Integrated Systems Architecture

Lab 3: design of a RISC-V-lite processor

Group 30:
Chisciotti Laura 274728
Fusto Federico 279925
Goti Gianluca 269825

Github repository: `https://github.com/Gotg3/ISA_Lab3.git`

A.Y. 20-21

Contents

1 Introduction

The goal of this Lab is to design in VHDL a RISC-V-lite processor with 5 pipeline stages, which has the purpose to compute first of all the absolute value of a specific vector and then to calculate the minimum value of this vector, which, after the absolute value, is composed only of positive values.
The implementation of this processor has started from the translation of an assembly program (minv-rv.s) in the machine language by means of RISC-V assembler and run-time simulator (RARS), from which it has been possible to understand the instructions flow and to identify better the values of the vector, that are: [10; -47; 22; -3; 15; 27; -4].
After implementing in VHDL the processor, this has been synthesized by means of Synopsys Design Compiler in order to obtain information about timing, area, resource and power; and the netlist has been simulated to verify the correct behaviour with respect to the RTL description.
Then, some instructions, used to implement the absolute value, have been substituted with a single new special instruction, *abs*, and the same previous procedure has been carried out in order to verify that everything was still working correctly.
As latter step, a Place & Route has been performed for both the implementations (with and without the *abs* instruction) and also in this case the information about timing, area and power have been extracted and the netlist has been simulated to verify the correct behaviour.

2 Instructions description

The RISC-V-lite processor, that has been designed, supports a subset of the whole RV32I. In this section, it will be analyzed in order to derive the very first architecture of the design.
The supported instructions are the following:

- Arithmetic: add, addi, auipc, lui
- Branches: beq
- Loads: lw
- Shifts: srai
- Logical: andi, xor
- Compare: slt
- Jump and Link: jal
- Stores: sw

2.1 Arithmetic

- **add**
  R-type instruction, it performs an addition between rs1 and rs2.
- **addi**
  I-type instruction, it adds the sign-extended 12-bit immediate to register rs1.
- **auipc**
  U-type instruction, it adds upper immediate to PC, it is used to build pc-relative addresses. This instruction forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then it places the result in register rd.
  It has been decided to execute this operation by means of the ALU, thus it has required to add the PC signal in the EX-stage to the input-set of the ALU.
- **lui**
  U-type instruction, it is used to build 32-bit constants. It places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.
  This instruction has required to bring the Immediate signal in the EX-stage to the Multiplexer in the WB-stage, in order to be saved inside the register file. The signal has been simply forwarded through the pipeline stages up to the WB one. This instruction does not need to use the ALU but, due to the datapath structure and to data hazards, it has been necessary to forward the extended immediate value through the ALU.

2.2 Branches

- **beq**
  SB-type instruction, the 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and multiplied by 2, that corresponds to a shift toward the left, and then this result is added to the address of the branch instruction to give the target address. This instruction takes the branch if registers rs1 and rs2 are equal.
  Thus, two components that work in parallel are placed, which are:
  - the AddSum block, that has the task to compute the Target Address (TA), which corresponds to the address where the instruction has to jump if the branch is taken;
  - the ALU block, that compares the content of rs1 and rs2 and based on that produces the jump bit "Z".

2.3 Loads

- **lw**
  I-type instruction, it loads a 32-bit value from memory into rd. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset using the ALU.

2.4 Shifts

- **srai**
  I-type instruction, it performs an arithmetic right shift according to the encoded value of the shift "shamt" (5-bit). The right shift operation type is encoded in bit30.

2.5 Logical

- **andi**
  I-type instruction, it is a logical operation that performs bitwise AND on register rs1 and the sign-extended 12-bit immediate.

- **xor**
  I-type instruction, it is a logical operation that performs bitwise XOR on register rs1 and rs2.

2.6 Compare

- **slt**
  R-type instruction, it performs a signed comparison between rs1 and rs2, writing 1 to rd if rs1<rs2.

2.7 Jump and Link

- **jal**
  J-type instruction, in which the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jal stores the address of the instruction following the jump (pc+4) into register rd and for this reason, it has been necessary to forward the output signal of the Add component in the IF-stage to the final Multiplexer in the WB-stage. The TA computation is done by means of the AddSum component in the EX-stage.

2.8 Stores

- **sw**
  S-type instruction, it stores the 32-bit value of rs2 to memory. TA is obtained by adding register rs1 to the sign-extended 12-bit by means of AddSum component.

In Table 1 all the major building blocks involved in the instructions are reported.

Table 1: Resource requirements

Operation	funct3	Opcode	type	ALU	Imm. gen.	AddSum
ADD	000	0110011	R	✓		
ADDI	000	0010011	I	✓	✓	
AUIPC	/	0010111	U		✓	
LUI	/	0110111	U	✓	✓	
BEQ	000	1100011	B	✓	✓	✓
LW	010	0000011	I	✓	✓	
SRAI	101	0010011	I	✓	✓	
ANDI	111	0010011	I	✓	✓	
XOR	100	0110011	R	✓		
SLT	010	0110011	R	✓		
JAL	/	1101111	J		✓	✓
SW	010	0100011	S	✓	✓	

3 Top entity RISC-V

The processor RISC-V has been implemented with a pipeline architecture in order to improve the performance. This architecture is reported in black in Figure 1 and in addition in this picture in red two memories are present: *Instruction memory*, where the instructions are contained, and *Data memory*, where the data are stored. The memories are not part of the processor, but these are located on the outside of this one.
In particular the processor (black part) is composed of five pipeline stages: **IF** stage, **ID** stage, **EX** stage, **MEM** stage and **WB** stage, that are separated by some pipeline registers, which are: **IF/ID** (between IF stage and ID stage), **ID/EX** (between ID stage and EX stage), **EX/MEM** (between EX stage and MEM stage) and **MEM/WB** (between MEM stage and WB stage).

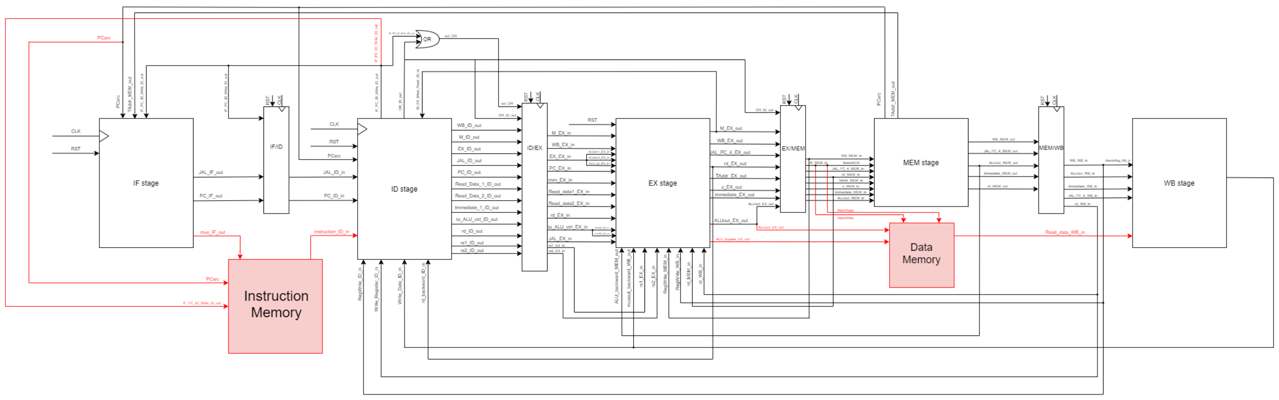

Figure 1: Top entity RISCV

Thus, an instruction is distributed along five cycles with the advantage of having the possibility to start the execution of a new instruction before completing the execution of the current instruction.
Even though in this architecture the latency does not have any advantage with respect to the single cycle execution, the throughput, which is the number instructions that are executed per unit time, has a significant advantage with respect to the single cycle execution. Thus this approach is able to speed up the execution and actually at new clock cycle, this processor is able to start a new instruction, provided that any hazard is occurring.

4 Base Version

In the first part of the Lab, the target was to design the complete architecture of the RISC-V processor and to execute a specific ASM code able to apply the absolute value to a given array of numbers and then write inside the Data Memory the minimum one.
Below an in-depth analysis of each stage is reported in order to understand better how each block has been designed and its task.

4.1 IF stage

The instruction fetch stage has the task to select the next instruction to be executed. This choice is dependent by the signal generated by the HDU (IF_PC_ID_Write_ID_out), which has the task to stall the current instruction if an hazard occurs, and the signal PCsrc, used when there are instructions that change the sequential execution of the code like : JAL (jump and link), which is used to execute call to subroutine and BEQ (branch if equal), which is a conditioned jump. A register called program counter (PC) stores the instruction address used to access the instruction memory (IM).
The behaviour of the PC based on the IF_PC_ID_Write_ID_out and the PCsrc signals is reported in Section 4.6.
As reported in Figure 2, several control signals are received at this stage:

- PC_src: used to switch between sequential address and branch instruction address. It is driven by a CU signal (BranchCtrl), when a branch jump is validated, and the Z flag generated by the ALU;
- PC_write: it enables writing the address in PC (that signal is the same generated by the HDU with the name IF_PC_ID_Write_ID_out).

