Politecnico di Torino

Integrated System Architecture
RISC-V lite-processor

Third Lab Report

`https://github.com/aledolme19/ISA_LAB.git`

Francesco Babbaro   Pierfrancesco Boccardi   Alessandra Dolmeta

2021-2022

1 Introduction

The third laboratory experience aims to design a RISC-V lite processor, in particular the RV32I, with a smaller Instruction Set Architecture (ISA) with respect to the one listed in the official documentation "*The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20190608-Base- Ratified, Andrew Waterman1, Krste Asanovic, June 8, 2019*". Moreover the whole experience can be divided into two main parts:

1. **Basic architecture**:
   - Design the processor, at the register transfer level, in order to support the given ISA.
   - Functional Verification of the designed processor through simulation using Modelsim Altera.
   - Logic synthesis of the final design and netlist verification with the Design Compiler by Synopsys.
   - Place and route with Virtuoso tool by Cadence.
2. **Advanced architecture:**
   - Modification of the desgined processor to support a function which computes the absolute value of a number.
   - Functional verification through simulations using Modelsim Altera.
   - Logic synthesis and netlist verification.
   - Place and route.

The implemented design must be able to run the binary code corresponding to the required Instruction Set Architecture. In order to test and compare the functionality of design, an assembly code running on the open source software RARS is provided. The test program reads the integer data from the Data Memory and calculate the minimum number using the required ISA. For the second part of the lab exercise, the same assembly code is modified to support the absolute function.

2 Instruction set

In the basic RV32I ISA, there are four core instruction formats (**R**/**I**/**S**/**U**/**B**/**J**), as shown in Figure 1. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory.

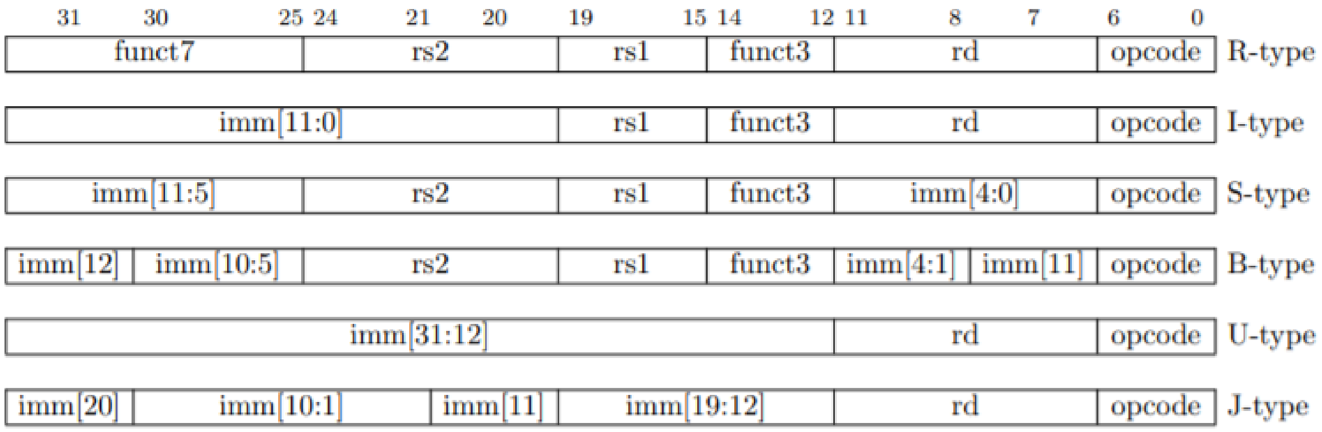
31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 rs2 rs1 funct3 rd opcode R-type
imm[11:0] rs1 funct3 rd opcode I-type
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode S-type
imm[12] imm[10:5] rs2 rs1 funct3 imm[4:1] imm[11] opcode B-type
imm[31:12] rd opcode U-type
imm[20] imm[10:1] imm[11] imm[19:12] rd opcode J-type

Figure 1: RISC-V base instruction formats, showing immediate variants

The official instruction set to be supported according to design constraints is the following:

- `ADD` (R-type): ADD instruction performs the addition of the two source operand register contents and store the result into the destination register.

