design rules verification and Geometry

The last check that has to be made is the verification of connectivity and design rules. The **Connectivity** verification produces **no violations**, this means that there are no errors like floating wires. The same thing holds for **Geometry**, this means that there are no errors like wrong constraints on the geometric feature during the place and route design flow. Finally, the post place and route verilog netlist and the .sdf file with delay annotation are saved.

3.6.4 Modelsim simulation and swithing activity recording

To simulate the circuit taking into account the Place&Route operation, the produced netlist must be compiled. This operation is done with the following commands:

```
#!/bin/bash
cd /home/isa37/git/lab3/sim/
source /software/scripts/init_msim6.2g
rm -r work
vlib work
vcom -93 -work ./work ../src/riscv_pkg.vhd
vcom -93 -work ./work ../src/DATA_MEMORY.vhd
vcom -93 -work ./work ../src/INSTRUCTION_MEMORY.vhd
vcom -93 -work ./work ../tb/tb_output_data_checker.vhd

#Assuming Testbench file is tb_fir.v
#and testbench module is tb_fir
#Compile the verilog type:
```

```
vlog -work ./work ../innovus/RISCV_PROCESSOR_postPlaceAndRoute.v
vcom -93 -work ./work ../tb/tb_one.vhd

#link to Modelsim the compiled library of the cells
vsim -L /software/dk/nangate45/verilog/msim6.2g work.tb_one
#link the delay file
vsim -L /software/dk/nangate45/verilog/msim6.2g -sdftyp /tb_one/UUT=../innovus/RISCV_PROCESSOR_postPlac

#create vcd file
vcd file ../vcd/design.vcd
#specify the signals to be monitored
vcd add /tb_one/UUT/*
```

After running the simulation, the activity information are written in the design.vcd file. Comparing the *result.csv* obtained post Place&Route and the precedent one it is possible to observe that the we obtain the same result and so there are no error.

3.7 Power consumption estimation

In this final step the *.vcd* file written by Modelsim is used by Innovus to estimate the power consumption of the processor. After loading it in Innovus and running the power analysis, the power report is generated. An offprint of it can be see in the following report.

Report 3.2: Extract of the Power report produced by Innovus

```
*       Power Units = 1mW
*
*       Time Units = 1e-09 secs
*
*       report_power -outfile powerReport/power_report_no_abs.txt -sort total -hierarchy all -cell_type all -cloc
*
-----------------------------------------------------------------------------------------------------

Group                                Internal    Switching      Leakage       Total   Percentage
                                     Power       Power          Power         Power   (%)
-----------------------------------------------------------------------------------------------------
Sequential                             0.7815      0.08186       0.1443       1.008        59.69
Macro                                       0            0            0           0            0
IO                                          0            0            0           0            0
Combinational                          0.1905       0.2516        0.107       0.549        32.52
Clock (Combinational)                       0       0.1316    2.871e-05      0.1316        7.797
Clock (Sequential)                          0            0            0           0            0
-----------------------------------------------------------------------------------------------------
Total                                   0.972        0.465       0.2513       1.688          100
-----------------------------------------------------------------------------------------------------
```

From the table it is possible to observe that the main contribution of the power consumption is related to the sequential element due to the high number of register involved to make pipeline the processor. The total power consumption is of $1.69mW$.

CHAPTER 4

Risc-V with ABS function

To compute the absolute value the following pseudo-code has to be executed in Hardware:

```
if(input>0)
    abs_value=input
else
    abs_value=-input
```

In order to be executed in the risc-v, the operation needs its own op-code. It has been choosen to map the operation on the same op-code of SRL instruction in order to easily extract the binary instructions using $rars1\_3\_1$.