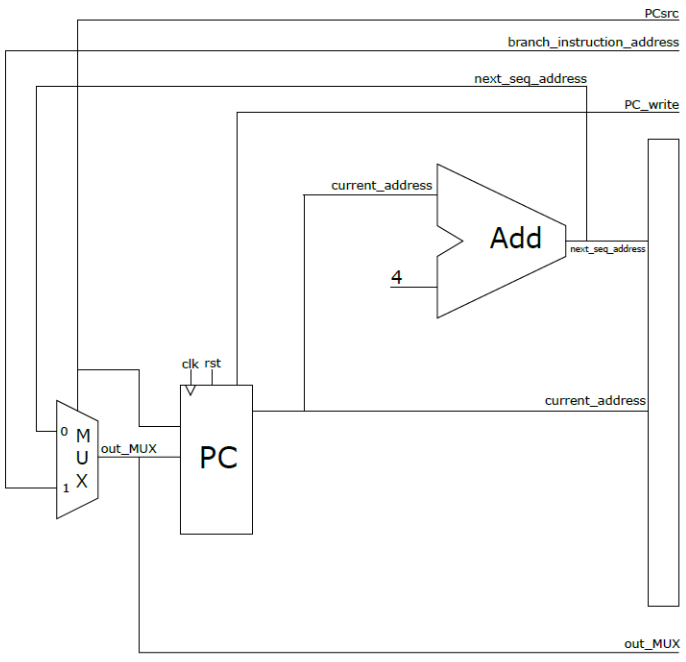

Figure 2: IF Stage data path

The output of the PC is used for two purposes:

- to compute next_sequential_address by adding 4 byte by means of a dedicated adder. This address will be propagated through all the pipeline registers to Write Back stage, where it will be used as return address for a JAL instruction.
- to compute current_address which will be used in Execute stage by AddSum to compute the target address for jump instructions.

This processor has an asynchronous reset, when it is switched on the PC is forced to point the first instruction of the code whose address is "0x00400000".

4.2 ID stage

This stage is the *Instruction Decode* (ID) and it is the stage where the instruction is decoded.
The architecture of this stage is reported in Figure 3 and it contains four important components that are the **register file**, the **hazard detection unit**, the **control unit** and the **immediate generator**. These are explained more in detail in the following parts.

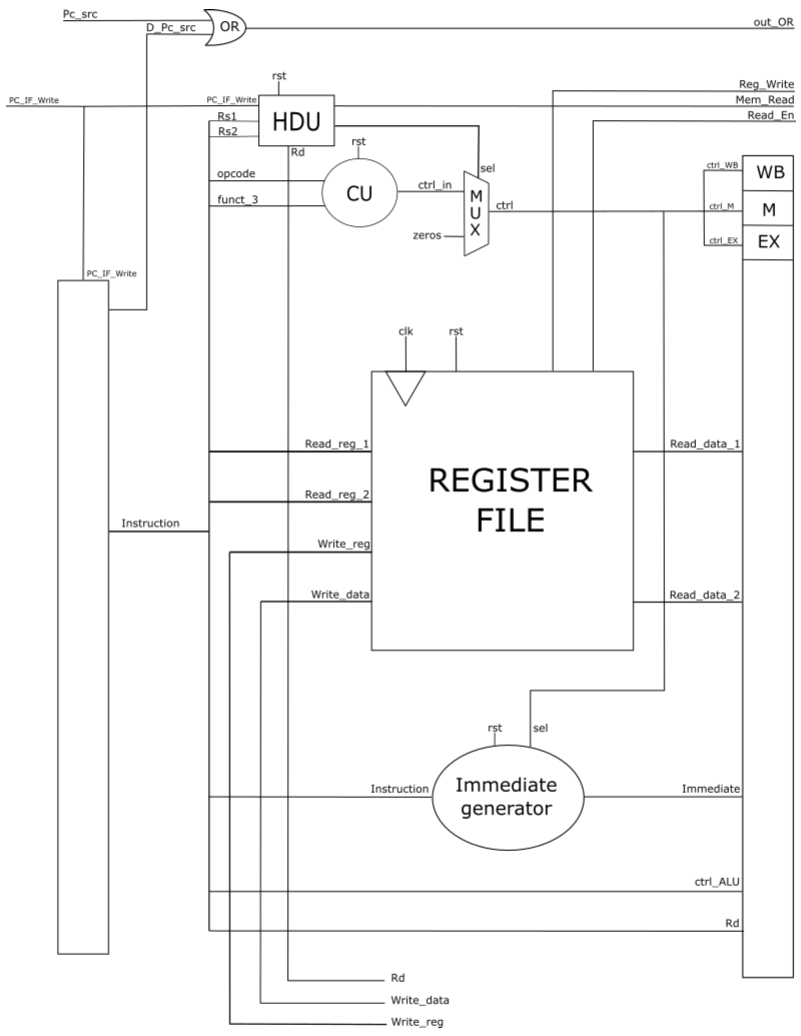

Figure 3: ID Stage data path

4.2.1 Register File

The register file (RF) is composed of 32 registers, where each register has 32 bits. This component has 8 inputs and 2 outputs, which are respectively:

- inputs: **clock**, **reset**, that is an asynchronous reset, **Read_reg_1** and **Read_reg_2**, which indicate both the address of the register where the processor wants to read the content, **RegWrite**, that is the enable for the writing operation, **read_en**, used to enable the reading, **Write_reg**, which contains the address where the processor wants to write the received data, and **Write_data**, that is the data transmitted from the WB stage to the ID stage and that the processor wants to write in the register pointed by the write register signal;
- outputs: **Read_data_1** and **Read_data_2**, that are the contents of the registers corresponding to the addresses indicated by the Read_reg_1 and Read_reg_2 signals. These are transmitted to the ID/EX pipeline register and then these signals are sent to the ALU component to be managed.

The RF has been designed to have an asynchronous reading and a synchronous writing, thus the register file is used as pipeline register for the write back operation.

Moreover, this component is shared between the ID stage and the WB stage, in fact it is appearing twice because in the first case it is used only as read source, while in the final stage (WB) it is used to store the result of the computation.
A characteristic of this register file is that by having multiple I/O ports, it can be read and written at the same time. Also, it is possible to read what is being written asynchronously. This particular behaviour has the task to avoid the structural hazard in the processor.

4.2.2 Hazard detection unit

The hazard detection unit (HDU) has the aim to avoid hazards that cannot be fixed by means of the forwarding unit.
The hazard is a critical condition that is able to block the execution of an instruction or to introduce some kind of error in the execution of the instruction. Moreover, the hazards can be divided into three groups: structural hazard, data hazard and control hazard. In this particular case, since the processor does not have high complexity, the type of hazard considered is the **data hazard**. This derives from the execution in pipeline of the instructions because of the fact that there is the necessity to reuse data after that they are generated.
The main way to avoid the data hazards is to use the forwarding unit, placed in the EX stage, but there are situations in which it is not able to completely solve these conflicts and thus the Hazard Detection Unit is used. The first operation done by this unit is to detect when a hazard cannot be fixed and this occurs when the signal **ID_EX_MemRead** (read enable for the Data memory), that comes from the EX stage is turned on and at the same time the address of the destination register (rd) is equal to the address of the source register 1 (rs1) or of the source register 2 (rs2). If these two conditions are satisfied, the hazard detection unit drives the selection bit to '1' thus replacing all control bits to '0'.
By replacing these control signals, it is possible either to stop the execution of a generic instruction or to transform a generic instruction into a NOP. The NOP instruction is represented by *addi* instruction that perform the sum of 0 to the content of X0 register: **addi X0,X0, 0x00000000**.
At the same time, when the two previous conditions are satisfied, the HDU has to reload from the instruction memory the same instruction again.
But there are special cases in which besides the stall condition even the branch condition has to be taken into account. This particular cases are analysed better in the next Section 4.6.

4.2.3 Control unit

The control unit (CU) provides all the control signals in order to drive the data paths.
To generate these signals the CU extracts the **opcode** and the **funct** fields from the instruction and thus provides the right value for each control signal.
The CU output cannot be directly connected to the input of the data path elements, but the control signals need to go through the pipeline registers in order to be shifted by the correct amount of time before taking action.
In particular the control signals are divided into 3 groups:

- control signal used in the ID stage: **immediate** (3 bits), which goes in the immediate generator in order to choose what is the proper extension for the immediate value. This signal does not go through a pipeline stage because it is used in the very same stage of the CU;

- control signals used in the EX stage: **ALU_op** (4 bits), that goes into the ALU Control, **ALUsrc1** (1 bit) and **ALUsrc2** (1 bit), which are the selectors of the multiplexers that feeds the ALU. These signals go through only one pipeline register, that is the ID/EX register;

- control signals used in the MEM stage: **BranchCtrl** (1 bit), that goes inside the AND gate in the MEM stage, this bit is a kind of flag for the branch operation. **MemWrite** (1 bit) and **MemRead** (1 bit), which are respectively the write enable and the read enable of the data memory. These signals go through two pipeline registers: ID/EX and EX/MEM;

- control signals used in the WB stage: **MemtoReg** (3 bits), that is the selector of the multiplexer in the WB stage , and **RegWrite** (1 bit), that is sent from the WB stage toward the ID stage, to go inside the register file as a write enable. These signals go through three pipeline registers (ID/EX, EX/MEM and MEM/WB) in order to arrive to the write back stage with the proper delay.

All the control signals at the output of the CU are sent to a multiplexer before being divided among the different stages. This mux has the task to choose between the control signals and the ones performing the NOP operation in case an hazard is detected by the HDU.