$$rd \leftarrow rs1 + rs2$$

- `ADDI` (I-type): ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

$$rd \leftarrow rs1 + imm$$

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

$$rd \leftarrow PC + imm$$

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

$$rd \leftarrow imm$$

- `BEQ` (B-type): Branch-If-Equal instruction compares the two source operand register contents. If equal, a branch is taken. The target address is calculated with the help of sign extended immediate operand.

$$PC \begin{cases} PC + 2 \cdot imm \; if \; rs1 = rs2 \\ PC + 4 \; if \; rs1 \neq rs2 \end{cases} \tag{1}$$

- `LW` (I-type): Load Word instruction performs the Load of a value from data memory into a destination register. The address of the data memory is obtained from the addition of the sign extended immediate field to the first source register content.

$$rd \leftarrow DMEM[rs1 + imm]$$

- `SRAI` (I-type): Right shifting of the source register content by an amount reported in the immediate field, the result is stored into the destination register.

$$rd \leftarrow rs1 >> imm$$

- `ANDI` (I-type): ANDI is the logical operation that perform bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

$$rd \leftarrow rs1 \; and \; imm$$

- `XOR` (R-type): Logical XOR operation between the two source register contents, the result is stored into the destination register;

$$rd \leftarrow rs1 \; xor \; rs2$$

- `SLT` (R-type): Comparison between the two source register content, if the first one is lower than the second, then decimal value ‘1’ is stored into the destination register, ‘0’ otherwise.

$$rd \leftarrow \begin{cases} '1' & if \; rs1 < rs2 \\ '0' & if \; rs1 \geq rs2 \end{cases} \tag{2}$$

- `JAL` (J-type): The PC is updated with a jump address obtained with the addition of the PC value with the sign extended immediate field. The return address is the PC current value +4 and it is stored into the destination register.

$$PC \leftarrow PC + 2 \cdot imm$$

$$rd \leftarrow PC + 4$$

- `SW` (S-type): Store the value of the second source register into the data memory. The memory address is obtained from the addition of the sign extended immediate field with the first source register.

$$DMEM[rs1 + imm] \leftarrow rs2$$

The RISC-V Instruction Set does not include explicitly any `NOP` operation. So, it is implemented as `ADDI x0, x0, 0`. Moreover, considering the reference RV32I Base Instruction Set, the different OPCODEs for our simplified ISA are shown in Table 1.

Instruction	OPCODE	Instruction	OPCODE
`LUI`	0110111	`SW`	0100011
`AUIPC`	0010111	`ADDI`	0010011
`JAL`	1101111	`ANDI`	0010011
`BEQ`	1100011	`SRAI`	0010011
`LW`	0000011	`ADD`	0110011
`SLT`	0110011	`XOR`	0110011

Table 1: Instruction OPCODEs

3 Architecture

Figure 2 shows the unprivileged state for the base integer ISA. For RV32I, the **32 x-registers are each 32 bits wide**. `Register x0` is hardwired with all bits equal to 0. General purpose registers x1–x31 hold values that various instructions interpret as a collection of Boolean values, or as two's complement signed binary integers or unsigned binary integers. There is one additional unprivileged register: the `program counter pc` holds the address of the current instruction

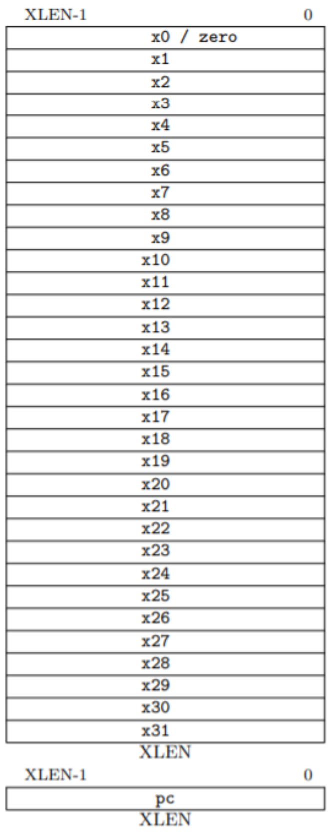

Figure 2: RISC-V base unprivileged integer register state

In general, each instruction belonging to ISA could be executed and ends in a single clock cycle. However, some instructions can requires memory access, that would mean a very long clock period. So to improve clock frequency and increase throughput, pipelining was adopted. The entire architecture can be divided into five different stages:

- Instruction Fetch (IF) Unit;
- Instruction Decode (ID) Unit;
- Execution (EX) Unit;
- Memory (MEM) Unit;
- Write-Back (WB) Unit;

Although, memories are not considered as part of the project.