31-25	24-20	19-15	14-12	11-7	6-0	
0000000	00000	rs1	101	rd	0110011	ABS

Table 4.1: ABS instruction format

The assembly code has been modified in order to execute our new instruction. Previously the absolute value was computed executing:

```
srai x9,x8,31       # apply shift to get sign mask in x9
xor x10,x8,x9       # x10 = sign(x8)^x8
andi x9,x9,0x1      # x9 &= 0x1 (carry in)
add x10,x10,x9      # x10 += x9 (add the carry in)
```

In the new code the instructions above has been substituted with the instruction:

```
srl x10,x8,x0       #x10 = abs(x8)
```

Note: The contents discussed in this chapter are referred to the branch *risc-V-abs-Branch_handle* of the github repository.

4.1 Hardware modifications

To support the execution of the ABS instruction, some additional features has been introduced in the RISC-V components.

Modifications in the *Control* unit

The Control unit is now able to recognize the ABS instruction and set the appropriate control signals for the datapath and the RF. This is shown in the following table, which is an extension to table 2.1.

INPUT	*OUTPUT*						
OP Code	**BRANCH**	**REGWRITE**	**ALUSrc**	**ALUOP**	**MEMWRITE**	**MemRead**	**WDataMux**
0110011	0	1	0	OP	0	0	ALU

Table 4.2: CU control signals to support the ABS function

Modifications in the *ALU Control* unit

The ALU Control, checking at the the *ALUOP* signal coming from the control unit and the *funct3* bits from the in struction, is able to inform the ALU about the instruction that needs to be performed. This is done according to the following table, which is an extension to table 2.2.

ISA INSTRUCTION	*INPUT*		*OUTPUT*
	AluOp	**funct3**	**Alu Operation**
ABS	OP	101	ALU_ABS

Table 4.3: AluCommand to support the ABS function

Modifications in the ALU

The new functionality has been implemented at high level, performing the 2's complement inversion in VHDL as follows:

```
when ALU_ABS=>
    if operand1(31)='0' then    --Check if the the operand is positive
        tmp_sum<=operand1;    --If positive, do nothing
    else
        tmp_sum<= std_logic_vector(signed(not(operand1)) + 1); --if negative, 2'sC inversion
    end if;
```

4.2 System Simulation

Following the same procedure addressed in section 3.3, the new system has been tested ad HDL level. This time the code in section 6.3 (**minv-rv-abs-rescheduling.s**) is run. Notice that this is the shortest code since all the instructions which calculated the absolute value has been replaced by a single instruction.
The result obtained by simulation is the same as in Report 3.1, thus we can say that the system works correctly.

4.3 Logic Synthesis

Repeating the same step seen in chap.2.3, the synthesis of the architecture which implement also the ABS instruction has these features:

- **Min Clock Period** 3.05 ns
- **Area** 15479.07 μm^2

Thus, we obtain the same result in terms of minimum clock period while the area increases.
The clock period does not change because the logic added in EX stage to implement ABS operation does not add further delay to the critical path being it in parallel with ALU.
Since further logic is introduced, the area is slightly increased.

Follwing the same procedure shown in section 3.5, the **netlist** is extracted and **simulated**. The file *results.csv* that is produced this time is shown in Report 4.1.

Report 4.1: *results.csv* using the instruction ABS

```
Mem Type, Address, Data
RF: 16, 7
RF: 4, 268500996
RF: 4, 268500992
RF: 5, 268501004
RF: 5, 268501020
RF: 13, 1073741824
RF: 13, 1073741823
RF: 8, 10
RF: 10, 10
RF: 11, 1
RF: 4, 268500996
RF: 16, 6
RF: 13, 10
RF: 8, -47
RF: 10, 47
RF: 11, 0
RF: 4, 268501000
RF: 16, 5
RF: 8, 22
RF: 10, 22
RF: 11, 0
RF: 4, 268501004
RF: 16, 4
RF: 8, -3
RF: 10, 3
RF: 11, 1
RF: 4, 268501008
RF: 16, 3
RF: 13, 3
RF: 8, 15
RF: 10, 15
RF: 11, 0
RF: 4, 268501012
RF: 16, 2
RF: 8, 27
RF: 10, 27
RF: 11, 0
RF: 4, 268501016
RF: 16, 1
RF: 8, -4
RF: 10, 4
RF: 11, 0
RF: 4, 268501020
RF: 16, 0
DM: 268501020, 3
```