4.2.4 Immediate generator

The immediate generator is the component that generates the immediate value from the instruction.
The immediate value is a constant value which is inside the instruction code and the immediate generator block has the task to select the current instruction based on the signal **immediate** that arrives from the control unit and then it has to bring the specific bits from the instruction for composing the immediate value. Thus, this component has to extend the number of bits up to 32 bit, because the immediate value will be sent toward the ID/EX register to the ALU block and the AddSum block, which work with operands composed of 32 bits.
In Table 2, it is possible to observe more in detail how each instruction is manipulated to obtain the right immediate value.
In particular:

- **ADDI**, **LW** and **ANDI** are I-type instructions and to obtain the immediate value from these ones it is necessary to take the bits from 20 to 31 and then to extend these with a resize operation where the MSB, that represents the sign, is replicated to 20 times in this case in order to reach the 32 bit parallelism;
- **SRAI** is a special I-type instruction because inside this one there is an integer value called **shamt** (5 bit: from bit 20 to 24 of the instruction), that is directly extracted by the ALU from the instruction. In this case the immediate generator takes the instruction without managing it and sends this one integrally;
- **LUI** and **AUIPC** are U-type instructions and to obtain the immediate value, the bits from 12 to 31 have to be taken and then these quantities are filled with 12 zeros, called trailing zeros, on the right part in order to achieve the 32 bit parallelism;
- **JAL** is a J-type instruction and its immediate value is located in different fractions of the instruction, that have to be concatenated in order to obtain the correct immediate value. In particular, according to the document "RISCVGreenCard", to obtain the immediate value, the bits in position 31, from 12 to 19, 20 and from 21 to 30 have to be taken from the instruction and these correspond respectively to the bits 20, from 12 to 19, 11 and from 1 to 10 of the 20 bits of the immediate value. Then, even in this case, a sign extension has been applied to obtain an immediate value on 32 bit;
- **SW** is a S-type instruction and also in this case the immediate value is fractionated in different parts of the instruction, in particular from bit 25 to 31 and from bit 11 to 7, that correspond respectively to the bits from 11 to 5 and from 4 to 0 of the 12 bits of the immediate value. Thus, a sign extension has been used to reach a 32 bit parallelism;
- **BEQ** is a SB-type instruction and as the SW and the JAL even in this case the immediate value is put in different parts of the instruction and thus there is the need to recompose it taking the bits 31, 7, from 30 to 25 and from 11 to 8 of the instruction, that correspond respectively to the bits 12, 11, from 10 to 5 and from 4 to 1 of the 12 bits of the immediate value, concatenating these and doing a sign extension to obtain an immediate value represented on 32 bit;
- for all the other cases, the immediate is replaced with 32 zeros.

Table 2: Immediate generator content

Instruction	Format	Position	Immediate parallelism	Resyze type	Control signal (**immediate**)
ADDI	I	31-20	12	Sign extension	000
LW	I	31-20	12	Sign extension	000
SRAI	I (special)	31-0	32	/	101
ANDI	I	31-20	12	Sign extension	000

Instruction	Format	Position	Immediate parallelism	Resyze type	Control signal (**immediate**)
LUI	U	31-12	20	Fill with 0s	001
AUIPC	U	31-12	20	Fill with 0s	001
JAL	J	31/19-12/20/30-21	20	Sign extension	010
SW	S	31-25/11-7	12	Sign extension	011
BEQ	SB	31/7/30-25/11-8	12	Sign extension	100
when others	/	/	32	all 0s	111

4.3 EX stage

In this stage all the different instructions are executed, as it is possible to observe from Figure 4, this stage is composed mainly of the **ALU** and its special control unit which is the **ALUcontrol**, several **multiplexer**, the **Forwarding Unit** and the **AddSum** component.

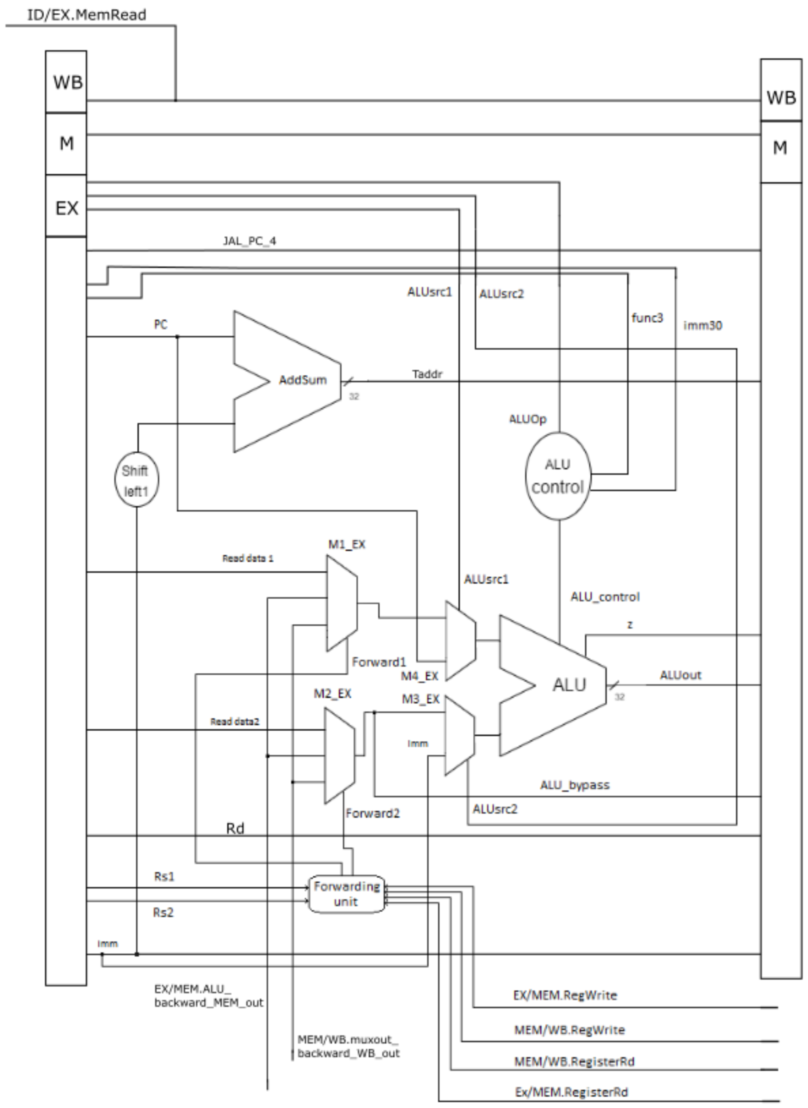

Figure 4: EX Stage data path

4.3.1 ALU

This component is the heart of the Execution Stage because it is responsible for all the arithmetic operations required by the different instructions.
The structure is organized in a multiplexer-like way, the parallelism is 32 bit since this specific version of RISC-V-LITE (RV32I) is capable to handle 32 bit data, all the operations (with the exception of the bitwise ones) are performed on signed numbers.
The ALU feeding network is composed by 4 multiplexers, two for each input. The first layer is necessary for Data Hazard handling and it is composed of two multiplexers 3x1, controlled by Forward1 and Forward2 signals, which are generated by the Forwarding Unit. With M1_EX and M2_EX it is possible to choose between three different inputs, one that is coming from the Register File and the two feedback data that are necessary to avoid Data dependencies and that are coming from MEM_Stage and WB_Stage. Once the proper input has been chosen in the first layer, this is forwarded to the second layer of multiplexers composed by two multiplexers 2x1 and controlled by signals ALUsrc1 and ALUsrc2 coming from the Control Unit. Each multiplexer has a different purpose, M4_EX, related to the first input of the ALU, is dedicated to the selection between the output of the previous mux and the PC. On the contrary, M3_EX is dedicated to the selection between the output of the previous multiplexer and the Immediate. This design solution has been adopted to have a simpler implementation of the feeding network and it also uses smaller multiplexers with a consequent save of space and power without any losses in terms of performance. In addition to that, this solution has led to a more efficient and easy control of the feeding of the ALU.
As it has been anticipated, the internal structure of this component is organized in a multiplexer-like way and the control signal ALU_ctrl provides the proper selection of the operation to perform. The special output "z" is necessary to decide either to perform a jump (z=1) or not (z=0). This flag is generated both during the execution of the JAL, independently on the operands (since it is an unconditional jump) and during the execution of the BEQ after the comparison of the operands rs1 and rs2 provided to the ALU. If this latter is the case, thus, if the content of the two source registers is equal, the signal z is set to 1, otherwise it assumes the value 0.
In the SRAI instruction, which is performing a right shift on the source operand, the integer value "shamt" (bits 24 to 20 inside the immediate field) is used to derive the correct amount of right shift to be applied to the operand. Shamt is extracted directly from the immediate which is provided by the second input. This solution has been preferred instead of adopting a special input since it reduces the number of connections inside the design and the number of ports of the component.
For the LUI instruction it has been necessary to forward the Immediate value inside the ALU in order for it to be available at the output since if a Data Hazard occurs the computed result can be immediately recycled to the input. In order to avoid the insertion of an additional mux (with the Immediate and the ALU output as inputs), this solution has been adopted.
In the following Table (Tab. 3), the complete encoding of the ALU is shown and therefore, for each instruction, the corresponding control code and the description of the operation are reported.

Table 3: ALU organization

Instruction	ALU_control (from ALU_ctrl)	Op. type
LW	0000	rs1+offset
ADDI	0001	rs1+imm
SRAI	0010	shift right
ANDI	0011	AND
AUIPC	0100	offset+PC
SW	0101	rs1+offset
ADD	0110	rs1+rs2
XOR	0111	XOR
SLT	1000	rs1<rs2
BEQ	1001	rs1=rs2
LUI	1011	Forward
JAL	When others	z=1

4.3.2 AluControl

This unit generates the command ALU_Control (4 bit) for the ALU starting from ALUop, imm30 and funct3 signals.
Funct3 (3 bit) is provided directly by the instruction and it selects the type of operation, ALUop (3 bit) is generated by the Control Unit, while imm30 is a single bit that specifies the SRAI operation, it has been used jointly to the other conditions to decode the proper command.
In Table 4, all the information about the ALU_control is reported, for each instruction every input and output are shown.