3.1 Fetch Unit

The first block of the processor is the **fetch unit**, whose purpose is to **compute the address for the instruction memory and to take the correct instruction for the processor**.

PC

This task is performed by the `program counter` (**PC**).
The current value of the program counter is updated and can have one of the three possible new value:

- **Next Address**: evaluated using `Adder1`;
- **Jump Address**: it is computed in the decode stage by adding the PC with Immediate field of unconditional jump `JAL` instruction
- **Branch Address**: it is the target address of branch which is also computed in the decode stage, in the same way of Jump Address, but the branching is validated in the memory stage.

To decide which new address must be used a `multiplexer` driven by one 2-bit signal is allocated. The truth table is shown in Table 2.

Address	`Mux_PC_sel`
Next Address	00
Jump Address	01
Branch Address	10

Table 2: PC multiplexer truth table

The selection signal `Mux_PC_sel` of the multiplexer is given by the concatenation of the PCSrc and the Jump signals. As it will be discussed in the following sections, in order to simplify the management of the branch and the jump instruction, the common execution will be freezed for some clock cycles. By doing so, there is no the possibility to have the selection signal `Mux_PC_sel` of the multiplexer equal to `11`.
The Branch Address and the Jump Address are both computed by the `Adder2`. However, they cannot be feedback in the same way. In fact, in the case of the Jump instruction, the new PC value is already available in the decoding stage since there is not any condition to evaluate, the jump has to be performed in any case. This means that we are able to feedback it in the `multiplexer` immediately in the decode stage. In the case of the Branch instruction, there is a condition that must be satisfied in order to load the new PC value into the program counter: this condition is evaluated in the execution stage, and forward back toward the `multiplexer` in the memory stage (*according to the specific of the complete RISC-V*).

`Adder1`

It is simply evaluating the next address value, by making the sum of the current value of the PC register and 4.
In fact, instruction memory has a parallelism of 1 Byte. Therefore, **each instruction is distributed among four adjacent locations**. All the instructions are aligned in memory in order to be stored starting from an address having the least two significant bits to '0'. Hence, to pick the next instruction from the memory the program counter must be incremented by four, exploiting the adder shown in the fetch stage.

Instruction Memory

After the program counter is fed properly, in the rest of the clock cycle the new instruction is fetched from the memory. Since the program counter has 32 bits, it is possible to have an instruction memory with 2^{32} locations. For this specific case, since the design of the memory was not part of the project, the `Instruction_Memory` has been put in the test-bench.

We decided to define a simple ROM as instruction memory, with a size lower than 2^{32}, since the number of instructions of the code was really low. Therefore, we used a ROM with 2^7=128 locations with 32 bits each, taking as address only 7 out of 32 bits coming from the RISC-V.

The outputs of the fetch unit are:

- Program Counter current value: this will be used in the decode unit in the case a `JUMP` or a `BRANCH` instruction is required;
- Program Counter next value: this will be forwarded until the write back stage. In particular, this is used in the case a `JAL` instruction is needed. In fact, in this case, the PC must be updated with the jump address, but the PC value +4 must be also store in the destination register;
- 32 bits instruction: this will be used in order to decode the instruction that must be performed.

Each of those will be used for different purposes in the following blocks.
The inputs of the fetch unit are:

- Branch Address and Jump Address;
- `Mux_PC_sel`;
- `PCWrite` and `IF/IDWrite`;

The last two signals will be better described in the following section, when analyzing the `Hazard Unit`.

3.2 Decoding Stage

The **decode unit** has to **receive the instruction fetched in the previous stage and has to identify which operations must be executed**. Therefore this stage must generate all control signals and operands for other stages.
Given the list of supported instructions in the ISA section, by looking from the free RISC-V green card, it can be understood how to identify an instruction, as shown in Figure 1.
The instructions previously presented can be divided into different formats, where each format has its own organization and its way to carry the information. The supported instructions previously discussed are divided as shown in Table 3.