The report shows that the system works coherently with the instruction flow, so the system works correctly. Notice that the number of operations on the RF are reduced with respect to report 3.1. This is because sequences of steps like

```
RF: 8, -47
RF:  9 ,  -1
RF:  9 ,  -1
RF:  10 , 46
```

```
    RF:  9 ,  1
    RF:  10 ,  47
```

has been replaced by

```
    RF:  8 ,-47
    RF:  10 ,  47
```

because the absolute value is computed just by one instruction.

4.4 Place&Route

Following the same step seen in the section 3.6, it is possible obtain the following layout
The final result is shown by fig.4.1.

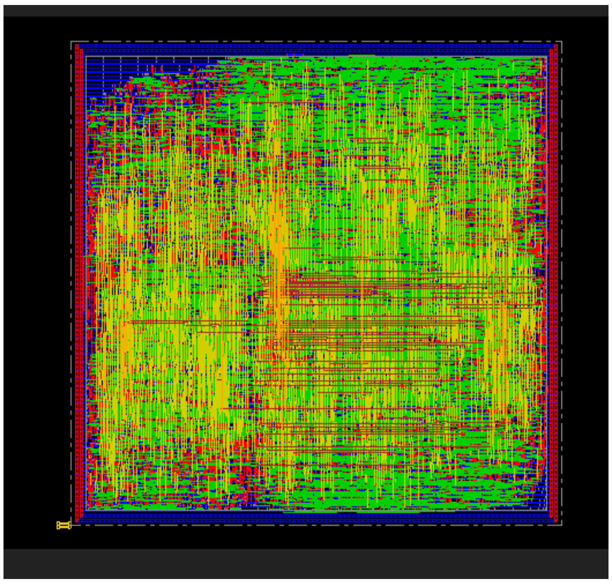


Figure 4.1: Layout

4.4.1 Parasitics extraction

The extracted values of the RC capacitance is shown in the following:

```
RC Corner Indexes            0
Capacitance Scaling Factor   : 1.00000
Coupling Cap. Scaling Factor : 1.00000
Resistance Scaling Factor    : 1.00000
Clock Cap. Scaling Factor    : 1.00000
Clock Res. Scaling Factor    : 1.00000
Shrink Factor                : 1.00000
Initializing multi-corner capacitance tables ...
Initializing multi-corner resistance tables ...
Checking LVS Completed (CPU Time= 0:00:00.1  MEM= 1130.8M)
Extracted 10.0014% (CPU Time= 0:00:00.4  MEM= 1199.8M)
Extracted 20.0017% (CPU Time= 0:00:00.5  MEM= 1199.8M)
Extracted 30.002% (CPU Time= 0:00:00.5  MEM= 1199.8M)
Extracted 40.0013% (CPU Time= 0:00:00.7  MEM= 1199.8M)
Extracted 50.0016% (CPU Time= 0:00:00.8  MEM= 1199.8M)
Extracted 60.0019% (CPU Time= 0:00:00.9  MEM= 1199.8M)
Extracted 70.0012% (CPU Time= 0:00:01.0  MEM= 1199.8M)
Extracted 80.0015% (CPU Time= 0:00:01.2  MEM= 1199.8M)
Extracted 90.0018% (CPU Time= 0:00:01.4  MEM= 1199.8M)
Extracted 100% (CPU Time= 0:00:02.0  MEM= 1203.8M)
Number of Extracted Resistors     : 144135
Number of Extracted Ground Cap.   : 151054
Number of Extracted Coupling Cap. : 250496
```

4.4.2 Verification of the setup and hold requirements

The clock period has been set by the design compiler in the previous part as $f_M/4$ (where f_M is the maximum operating clock frequency achievable by the system). After running the Timing Analysis and generate the reports, it has been verified that **the setup and hold requirements are met**: every slack computed **is positive**.