Table 4: ALUControl signals

Instruction	ALU_op (from CU)	funct3	imm30	ALU_control
LW	0000	010		0000
ADDI	0001	000		0001
SRAI	0001	101	✓	0010
ANDI	0001	111		0011
AUIPC	0010	/		0100
SW	0011	010		0101
ADD	0100	000		0110
XOR	0100	100		0111
SLT	0100	010		1000
BEQ	0101	000		1001
LUI	1000	/		1011
JAL	0111	/		When others

In this block, first of all the ALU_op signal is checked and then, based on the value of funct3, the proper control is generated. This solution has been necessary since the first signal selects the instruction "type" and then the second is used to perform the proper selection. In addition, for the SRAI isntruction, imm30 has been used to decode the proper control.
Moreover, in some instructions the field funct3 was not present, thus only the ALU_op signal has been used to generate the proper control.

4.3.3 AddSum

This component is dedicated to the computation of the Target Address (TA), it is nothing more that a simple plain 32-bit adder whose inputs are the Program Counter (Base Address) and the Immediate (Offset). The sum is performed in 2's complement since also backward jumps are possible. Before starting the summation of the two operands, the Immediate has to be multiplied by two since the Offset value is encoded in multiples of 2 bytes. This operations is carried out by means of a single left shift operation directly inside the component.
This address is provided to the pipeline stage EX/MEM and then it can be selected, based on the value of PCsrc as new address for the next cycle.

4.3.4 Forwarding_Unit

This unit is responsible to resolve a large part of the Data Hazard that can be present during the execution of the code. First of all, a Data Hazard occurs when two or multiple following instructions are trying to use a result that has not been written inside the register file yet. This situation can be easily resolved by means of forwarding, the idea is to reuse the computed result as inputs of the ALU. Two different situations are possible, the first one occurs when between two consecutive instructions i.e. i1 and i2, the second one (i2) uses as source register the destination register of the first one (i1). In the second case these two instructions are separated by an instruction that does not include a data dependency.
These situations can be identified by looking at the source registers of the second (or third) instruction and destination register of the first one.
In both cases the results are still in the pipeline registers, in the first one the result is in the EX/MEM register and in the second one the result is in MEM/WB register. Thus, the Forwarding mechanism is able to forward these results to the ALU in order to be reused without waiting their commit (WB phase).

This unit controls multiplexer M1_EX and M2_EX and its structure is organized as a decoder. In Table 5 it is possible to observe all the possible combinations for the multiplexers.

Table 5: Forwarding unit control values

Mux Control	Source
Forward1=00	ID/EX (from RF)
Forward1=01	EX/MEM
Forward1=10	MEM/WB
Forward2=00	ID/EX (from RF)
Forward2=01	EX/MEM
Forward2=10	MEM/WB

This unit is able to handle and fix two kinds of Data Hazard, the EX hazard and the MEM hazard. In the first case the unit has to forward a result that has been previously computed by the ALU. This hazard occurs if the previous instruction is going to write in the register file and the destination register is either the source register of ALU's input 1 or 2, thus provided that it is not zero, then move the multiplexer to pick the last value from the pipeline register EX/MEM.
A similar situation can be found for MEM hazard, in which a data, that has been taken from Data Memory or it is simply in the MEM stage, can be forwarded directly to the ALU. This happens if the instruction, that was in the MEM stage, was trying to write into the register file and the destination register matched the ALU's source registers, provided that it is not zero, then move the multiplexer to pick the last value from the pipeline register MEM/WB.
There is an additional consideration to take into account, which is when in the code several instructions are reading and writing the same register and this situation has to be handled by forwarding the result from the MEM/WB pipeline stage since it is the more recent result. Thus, in the handling of MEM hazards there is a sort of priority for the MEM/WB results. Therefore, the Forwarding Unit has to perform an additional check to verify if all the conditions of an EX hazard are occurring and if it is so and all the conditions for a MEM hazard are occurring too, the forwarding unit gives way to this latter.
All the hazards that are not possible to fix with this method are handled by means of the Hazard detection Unit.

4.4 MEM stage

The memory stage (MEM), which is reported in Figure 5, is composed of an AND gate which takes the zero flag generated by the ALU and the control bit BranchCtrl coming from the M register in the EX/MEM pipeline register. This flag is initially generated by the control unit and it indicates that the instruction which is being executed and that is in the memory stage, is a branch instruction like 'beq' or 'jal'.

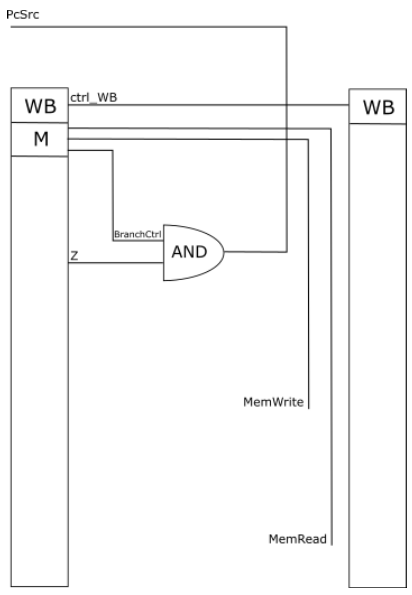

Figure 5: MEM stage data path

When there is a branch instruction (and all the conditions are true), the AND gate sets to 1 the output PCSrc, which is the selector of the mux in the IF stage. Thus the mux in the IF stage takes as input the output generated by the AddSum (Target Address) in the EX stage, obviously respecting the correct timing.
In this stage the control signals for the Data Memory are applied to it.
As previously said, this memory is not present in the architecture of the RISC-V because it is included inside the test bench.

4.5 WB stage

Write back is the stage where data is feedback into register file or in other stages in case of data dependencies that have to be fixed by forwarding unit.
As mentioned in previous sections, there are cases in which a command or an instruction has to be delayed by a certain amount of clock cycles in order to maintain the correct timing.
As for R-type instruction, the destination register R_d has to be accessed 3 clock cycles later with respect to when the source registers R_{s1} and R_{s2} are received in the ID stage, since the data, generated by the processing of the contents of the source registers, has to pass through 3 stages before being written in the register file. Thus, the destination address R_d is propagated through pipeline registers in order to be sent back correctly aligned with the data that has to be stored in the register file. As showed in Figure 6, WB stage essentially consists of a 4x1 multiplexer.

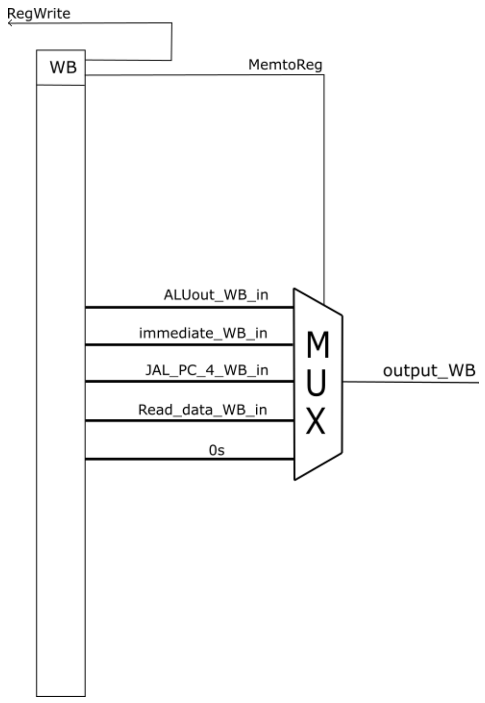

Figure 6: WB Stage data path

Received signals are:

- **ALUout_WB_in**: this is the value computed by ALU and provided to MEM/WB pipeline register. It is used to feed back data to be written in register file, as for R-type instruction;
- **immediate_WB_in**: it is used if LUI instruction (U-type) is being executed. This value is coming from the immediate generator without being changed, because the LUI contains already the right immediate value and the only operation that has to be done is to extend the number of bits;
- **JAL_PC_4_WB_in**: this value represents the next sequential address computed in IF stage. When a JAL instruction is being executed the return address has to be stored, so it is sent to WB that will write [PC] + 4 byte in the provided destination register address;
- **Read_data_WB_in**: this input is selected when the data is received from the data memory, which is located at the output of the RISC-V processor;
- 0s (when others): this last input is given to cover all the other cases.

Control signal in WB stage are:

- MemToReg: this is the selection signal used to drive the multiplexer.

MemToReg	Output
000	read data
001	jal
010	immediate
011	out_ALU
100	0s

- RegWrite: it is the control bit used to signal when destination register address can be accessed to write computed data.