R	I	S	B	U	J
ADD	LW	SW	BEQ	LUI	JAL
XOR	SRAI			AUIPC	
SLT	ADDI				
	ANDI				

Table 3: Instruction format division

Register File

The `register file` is the unit of the processor where data to be used as operands are stored, since the processor can perform operations **only** on these data. As consequence, if a value is stored into the data memory, it has to be firstly moved in the register file in order to be used as operand. RF contains 32 registers (FFs in our implementation) that can be read (0-31) and 31 of those can also be written (1-31). As already mentioned, the first register `x0` is a read-only register containing only zeroes and it can not be written, even if the `RegWrite` signal is asserted.
All these registers are able to sample input data when clock rising edge occurs, storing it in the memory location pointed by the writing address (`rd`). Being FFs, the reading operation is performed in a combinational way: the output of the FFs

is connected to MUXes that selects the correct output of the registers according to the given two reading addresses (`rs1` and `rs2`). This is a 3-port memory component, therefore we can read two registers and write one register at the same time. Since there are four pipeline stages, the write signal `RegWrite` is not corresponding to the one of the current instruction but to one that has been fetched 5 cycles before.
The input of the `register file` is given by the output of the `Mux_Instr`, whose purpose is described in the following subsections.

Control Unit

The mind of the processor is the `control_unit`, which can identify the signals to send to the execution unit according to the `opcode`. This operation is totally combinational, therefore the control unit can be seen as a simple **look up table**. This is shown in Table 4.

Instr	OPCODE	Branch	MemRead	ALUOp	MemWrite	ALUSrc	RegWrite	Jump	MemtoReg	Lui	Auipc
`add`	0110011	0	0	10	0	0	1	0	10	0	0
`addi`	0010011	0	0	00	0	1	1	0	10	0	0
`andi`	0010011	0	0	00	0	1	1	0	10	0	0
`xor`	0110011	0	0	10	0	0	1	0	10	0	0
`srai`	0010011	0	0	00	0	1	1	0	10	0	0
`beq`	1100011	1	0	01	0	0	0	0	XX	0	0
`slt`	0110011	0	0	10	0	0	1	0	10	0	0
`lw`	0000011	0	1	00	0	1	1	0	01	0	0
`sw`	0100011	0	0	00	1	1	0	0	XX	0	0
`auipc`	0010111	0	0	11	0	1	1	0	10	0	1
`lui`	0110111	0	0	11	0	1	1	0	10	1	0
`jal`	1101111	0	0	XX	0	X	1	1	11	0	0

Table 4: Control Unit LUT

A short description of each control signal can be shown as:

- `Branch` is the signal which is raised when `BEQ` instruction has been fetched.
- `MemRead` is the control signal to read the content from data memory.
- `ALUOp`: 2-bit wide, is a vector that must be propagated to the ALU control block in the execution unit. In particular:
  - `10`: R-type instructions;
  - `01`: B-type instructions;
  - `11`: J-type instructions and U-type instructions. For the `LUI` instruction, we will perform an ALU operation, given by the sum between "0" and the immediate field;
  - `00`: I-type instructions;
  - `XX`: for `JAL` they are set to DON'T CARE (X) because no operation has to be performed by the ALU.

  This control signals has been chosen according to the RISC-V policy. The different instructions are distinguished one from the other more in terms of type of instructions rather than in terms of operation to be performed. This is done instead by the successive unit, which is the `ALU control`.
- `MemWrite` is used to enable the memory to be written (store instruction).
- `ALUSrc` drives one of the multiplexer at the input of the ALU that is in charge of choosing the second operand of the ALU. In particular:

  - `0`: the second operand of the ALU will be the value stored in `rs2`;
  - `1`: the second operand of the ALU will be the immediate field.

- `RegWrite` is the control signal to write inside the register file and must be propagated with the data of the corresponding instruction. It is used during the write-back stage.
- `Jump` is set to 1 when the fetched instruction is the `JAL`.
- `MemtoReg`: is a 2 bit wide vector containing the signals to drive the multiplexer in the write back unit.
- `Lui`: it is asserted only when a `LUI` operation is fetched and it drives a MUX in front of the ALU in order to select `0` as first operand of the ALU. In this way the operation `LUI: rd=0+imm` is performed.
- `Auipc`: it is asserted only when a `AUIPC` operation is fetched and it drives the same MUX as `Lui` signal in order to select as first operand of the ALU the value currently stored in the PC. In this way the operation `AUIPC: rd=PC+imm` is performed.