4.4.3 Connectivity and Design rules verification and Geometry

The **Connectivity** verification produces **no violations**, this means that there are no errors like floating wires.
The same thing holds for **Geometry**, this means that there are no errors like wrong constraints on the geometric feature during the place and route design flow. Finally, the post place and route verilog netlist and the .sdf file with delay annotation are saved.

4.4.4 Modelsim simulation and switching activity recording

To simulate the circuit taking into account the Place&Route operation, the produced **netlist** must be extracted. After running the simulation, the activity information are written in the design.vcd file.

Comparing the *result.csv* obtained post-Place&Route and the precedent one (Report 4.1) it is possible to observe that the we obtain the same results and so there are no errors.

4.5 Power consumption estimation

In this final step the *.vcd* file written by Modelsim is used by Innovus to estimate the power consumption of the processor. After loading it in Innovus and running the power analysis, the power report is generated. An offprint of it can be see in the following report.

Report 4.2: Extract of the Power report produced by Innovus

```
*       Power Units = 1mW
*
*       Time Units = 1e-09 secs
*
*       report_power -outfile powerReport/power_report_abs_branch_handle.txt -sort total -hierarchy all -cell_typ
*
------------------------------------------------------------------------------------------------

Group                               Internal   Switching    Leakage     Total  Percentage
                                    Power      Power        Power       Power  (%)
------------------------------------------------------------------------------------------------
Sequential                            0.7566     0.07465     0.1441    0.9753       58.76
```

```
Macro                               0          0          0          0          0
IO                                  0          0          0          0          0
Combinational                  0.1938     0.2517     0.1101     0.5557      33.48
Clock (Combinational)               0     0.1288  2.871e-05     0.1289      7.763
Clock (Sequential)                  0          0          0          0          0
-----------------------------------------------------------------------------------
Total                          0.9504     0.4552     0.2543       1.66        100
-----------------------------------------------------------------------------------
```

From the table it is possible to observe that the main contribution of the power consumption is related to the sequential element due to the high number of register involved to make the processor pipelined. The total power consumption is of $1.66mW$.

4.6 RISC-V with ABS function, without Branch Handling

Note: The contents discussed in this section are referred to the branch *risc-V-abs-No_branch_handle* of the github repository.
For the sake of completeness, the whole process (vhdl design, hdl simulation, syhtesis) has been performed for a version of the RISC-V **provided** with the **ABS** unit, but **without** the capability to manage branch instructions efficiently.
The code for this kind of architecture is shown in section 6.5. Notice that all the instructions which calculated the absolute value has been replaced by a single instruction, but the NOP instructions are still present.
The aim of this last step is to provide another alternative to the "basic" version of the processor synthesized in section 2.3 and to check if there could be any saving in terms of area. The VHDL implementation is obtained simply by adding the ABS instruction to the basic version of the processor presented in chapter 2.
The synthesis of the architecture brings the following results:

- **Min Clock Period** 3.05 ns

- **Area** 15462.04 μm^2

The comparison between the different architectures is done in chapter 5, with some comments about the obtained results.

CHAPTER 5

Comparison between different architectures

5.1 Speed and Area

In this laboratory experience, varying the two different parameters (Branch Handling, ABS instruction support), four different architectures have been designed, with different results in terms of speed and cost. In table 5.1 Synopsis evaluation is reported for the four different implementations.

RISC-V	T_crit	f_ck	Area (μm^2)
Base (No ABS, No Branch handling)	3.08 ns	324,67 MHz	15298.99
Base + Branch handling	3.05 ns	327,87 MHz	15338.09
ABS	3.05 ns	327,87 MHz	15462.04
ABS + Branch handling	3.05 ns	327,87 MHz	15479.07