Instruction	MemToReg	RegWrite
ADD	011	1
ADDI	011	1
AUIPC	011	1
LUI	010	1
BEQ	100	0
LW	000	1
SRAI	011	1
ANDI	011	1
XOR	011	1
SLT	011	1
JAL	001	1
SW	100	0
ABS	011	1

4.6 Stall and branch taken

To analyze different condition when there is the stall and when the branch is taken, it is necessary to look at the code to understand better how the instructions are processed.

```
Address     Code        Basic                          Source

start:
0x00400000  0x00700813  addi x16,x0,0x0000000723    li x16,7
0x00400004  0x0fc10217  auipc x4,0x0000fc10   24    la x4,v
0x00400008  0xffc20213  addi x4,x4,0xfffffffc
0x0040000c  0x0fc10297  auipc x5,0x0000fc10   25    la x5,m
0x00400010  0x01028293  addi x5,x5,0x00000010
0x00400014  0x400006b7  lui x13,0x00040000    26    li x13,0x3ffffff
0x00400018  0xfff68693  addi x13,x13,0xfffffff

loop:
0x0040001c  0x02080863  beq x16,x0,0x00000018 28    beq x16,x0,done
0x00400020  0x00022403  lw x8,0x00000000(x4)  29    lw x8,0(x4)
0x00400024  0x41f45493  srai x9,x8,0x0000001f 30    srai x9,x8,31
0x00400028  0x00944533  xor x10,x8,x9         31    xor x10,x8,x9
0x0040002c  0x0014f493  andi x9,x9,0x00000001 32    andi x9,x9,0x1
0x00400030  0x00950533  add x10,x10,x9        33    add x10,x10,x9
0x00400034  0x00420213  addi x4,x4,0x00000004 34    addi x4,x4,0x4
0x00400038  0xfff80813  addi x16,x16,0xffffffff35    addi x16,x16,-1
0x0040003c  0x00d525b3  slt x11,x10,x13       36    slt x11,x10,x13
0x00400040  0xfc058ee3  beq x11,x0,0xffffffee 37    beq x11,x0,loop
0x00400044  0x000506b3  add x13,x10,x0        38    add x13,x10,x0
0x00400048  0xfd5ff0ef  jal x1,0xffffffea     39    jal loop

done:
0x0040004c  0x00d2a023  sw x13,0x00000000(x5) 41    sw x13,0(x5)
```

```
endc:
0x00400050  0x000000ef  jal x1,0x00000000     43    jal endc
0x00400054  0x00000013  addi x0,x0,0x00000000 44    addi x0,x0,0
```

The first column here contains the addresses of the instructions to understand how these are placed inside the instruction memory. The second column has the machine language code corresponding to each instruction. In the third column, there is explicitly reported which instruction each code corresponds to and in the fourth column there are explicitly expressed the instructions used in the ASM code.
Thus, starting from this code, it is possible to analyse different choices that have been taken in the architecture of the RISC-V because of specific cases about the stall and the branch condition which are influenced by the fact that the processor is pipelined.
The different situations are:

- when the branch condition in the beq corresponding to the code '0xfc058ee3' is taken, there is the jump to the label 'loop' and thus at the instruction beq (0x02080863). In this last instruction, the branch is taken only when the processor is at the end of the computation and so there is the jump to the label 'done' and the RISC-V executes the store word (sw) instruction, placing the final result inside the data memory.
  It is necessary to take into account that the processor does not know immediately if the branch is taken or not, but it understands this condition only in the third cycle with respect to the moment when the fetch operation of the branch instruction is done and this occurs because the RISC-V is pipelined with 5 pipeline stages.
  The processor knows if the branch of the beq (0x02080863) is taken only in the MEM stage and this moment corresponds with a stall due to the data dependency condition between the lw and srai instructions. This coincidence of events would lead to the stall of the PCs (that are located in the IF stage and at the entry of the Instruction memory) and the loss of the sw instruction in the case of branch taken for the beq (0x02080863) instruction, causing a wrong behaviour of the processor. Therefore, to avoid this mistake, PCs have to depend both on **PCsrc** (signal asserted from the AND gate in the MEM stage to indicate if the branch is taken or untaken) and **IF_PC_ID_Write_ID_out** (signal that is asserted from the HDU when there is a data dependency that the forwarding unit cannot fix).
  In particular the PCs follow these conditions:
  - if PCsrc=1 and IF_PC_ID_Write_ID_out=1, there is an overlapping of the stall and jump condition. Thus in this case the stall is avoided and the jump is done in order to take the store word instruction and to store the correct value in the data memory, instead of stall the *srai* instruction in both the PCs;
  - if PCsrc=1 and IF_PC_ID_Write_ID_out=0, the behaviour is the normal one because only the jump condition is activated;
  - if PCsrc=0 and IF_PC_ID_Write_ID_out=1, the only condition that has to be satisfy is the stall condition by means of inserting a NOP and maintaining the stalled instruction in the PC for the next cycle;
  - if PCsrc=0 and IF_PC_ID_Write_ID_out=0, there is the simplest situation in which the behaviour is the normal one because there is not the need to stall or to jump.
- the processor understands if a branch is taken or untaken only in the third clock cycle with respect to the fetching, in particular, this choice is done in the MEM stage from the AND gate. Therefore, in the meantime, other instructions are started and in the case of the branch is taken there is the need to erase what has been processed for the instructions from when the branch instruction has been fetched to when the branch has been taken. Thus, to do that, an OR gate has been placed in the ID stage and in this one, the PCsrc and the delayed version of the PCsrc signal enter in order to reset with all zeros the registers in the ID/EX and EX/MEM registers and to delete what has been computed of the previous instruction.
  The PCsrc and its delayed version are placed as inputs of the OR gate because there is the need to reset the ID/EX and EX/MEM registers for two consecutive cycles, since when the branch is taken, there is an instruction which is being fetched in the IF stage, an instruction in the ID stage and even one in the EX stage. Thus, the first reset cycle has the task to delete what has been done by the instructions in the ID and EX stage, while the second reset cycle has the purpose to erase the instruction that was in the fetch stage and it is moved on the ID stage;

- the signal at the output of the OR described in the previous point does not go directly into all the registers of the ID/EX and EX/MEM registers, because there are some registers in the ID/EX register that have to have all zeros at the output even when there is a data dependency and the HDU has to simulate a NOP to stall the processor. Thus, to do that another OR gate has been used and this is reported Figure 1 of the top entity of the RISC-V. This has as inputs the output of the previous OR gate and the signal IF_PC_ID_Write_ID_out that is asserted when there is the need to stall the processor and its output arrives to the pipeline registers in the ID/EX stage referred to the signals: read_data_1, immediate, rd, rs1 and rs2; in order to that could have the output with all zeros both when there is the branch taken, when there is the stall condition and when there are these two conditions at the same time;
- the signal IF_PC_ID_Write_ID_out goes also inside the registers referred to the jal and pc signal in the IF/ID register, in order to set the output of these ones to all zeros when the HDU wants to stall the processor and thus the signal IF_PC_ID_Write_ID_out is asserted.

4.7 Instruction Memory

Since the instruction memory is designed to have synchronous write and read, a particular solution has been required to correctly drive the memory. As shown in Figure 2, instead of being addressed by PC, the IM is driven by the output of the mux (out_MUX) signal, so the PC stage is directly embedded and replicated into the memory. This choice was made to not introduce additional delay to signals. As in the previous case, the output pipeline register is simply the replica of the IF/ID pipeline register of the RISC-V architecture, even if in this case when the signal IF_ID_write signal is 1 the register is stalled instead of having all zeros at the output, as it is in the IF/ID pipeline register.
Memory arrays in RISC-V architectures are organized in rows and columns, and in the specific case of this memory, one row contains a single instruction. Particularly, words are accessed by 4 bytes increment w.r.t. address. In addition to that, addresses are given with an offset, so a decoding stage has been included in IM. It simply consists of a subtraction of the offset (0x00400000) and a division by 4, as the number of bytes needed for addressing words.
IM content is defined by the program executed with the RARS. However, some modification were required. Since the last instruction in the code is a JAL there is the need to insert NOP instructions to fill pipeline stages. This is required since the outcome of the jump is computed in MEM stage thus to give "space" to the jump instruction to be executed correctly. In Figure 7 the Instruction memory structure is shown.

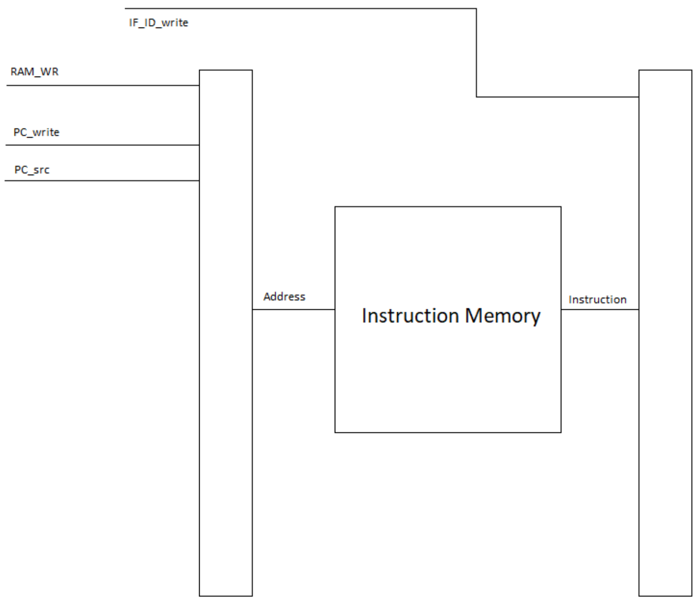

Figure 7: Instruction Memory

As it is possible to observe from the Figure 7, the two pipeline registers of the memory replicate the structure of the PC register and of the IF/ID register, thus they are provided with the same signals to ensure the same behaviour.

4.8 Data Memory

This memory has a similar structure to the previous one, since it has a synchronous read and write. As for Instruction Memory, also for Data Memory an address decoding stage has been required. The initial address 0x10010000 provided by RARS has been used as initial offset for the internal decoding stage. The content of this memory, which contains the values of the vector to be analyzed, has been generated by the RARS and stored in the proper locations.
The content generated by the RARS is the one reported below, which corresponds to the decimal values [10; -47; 22; -3; 15; 27; -4], encoded as:

```
0x0000000a 0xffffffd1 0x00000016 0xfffffffd 0x0000000f 0x0000001b 0xfffffffc 0x00000000
```

In Figure 8, the Data memory structure is shown.