Immediate Generate

Another functional unit that is instruction-dependent is the `Immediate Generate`.
This block takes the whole instruction and it assembles the immediate in the correct way according to the `OPCODE` and `FUNCT3`. In particular, the immediate field is composed according to Table 5, where the numbers between square brackets are referred to the indexes of the bits of the **instruction** to be taken to correctly form the immediate.

Instructions	**Immediate pattern**	**32-bit extension**
`ADDI - ANDI - LW`	[31 : 20]	Sign extension
`SRAI`	[24 : 20]	0 padding (unsigned)
`BEQ`	[31]—[7]—[30 : 25]—[11 : 8]—'0'	Sign extension
`SW`	[31 : 25]—[11 : 7]	Sign extension
`AUIPC - LUI`	[31 : 12]—"000000000000"	No need
`JAL`	[31]—[19 : 12]—[20]—[30:21]—'0'	Sign extension

Table 5: Immediate Generate Unit

Adder2

`Adder2` is an adder which is used in order to compute the Jump Address and the Branch address of which we have discussed in the previous section. To implement `JAL` and the `BEQ` instructions, we must compute the target address by adding the sign extended offset field of the instruction to the PC.
As previously stated, `JAL` instruction belongs to the unconditional type. Therefore, there is no jump condition to be evaluated. Once it is fetched one can immediately compute the jump address and forward it within the jump signal to the fetch unit. This is one reason for which an additional `Adder2` is allocated.
Although it is important to notice that the jump address evaluation in this stage improves the performance of the processor, because if it was placed in the execution unit, then another useless instruction would be fetched before the jump address computation.
For what concern the `BEQ` instruction, we will instead feed-forward the address only after having evaluating the condition. Nevertheless, `Adder2` is needed to evaluate the target address, since the ALU must compute the `BEQ` condition (therefore, the difference between `RS1` and `RS2`). Moreover, we have that the offset field is shifted left 1 bit so that it is a **half word offset**; this shift increases the effective range of the offset field by a factor of 2. To deal with the latter complication, we will need to shift the offset field by 1 (this is done directly in the `Immediate Generate`).

Hazard Unit

The **Hazard Unit** handles both *Data Hazards* and *Control Hazards* inserting NOPs in the sequence of instructions.
As far as *Data Hazards*, this unit inserts a NOP after each `load` instruction, when the following instructions requires as source register the destination register employed to store the value read from the memory. It is required because before the *Write Back* stage the destination register is not updated, so until that moment that register should not be read.
However, exploiting the **Forwarding Unit**, it is possible to move up this reading, so only one NOP is necessary.
As a consequence, PC and IF/ID pipe register are not updated in order to guarantee that the instruction currently decoded is executed. This is done exploiting the control signal `HU_OUT_IF_ID_PC_WRITE` in output of the **Hazard Unit**.
Instead, to manage *Control Hazards* NOPs are inserted until the branch condition is not evaluated. At the clock cycle in which a `branch` instruction is decoded PC and IF/ID pipe register are disabled. In this manner, the branch control signal propagates towards the next pipeline stage (EX), while PC and decoded instruction do not change. It is important to avoid that the program execution goes on and new instructions are fetched. In both clock cycles in which the branch control signal is in EX and MEM stage, a NOP has to be inserted. However, in the first case the decoded instruction does not change, instead in the latter case PC and decoded instruction are updated, so that a new instruction can be decoded at the next clock cycle.
In the MEM stage, the outcome of evaluation is obtained (branch taken or not), so at the next clock active edge the new PC is sampled:

- Branch taken: $PC \leftarrow PC + 2 \cdot IMM$
- Branch not taken: $PC \leftarrow PC + 4$

Then, in case of branch taken, at the following clock cycle another NOP operation will have to be inserted, while the PC and IF/ID register have to be updated. In this way, at the next clock cycle the right instruction will be decoded.
In the opposite case, instead, the new instruction can be directly decoded with no need of any NOP. This is because the PC already store the right address.
We decide to insert NOP operations instead of going on with the execution in order to not waste power for useless operations, and avoid the design of a branch prediction unit, which is not necessary in a simple lite RISC-V as the one designed.

`Mux_Instr` and `Mux_Opcode`

`Mux_Instr` and and `Mux_Opcode` have been inserted in the case a NOP operation is required. The selector of the two muxes is the same, and it is the `HU_OUT_NOP_SEL` output signal of the **Hazard Unit**. In the case NOP operation must be perfomed, both the muxes select as input a string of zeros, in order to block the execution.