Table 5.1: Synopsis evaluation for the four different implementations

According to table 5.1, the four different architectures achieve all almost the same speed in terms of **clock frequency**, except for the first version of the architecture.
This does not mean that the execution time for the algorithms is the same. Considering the code on which the tests were based, the fastest architecture is the one able to support both the ABS function and Branch Handling. This is because there are **less** time losses due to control hazards and there are **less** instructions needed to perform the Absolute value computation (1 vs 4).
If the code to be executed contains few branches and no absolute value computation, all the architectures could be similar in terms of execution time. Thus, the choice about the optimal architecture **depends** on the typical usage of the processor.
In terms of **Area**, an increasing of size can be seen while the complexity of the system grows, this is an understandable outcome. In particular, the maximum increase with respect to the smaller processor is of $\simeq 1\%$, so if the area constraints are not tight, the choice of the most efficient system (*ABS + Branch handling)* could be reasonable.

5.2 Power consumption

A comparison in terms of **power consumption** can be done between the two architectures on which the Place&Route has been performed. In table 5.2, the estimations provided by Innovus are reported.

	Power			
Architecture	Internal (mW)	Switching (mW)	Leakage (mW)	Total (mW)
Base + Branch handling	0.972	0.465	0.251	1.68
ABS + Branch handling	0.950	0.455	0.254	1.66

Table 5.2: Innovus power extimations

Even if the algorithm performed is the same, the code executed by the two processors is different (in section 6.3 vs 6.4), so the comparison is done only assuming the same task, not the same code. The different instructions required to complete the task bring to a slightly **reduced power consumption** ($\simeq$ 1%) for the **more complex** architecture. This means that, for this particular task, the fastest architecture leads to lower power consumption. This result, however, may vary with a different task to be achieved, so it **depends** on the particular usage of the processor.

CHAPTER 6

Appendix

6.1 minv-rv.s

Report 6.1: Code *minv-rv.s*

```
#################
# Basic VERSION
# This program takes an array v and computes
# min{|v[i]|}, the minimum of the absolute value,
# where v[i] is the i-th element in the array
        .data
        .align  2
v:
        .word   10
        .word   -47
        .word   22
        .word   -3
        .word   15
        .word   27
        .word   -4
m:
        .word   0

        .text
        .align  2
        .globl  __start
__start:
        li x16,7            # put 7 in x16
        la x4,v             # put in x4 the address of v
        la x5,m             # put in x5 the address of m
        li x13,0x3fffffff   # init x13 with max pos (maximum positive number)
loop:
        beq x16,x0,done     # check all elements have been tested
        lw x8,0(x4)         # load new element in x8
        srai x9,x8,31       # apply shift to get sign mask in x9
        xor x10,x8,x9       # x10 = sign(x8)^x8
        andi x9,x9,0x1      # x9 &= 0x1 (carry in)
        add x10,x10,x9      # x10 += x9 (add the carry in)
        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
done:
        sw x13,0(x5)        # store the result
endc:
        jal endc            # infinite loop
        addi x0,x0,0
```

6.2 minv-rv-NOPs.s

The code shown in section is the one executed on the first version of the processor (*Base*). The NOPs are needed to face the control hazards.

Report 6.2: Code *minv-rv-NOPs.s*

```
#################
# Basic VERSION
# This program takes an array v and computes
# min{|v[i]|}, the minimum of the absolute value,
# where v[i] is the i-th element in the array
        .data
        .align  2
v:
        .word   10
        .word   -47
        .word   22
        .word   -3
        .word   15
        .word   27
        .word   -4
m:
        .word   0

        .text
        .align  2
        .globl  __start
__start:
        li x16,7            # put 7 in x16
        la x4,v             # put in x4 the address of v
        la x5,m             # put in x5 the address of m
        li x13,0x3fffffff   # init x13 with max pos
loop:
        beq x16,x0,done     # check all elements have been tested
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        lw x8,0(x4)         # load new element in x8
        srai x9,x8,31       # apply shift to get sign mask in x9
        xor x10,x8,x9       # x10 = sign(x8)^x8
        andi x9,x9,0x1      # x9 &= 0x1 (carry in)
        add x10,x10,x9      # x10 += x9 (add the carry in)
        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
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        add x13,x10,x0      # update min
        jal loop            # next element
        addi x0,x0,0        # NOP (jal control hazard)
        addi x0,x0,0        # NOP (jal control hazard)
        addi x0,x0,0        # NOP (jal control hazard)
done:
        sw x13,0(x5)        # store the result
endc:
        jal endc            # infinite loop
        addi x0,x0,0
        addi x0,x0,0    # NOP (jal control hazard)
        addi x0,x0,0    # NOP (jal control hazard)
```