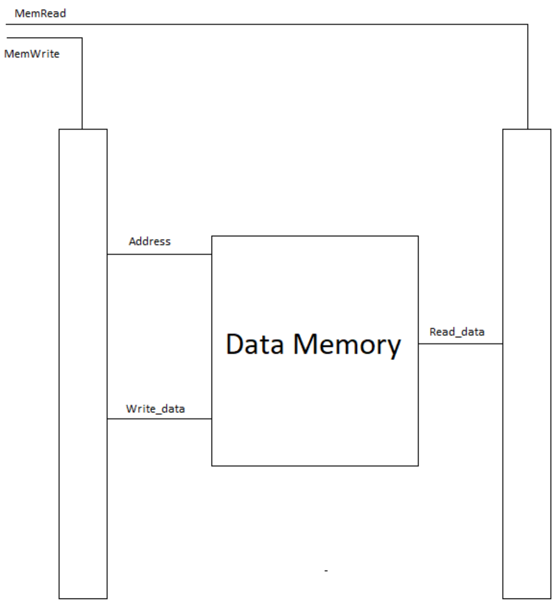

Figure 8: Data Memory

Enabling signals MemRead and MemWrite are generated by the CU and delayed by the pipeline registers of the RISC-V architecture, the parallelism of the Data is 32 bit as the one for the Addresses, that are generated by the ALU.

4.9 Simulation

To check the correctness of the structure, the design has been simulated using ModelSim. The execution of the ASM code has been compared with the one on the RARS simulator. The purpose of this code is to find the minimum value in an array, after applying the absolute value and writing it in the Data Memory after the elements of the array.
In Figure 9, the instant in which the minimum value of the array is written into the Data Memory is depicted. Therefore, the simulation has been performed adopting a clock period equal to T$_{ck}$=40 ns.
From the moment in which the store operation (00D2A023) is at the output of the IF/ID register, after 3 clock cycles the number 3, which is the minimum value computed after calculating the absolute value of the numbers contained in the Data Memory, is stored in this one and this is due to the fact that the data has to pass through the pipeline registers ID/EX and EX/MEM and then at the next clock cycle it is written inside the memory.

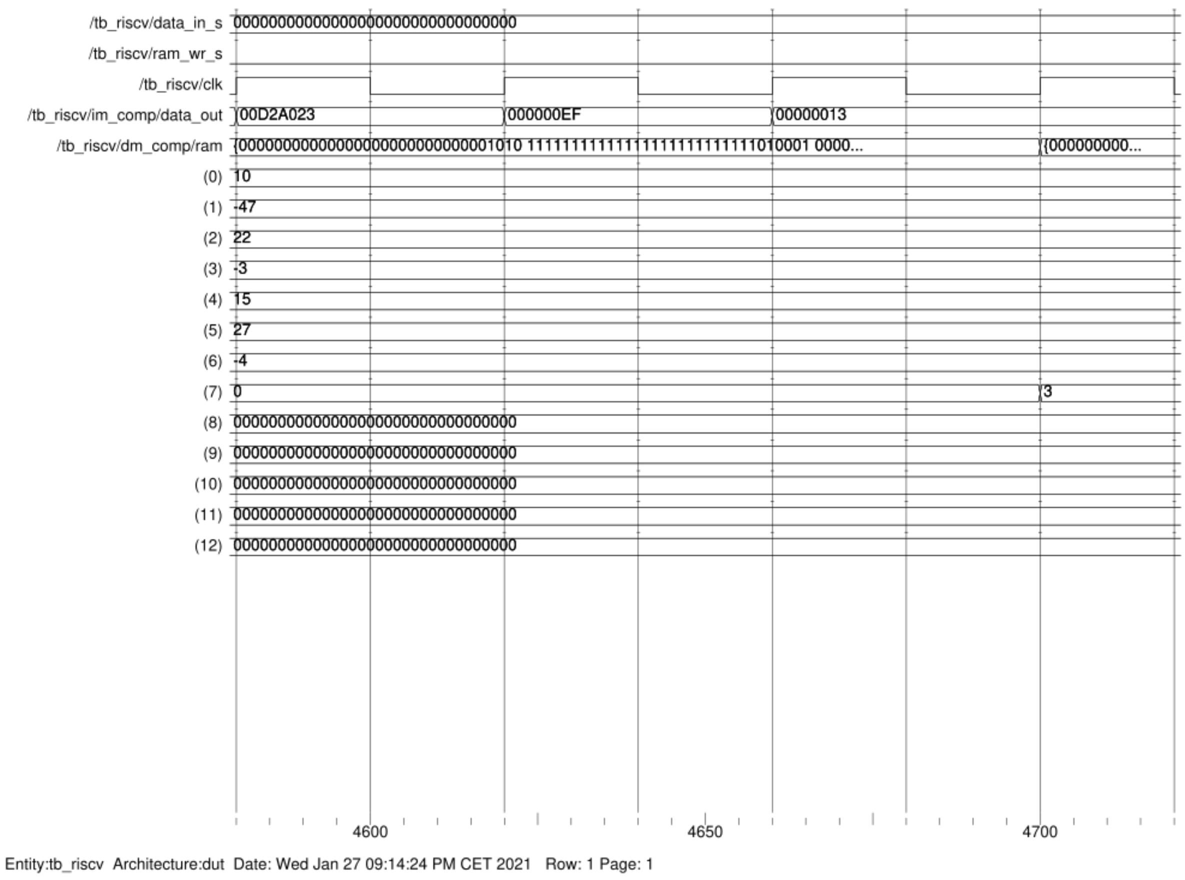

Figure 9: RISCV simulation

4.10 Synthesis

The design has been synthesized by means of Synopsys Design Compiler. First of all the minimum clock period has been found forcing the clock period to 0 ns and assuming, like in the first experience, an uncertainty of 0,07 ns due to the jitter and an input and output delay equal to 0,5 ns. To perform all the correct analysis, a load for each output has been set and for the sake of simplicity a single reference buffer has been used, named BUF_X4 and available in the provided library.
As first step, the Elaborare report has been generated and in this file all the information about the adopted resources are present. Therefore, from this report has been verified that any latch was placed.
Then, by looking to the negative value of the slack, it has been possible to derive the correct clock period which is $T_{ck} = 1,87$ ns that has ensured to have a slack equal to zero. Thus the maximum achievable clock frequency of the design is $f_{ck} = 534,8$ MHz.
From the area report it has been found that the total cell area is $A_{f_M} \simeq 15247,9\ \mu m^2$.
Then, the Verilog netlist has been generated and verified by means of a simulation like in the previous case. After doing that, the same result reported in Figure 9 has been obtained, thus, the correctness of the result has been ensured. Furthermore the correct behaviour of the design has been verified comparing all the instructions with the RARS simulator.
Moreover, to obtain the switching activity of the system the "backannotation" process has been exploited. First, the VCD (Value Changed Dump) file has been generated using ModelSim, this file contains all the information about the switching activities of netlist's nodes, annotated by the simulator during a more realistic simulation of the circuit. Then, through a given script, the VCD file has been turned into a SAIF file (Standard Activity Interchange Format), that has led to a more precise power estimation.
Finally, the netlist and the SAIF file have been analyzed by Synopsys Design Compiler, which has produced a power report (Report 1) containing all the information concerning the power consumption of the architecture under design.

Report 1: Power report RISCV "backannotated"

```
                Internal         Switching          Leakage            Total
Power Group     Power            Power              Power              Power    (   %    )  Attrs
---------------------------------------------------------------------------------------------------
io_pad            0.0000           0.0000             0.0000             0.0000  (   0.00%)
memory            0.0000           0.0000             0.0000             0.0000  (   0.00%)
black_box         0.0000           0.0000             0.0000             0.0000  (   0.00%)
clock_network     0.0000           0.0000             0.0000             0.0000  (   0.00%)
register        274.1708           2.9810         1.3220e+05           409.3548  (  63.05%)
sequential        0.0000           0.0000             0.0000             0.0000  (   0.00%)
combinational    22.1212          38.3017         1.7945e+05           239.8713  (  36.95%)
---------------------------------------------------------------------------------------------------
Total           296.2921 uW       41.2828 uW      3.1165e+05 nW        649.2261 uW
```

As it is possible to observe from the Report 1, the Leakage power is very high and it is comparable to the Dynamic Power. This may be due to the very high complexity of the design and the presence of very complex components like the ALU. The power contribution to the registers is very high, the higher parallelism (32 bit) and the higher number of pipeline stages (5) has led to a very high number of flip-flops with a consequent large amount of dynamic power.

4.11 Place & Route

After the synthesis of the RISC-V, place and route operations have been performed with Cadence Innovus, the clock frequency has been set at f$_{clk}$=534,8 MHz, which is the maximum clock frequency when the slack is equal to 0 ns.
This choice has led to negative slacks inside the timing report files (.slk) , this means that timing constraints have been violated. Therefore, it has been necessary to relax the timing constraints inside the .sdc file, thus a new clock equal to $T_{ck'} = 1,97$ ns has been set. This new period has led to a lower clock frequency equal to $f_{ck'} = 507,6$MHz. In this phase all the steps did in Laboratory 1 has been performed. In Figure 10, the layout that has been created is reported.

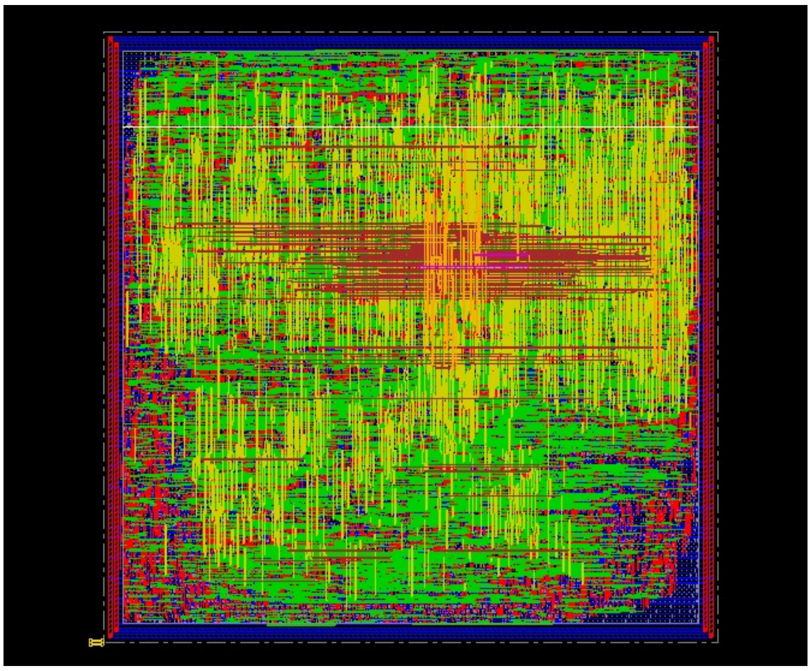

Figure 10: RISCV layout

It has been possible to extract the value of the area, the gates and the cells of the RISC-V design, that are:

- Area= 14273.8 μm^2;
- Gates= 17887;
- Cells= 6499.

Then the last step of the place and route operation has consisted of saving the netlist as a Verilog file.

4.12 Post place and route simulation and switching-activity-based power consumption estimation

The Verilog netlist generated in the previous step has been simulated in order to verify the correct behaviour of the design and to generate the *vcd* file which stores the switching activity information.
The design behaved as expected and the minimum (in terms of absolute value) element of the initial vector has been written in the correct position.
By means of Innovus a new Timing Analysis has been performed and all the parasitics have been extracted. After that step, a new power estimation with an input activity equal to 0,2 and a dominand frequency equal to f=100 MHz has been performed using the result previously annotated in the *vcd* file. All these results are reported in Report 2.

Report 2: Power report RISCV post P&R

Group	Internal Power	Switching Power	Leakage Power	Total Power	Percentage (%)
Sequential	0.2833	0.004872	0.1322	0.4204	59.56
Macro	0	0	0	0	0
IO	0	0	0	0	0
Combinational	0.05119	0.07236	0.162	0.2855	40.44
Clock (Combinational)	0	0	0	0	0
Clock (Sequential)	0	0	0	0	0
Total	0.3345 mW	0.07723 mW	0.2942 mW	0.7059 mW	100

Comparing these results with the ones from the power estimation of the Synthesis present in Report 1, it has resulted that all the values are very similar. The dynamic power is slightly larger than the previous one, this may result from the parasitic extraction, so all the additional capacity associated to the connections are taken into account during the computation of the power.

5 Enhanced Version

5.1 ABS instruction

In this section, the task was to apply some changes to the architecture in order to compute directly the absolute value. First of all the Assembly code has been modified in order to support the new instruction, thus all the instructions associated to the computation of the absolute value have been replaced with the new one called "abs". By means of the new instruction the code resulted shorter than the original version, this is a very good result since the occupied space in the Instruction Memory is lower. The drawback is that in order to support such special instruction it has been necessary to modify the structure of the RISC-V and to design the new encoding of the new instruction from scratch.
In the following piece of code is performed the absolute value by means of several operations, after the load of a new element in the location x8, the absolute value is performed with 4 different instructions.

```
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
```

The new instruction **abs** leads to a shorted code.

```
beq x16,x0,done   # check all elements have been tested
lw x8,0(x4)       # load new element in x8
abs  x10,x8,x0   # put in x10 the absolute value of x8
addi x4,x4,0x4   # point to next element
```

The definition of the ABS operation is reported in Table 6.

Table 6: ABS instruction

Format	funct7	rs2 [24-20]	rs1 [19-15]	funct3	rd [11-7]	Opcode
bin	0000000	01000	00000	001	01010	0110011
hex	0x00801533					

The instruction is an R-type, since the absolute value is applied to only one input, one of the two source register has a dummy address equal to x0, the second source register is referred to the register x8 since in this position is loaded the new element to process. The destination register is x10.
The instruction is executed inside the ALU, the processed data are in 2's complement, this has been exploited to compute the absolute value. First of all a sign check is performed, if the MSB is 0 nothing has to be done, while if it is 1 the processed operand is first negated and after a 1 in the least significant position is added, this solution is less demanding in terms of resources rather than multiplying the number by -1. In terms of hardware resources this solution avoid the allocation of a multiplier, which is in general more expensive rather than few NOT gates and a plain adder.
In Table 7, all the controls for the ALU and the type of operations are reported, for the **abs** operation the control signal "1010" has been chosen.
Since some instructions have been removed from the original code, the immediate values generated by the RARS tool, used as offset to compute the target addresses, were no more valid, thus has been necessary to recompute them. Since the abs instruction is a custom one, it has been replaced with a dummy instruction (NOP), like a placeholder for the abs, to give the possibility to the RARS to compile the code and then, once the binary content of the instruction memory has been generated, the NOP instruction has been replaced with the abs.

In Table 7 and in Table 8 are reported respectively the ALU's control signal with the correspondent Operation and the coding of the ALU_control signal with the new abs instruction.

Table 7: ALU organization with ABS instruction

Instruction	ALU_control (from ALU_ctrl)	Op. type
LW	0000	rs1+offset
ADDI	0001	rs1+imm
SRAI	0010	shift right
ANDI	0011	AND
AUIPC	0100	offset+PC
SW	0101	rs1+offset
ADD	0110	rs1+rs2
XOR	0111	XOR
SLT	1000	rs1<rs2
BEQ	1001	rs1=rs2
ABS	1010	|rs1|
LUI	1011	forward
JAL	When others	z=1

Table 8: ALUControl ABS instruction

Instruction	ALU_op (from CU)	funct3	Bit30	ALU_control
LW	0000	010		0000
ADDI	0001	000		0001
SRAI	0001	101	✓	0010
ANDI	0001	111		0011
AUIPC	0010	/		0100
SW	0011	010		0101
ADD	0100	000		0110
XOR	0100	100		0111
SLT	0100	010		1000
BEQ	0101	000		1001
ABS	0100	001		1010
LUI	1000	/		1011
JAL	0111	/		When others

5.2 Simulation

The system has been simulated adopting a clock period equal to T$_{ck}$=40 ns. In Figure 11, the simulation of the new design is reported. Also in this case is possible to observe the store instruction ("00D2A023") and after three clock cycles the minimum value (+3) is written inside the data memory. It is also possible to observe the abs instruction ("0x0801533") and the stall condition in which the latter instruction is stalled.