6.3 minv-rv-rescheduling.s

Since in the second version of the RISC-V-lite the Branch are handled with the "Branch Taken" technique, the NOPs are no longer needed.
The rescheduling is necessary to avoid data hazards which involves branch instructions, since the forwarding unit is ineffective in that case due to the Branch calculation anticipation.

Report 6.3: Code *minv-rv-rescheduling.s*

```
#################
# Basic VERSION
# This program takes an array v and computes
# min{|v[i]|}, the minimum of the absolute value,
# where v[i] is the i-th element in the array
        .data
        .align  2
v:
        .word   10
        .word   -47
        .word   22
        .word   -3
        .word   15
        .word   27
        .word   -4
m:
        .word   0

        .text
        .align  2
        .globl  __start
__start:
        li x16,7             # put 7 in x16
        la x4,v              # put in x4 the address of v
        la x5,m              # put in x5 the address of m
        li x13,0x3fffffff    # init x13 with max pos (maximum positive number)
loop:
        beq x16,x0,done      # check all elements have been tested
        lw x8,0(x4)          # load new element in x8
        srai x9,x8,31        # apply shift to get sign mask in x9
        xor x10,x8,x9        # x10 = sign(x8)^x8
        andi x9,x9,0x1       # x9 &= 0x1 (carry in)
        add x10,x10,x9       # x10 += x9 (add the carry in)
        slt x11,x10,x13      # x11 = (x10 < x13) ? 1 : 0
        addi x4,x4,0x4       # point to next element
        addi x16,x16,-1      # decrease x16 by 1
        beq x11,x0,loop      # next element
        add x13,x10,x0       # update min
        jal loop             # next element
done:
        sw x13,0(x5)         # store the result
endc:
        jal endc             # infinite loop
        addi x0,x0,0
```

6.4 minv-rv-abs-rescheduling.s

This code is the one supported by the advanced version of the RISC-V-lite, which supports both the ABS function and the Branch Handling.

Report 6.4: Code *minv-rv-abs-rescheduling.s*

```
##################
# Basic VERSION
# This program takes an array v and computes
# min{|v[i]|}, the minimum of the absolute value,
# where v[i] is the i-th element in the array
        .data
        .align  2
v:
        .word   10
        .word   -47
        .word   22
        .word   -3
        .word   15
        .word   27
        .word   -4
m:
        .word   0

        .text
        .align  2
        .globl  __start
__start:
        li x16,7            # put 7 in x16
        la x4,v             # put in x4 the address of v
        la x5,m             # put in x5 the address of m
        li x13,0x3fffffff   # init x13 with max pos
loop:
        beq x16,x0,done     # check all elements have been tested
        lw x8,0(x4)         # load new element in x8
        srl x10,x8, x0      # abs x10, x8
        slt x11,x10,x13     # x11 = (x10 < x13) ? 1 : 0
        addi x4,x4,0x4      # point to next element
        addi x16,x16,-1     # decrease x16 by 1
        beq x11,x0,loop     # next element
        add x13,x10,x0      # update min
        jal loop            # next element
done:
        sw x13,0(x5)        # store the result
endc:
        jal endc            # infinite loop
        addi x0,x0,0
```