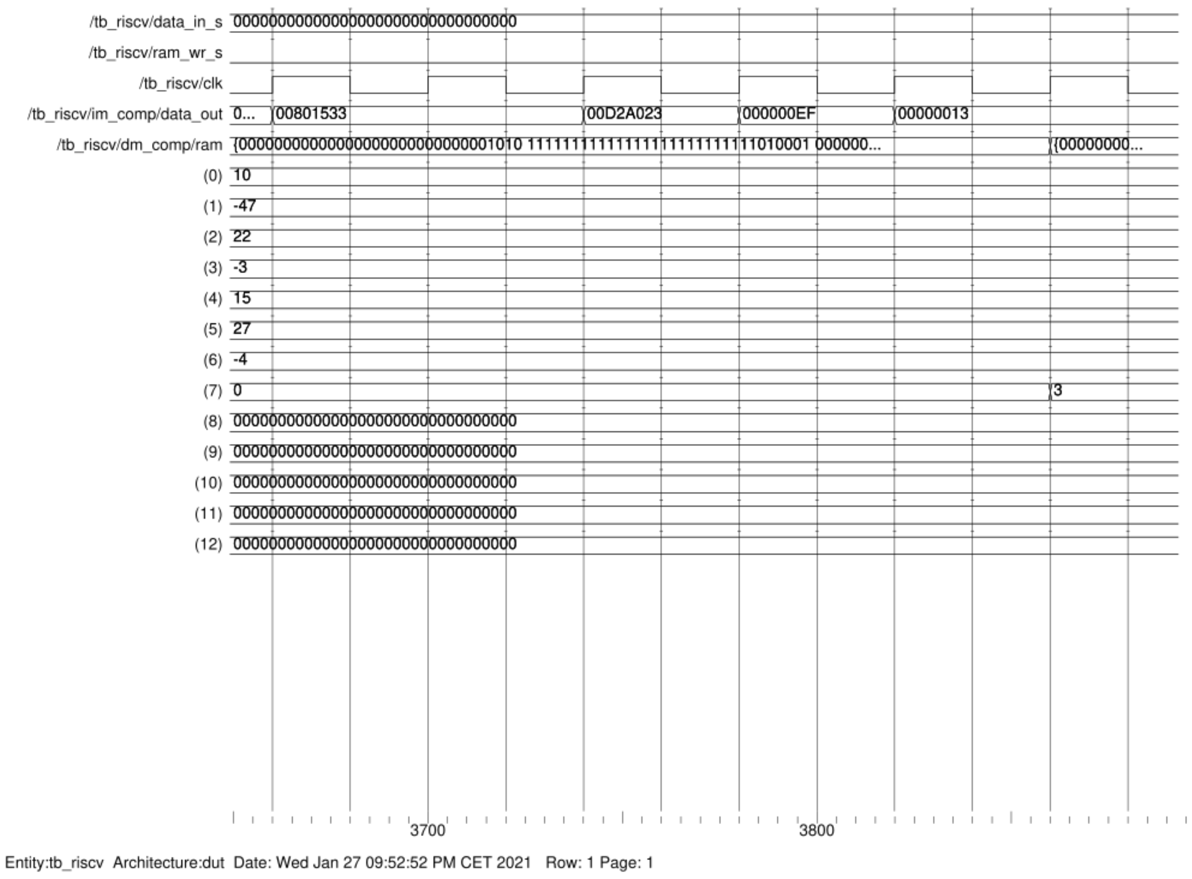

Figure 11: RISCV simulation

5.3 Synthesis

Even in this enhanced case, the design has been synthesized by means of Synopsys Design Compiler. Thus, as first step the Elaborate has been checked and any latch was present. Then the clock period has been set to 0 ns, which gives a negative slack. Therefore, starting from the uncorrect result of the slack, the minimum clock period has been found in order to obtain a slack equal to zero, that corresponds to the case in which the data required time and the data arrival time are equal. The minimum clock period that has been obtained is T$_{ck}$= 1,99 ns to which corresponds the maximum frequency, that the design could reach without errors, equal to f$_{ck}$= 502,5 MHz. This frequency is lower than the base version ($f_{ck} = 534,8$ MHz), this may be due to the fact that the path crossed by the new abs instruction is longer. In fact there is a comparator that checks sign of the operand and then, if it is the case, the number is flipped and then a 1 in the LSB is added, thus the path may be longer.
Regarding the area of the new version of the processor, this has been resulted equal to $A_{f_M} \simeq 15303\ \mu m^2$, thus this new implementation occupies an higher area with respect to the previous one and it is correct because in the ALU block of the enhanced version some components have been added in order to perform the abs instruction.
Then, the Verilog netlist has been generated and simulated as in the previous cases and the behaviour of the processor has been resulted to be the same as all the previous simulations, thus confirming the correctness of the obtained result.
As before, the switching activity of the system has been obtained by means of the "backannotation" process, in which the VCD file has been generated using ModelSim in order to compute the switching activities of netlist's nodes and then this file is turned into a SAIF file, to have a more precise power estimation. Thus, as final step, the netlist and the SAIF file have been analyzed by Synopsys in order to produce a power report, that is present in Report 3, in which all the information concerning the power consumption of the enhanced architecture are reported.

Report 3: Power report RISCV "backannotated" enhanced version

```
                 Internal         Switching          Leakage            Total
Power Group      Power            Power              Power              Power    (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000             0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000             0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000             0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000             0.0000             0.0000  (   0.00%)
register         276.7549           4.5926         1.3453e+05           415.8753  (  64.09%)
sequential         0.0000           0.0000             0.0000             0.0000  (   0.00%)
combinational     25.0950          42.9825         1.6496e+05           233.0372  (  35.91%)
--------------------------------------------------------------------------------------------------
Total            301.8499 uW       47.5751 uW      2.9949e+05 nW       648.9125 uW
```

Comparing the power results of the enhanced version with the base one, it is possible to observe that the internal power and the switching power are increased. In particular, the combinational internal power is increased since new components (the comparator, the adder and additional NOT gates) have been inserted in the architecture.
Moreover, the fact that the switching power is increased may be due to the possible flipping of the number if the comparator states that this is the case, since this could lead to a higher switching activity. But at the same time the leakage power is decreased with respect to the base version and this could be due to the types of cells that Synopsys has chosen during the synthesys.This lower leakage power compensates the higher internal and switching powers and it leads to a final total power that results a little bit lower than the previous case without the abs instruction. So even if the dynamic power is increased due to the additional hardware requirements in the architecture and thus additional load in the nodes, the smaller number of instructions that are executed to compute the absolute value have probably led to a small amount of static power since there are few clock cycles in which the other hardware components are not used and contributing to the leakage power.

5.4 Place & Route

As in the previous case, a place and route of the new version of the processor has been done with Cadence Innovus.
At the beginning the value of frequency has been set to $f_{ck} = 502,5$ MHz that corresponds to a period of 1,99 ns. But, even in this case, this choice has led to a negative slack in the setup and hold timing analyses and thus to a violation of the timing constraints.
Therefore, these problems have been fixed in these ways:

- to avoid the negative slacks in the setup analysis, the timing constraints inside the .sdc file has been relaxed, thus a new clock equal to t$_{ck'}$= 2,8 ns has been set, that corresponds to a clock frequency of 357,1 MHz;
- After sizing the clock cycle to achieve the timing constraints, it has been identified a negative slack for the hold time. First of all the paths with negative slack have been identified using the tool Timing Debug provided by Innovus, this has allowed to analyze in detail these paths, showing for example all the placed cell with their driving strength. To fix the problem it has been necessary to slow down the involved lines and to do that the cells at the end of the paths have been replaced with others with a higher drive strength (for example from X1 to X2). This operation has been carried out adopting the command **ecoChangeCell -inst instance_name -cell cell_name**, specifying first the instance to replace and then the type of cell. This operation has led to the insertion of cells with a higher drive strength, that are obviously faster, but they have a higher input capacitance that loads the lines that are placed before these cells. This additional load has allowed to add a negligible delay (in terms of overall delay) but high enough to fix the hold violations.
  After replacing the cells, has been necessary to remove and insert a new time the fillers and then a new routing phase has been performed.

After obtaining all the slack values as positives, even in this case, all the steps did in Laboratory 1 has been performed.
The layout of the enhanced version of the processor that has been created after the placement step is reported in the figure below (Figure 12).

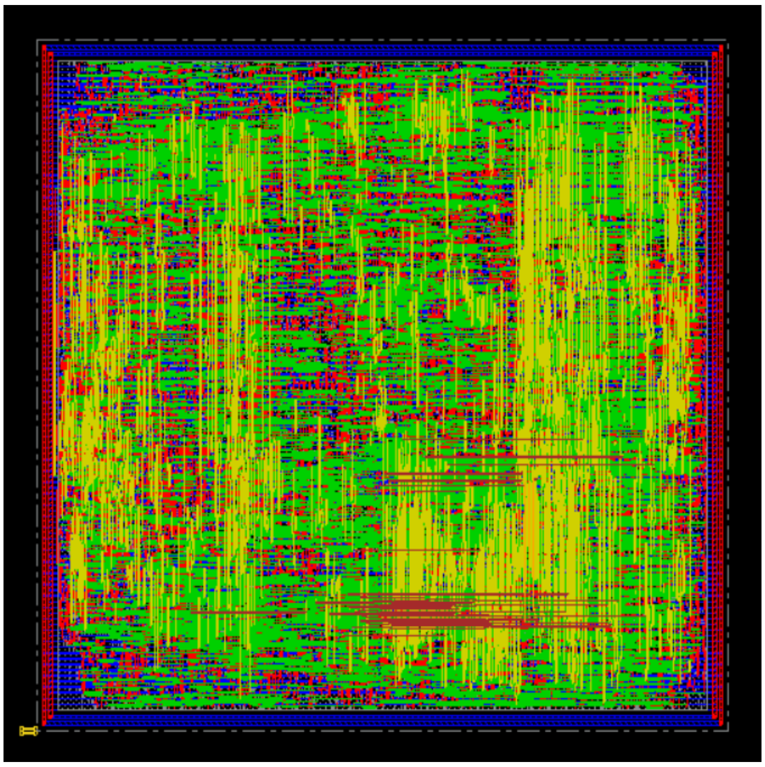

Figure 12: RISCV enhanced layout

Thus, it has been possible to extract the values of area, gates and cells of this new RISC-V design, that correspond to:

- Area= 14817.3 μm^2;
- Gates= 18568;
- Cells= 6552.

These values are a bit higher than the values of the base version and this is due to the fact that in the enhanced version new components have been added into the ALU block in order to implement the abs instruction.
Then the last step of the place and route operation has consisted of saving the netlist as a Verilog file.

5.5 Post place and route simulation and switching-activity-based power consumption estimation

In this step of analysis, the Verilog netlist generated previously has been simulated with ModelSim in order to verify the correct behaviour of the design and to create the *vcd* file, where the switching activity information are stored.
Even in this case the behaviour of the design has been found to be correct. Thus. by means of Innovus a new Timing Analysis has been performed, extracting all the parasitics. Therefore, a new power estimation has been performed with the same values of input activity (0,2) and dominand frequency (f=100 MHz) of the base version. To do that the results previously annotated in the *vcd* file have been used. This results could be observed in Report 4.

Report 4: Power report RISCV post P&R enhanced version

```
Group                       Internal   Switching    Leakage      Total   Percentage
                            Power      Power        Power        Power   (%)
-----------------------------------------------------------------------------------
Sequential                    0.2822     0.00524     0.1356      0.423       62.86
Macro                              0           0          0          0           0
IO                                 0           0          0          0           0
Combinational                0.03566     0.06085     0.1534     0.2499       37.14
Clock (Combinational)              0           0          0          0           0
Clock (Sequential)                 0           0          0          0           0
-----------------------------------------------------------------------------------
Total                   0.3179 mW  0.06609 mW     0.289 mW   0.6729 mW         100
```

These power values are a bit higher than the values estimate by the Synopsys (Report 3) and this is right because Synopsys could not manage to take into account all the real power contributions since the library that it considers is an academic library.