6.5 minv-rv-abs-NOPs.s

Report 6.5: Code *minv-rv-abs-NOPs.s*

```
#################
# Basic VERSION
# This program takes an array v and computes
# min{|v[i]|}, the minimum of the absolute value,
# where v[i] is the i-th element in the array
        .data
        .align  2
v:
        .word   10
        .word   -47
        .word   22
        .word   -3
        .word   15
        .word   27
        .word   -4
m:
        .word   0

        .text
        .align  2
        .globl  __start
__start:
        li x16,7            # put 7 in x16
        la x4,v             # put in x4 the address of v
        la x5,m             # put in x5 the address of m
        li x13,0x3fffffff   # init x13 with max pos
loop:
        beq x16,x0,done     # check all elements have been tested
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        lw x8,0(x4)         # load new element in x8
#       srai x9,x8,31       # apply shift to get sign mask in x9
#       xor x10,x8,x9       # x10 = sign(x8)^x8
#       andi x9,x9,0x1      # x9 &= 0x1 (carry in)
        srl x10,x8, x0      # abs x10, x8
        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
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        addi x0,x0,0        # NOP (beq control hazard)
        add x13,x10,x0      # update min
        jal loop            # next element
        addi x0,x0,0        # NOP (jal control hazard)
        addi x0,x0,0        # NOP (jal control hazard)
        addi x0,x0,0        # NOP (jal control hazard)
done:
        sw x13,0(x5)        # store the result
endc:
        jal endc            # infinite loop
        addi x0,x0,0
        addi x0,x0,0    # NOP (jal control hazard)
        addi x0,x0,0    # NOP (jal control hazard)
```

6.6 elaborate_results.txt

Report 6.6: Log file *elaborate_results.txt*

```
Loading db file '/software/synopsys/syn_current_64.18/libraries/syn/gtech.db'
Loading db file '/software/synopsys/syn_current_64.18/libraries/syn/standard.sldb'
  Loading link library 'NangateOpenCellLibrary'
  Loading link library 'gtech'
Running PRESTO HDLC

Statistics for case statements in always block at line 135 in file
        '../src/RISCV_PROCESSOR.vhd'
===============================================
|           Line           |  full/ parallel  |
===============================================
|           137            |    auto/auto     |
|           150            |    auto/auto     |
===============================================
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'RISCV_PROCESSOR'.
Information: Building the design 'PROGRAM_COUNTER_MANAGER'. (HDL-193)
Presto compilation completed successfully.
Information: Building the design 'PIPE1_REG'. (HDL-193)

Inferred memory devices in process
        in routine PIPE1_REG line 16 in file
                '../src/PIPE1_REG.vhd'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        Q_reg        | Flip-flop |  96   |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.

[...]

Inferred memory devices in process
        in routine register_file_32x32 line 27 in file
                '../src/REGISTER_FILE.vhd'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|   reg_memory_reg    | Flip-flop | 1024  |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Statistics for MUX_OPs
==================================================================
|    block name/line      | Inputs | Outputs | # sel inputs |
==================================================================
| register_file_32x32/40  |   32   |   32    |      5       |
| register_file_32x32/41  |   32   |   32    |      5       |
==================================================================
Presto compilation completed successfully.

[...]

Inferred memory devices in process
        in routine PIPE2_REG line 16 in file
                '../src/PIPE2_REG.vhd'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        Q_reg        | Flip-flop |  189  |  Y  | N  | Y  | N  | N  | N  | N  |
|        Q_reg        | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.

[...]

Inferred memory devices in process
        in routine PIPE3_REG line 16 in file
                '../src/PIPE3_REG.vhd'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        Q_reg        | Flip-flop |  176  |  Y  | N  | Y  | N  | N  | N  | N  |
|        Q_reg        | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.
Information: Building the design 'PIPE4_REG'. (HDL-193)

Inferred memory devices in process
        in routine PIPE4_REG line 16 in file
                '../src/PIPE4_REG.vhd'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        Q_reg        | Flip-flop |  169  |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.

[...]

Inferred memory devices in process
        in routine REGN_EN_PRES_FP_N32_P4194304 line 14 in file
                '../src/REGN_PRES.vhd'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        Q_reg        | Flip-flop |  31   |  Y  | N  | Y  | N  | N  | N  | N  |
|        Q_reg        | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
===============================================================================
Presto compilation completed successfully.
1
```