The output signals of the decode unit are:

- forwarded next PC value from previous stage;
- result of the addition of PC + immediate operand;
- control signals for execution unit;
- control signals for memory unit;
- control signals for write back unit;
- two registers' contents RS1 and RS2;
- two registers' pointers RS1 and RS2;
- the destination register RD once the result of the instruction must be stored;
- immediate operand;

According to the instruction one or more of the output signals can have non valid value since they are not used for that operand.

3.3 Execution Unit

Once all control signals and correct operands have been generated and fed through the pipeline stages, the processor is able to compute first useful data, like the address for the data memory or a new value to be stored.
All logical and arithmetic functions are performed inside the ALU block based on the control signals generated by ALU Control Block.

ALU Control

A special block called `ALU Control` is described to generate the correct value of the `ALUControl` signal, which drives the operation to be performed by the ALU for a given instruction. To decide which values has to be assigned to `ALUcontrol` signal one can look at `Funct3` field of the instruction fetched one cycle before (coming from the ID/EX pipe reg) and at the `ALUOp` signal of the same clock cycle.
The signals generated are shown in Table 6.

Instruction	**ALUop**	**Funct3**	**ALUControl**
`add`	10	000	0010
`addi`	00	000	0010
`andi`	00	111	0000
`xor`	10	100	0011
`srai`	00	101	0100
`beq`	01	000	0110
`slt`	10	010	0110
`lw`	00	010	0010
`sw`	00	010	0010
`auipc`	11	XXX	0010
`lui`	11	XXX	0010
`jal`	XX	XXX	XXXX

Table 6: ALU Control block truth table

For the comparison A=B of the `BEQ`, we decided to use a dedicated comparator that works continuously besides the `ALUControl` signal. In other words, the output `ZERO` of the ALU is always driven by the comparator, even for the instructions in which there is no comparison to be made.
In order to implement a comparator, we could have used a subtractor followed by a 32-bit-NOR, but we preferred to use a totally behavioral definition of the comparator in order to have a lower delay. For this reason, we decided to maintain `ALUControl=0110` specified by the RISC-V official documentation for the `beq` (that is the same of the `sub`), but that value is not actually used in our implementation, since the comparator is always on, besides the `ALUControl` signal (in other word, in our case `ALUControl=XXXX` would have been enough).

For the `JAL` instruction the ALU is not needed since the return address (PC) is written-back entering directly in the MUX of the last pipe stage.

Moreover, the `Funct7` field has not been used in our implementation, because there were no instructions with the same `opcode` differing for `Funct7`. For instance, `add` and `sub` are two of those, but `sub` is not implemented in RISC-V lite.

ALU

The `ALU` is described in a behavioral way in order to make the design easier and to let the synthesis tool to do any optimization needed. Based on the signals generated by `ALU Control`, the ALU can decide what kind of operation can perform depending on the bit sequence of ALU control signals.
Concerning the input pins of the ALU block, there are different possible combinations.

Both for input A and B, first of all, we have two multiplexers, `MuxA` and `MuxB`, whose selectors are the 3-bit signals `ForwardA` and `ForwardB`, coming from the `Forwarding Unit`.
In particular, we have:

- `100`: EX/MEM PC+4 value;
- `011`: RS1 in the case of `MuxA` and RS2 in the case of `MuxB`;
- `010`: EX/MEM ALU result;
- `001`: MEM/WB Read Data;
- `000`: MEM/WB PC+4 value;

Then, at the output of these multiplexers, we have two further multiplexers, different one from the other. In particular:

- `Mux2` for input A. The selector, is given by the concatenation of the two signals `AUIPC` and `LUI`. In particular:
  - `01`: 32-bit of zeros proceeds, in order to perform a symbolic addition with the immediate field. In this way, when a `LUI` instruction is required, we are able to correctly store the immediate without modyfing it as required;
  - `10`: the value of the current ID/EX PC proceeds, in order to build the properly address required by the `AUIPC` instruction;
  - `00`: the output of `MuxA` proceeds to the ALU;
- `Mux3` for input B. The selector is given by the `ALUScr` signal, coming from the Control Unit. In particular:
  - `0`: the output of `MuxB`;
  - `1`: the immediate operand;

The ALU can provide two outputs, one is the result of the operation, the other one is the zero flag which is raised to one when the two input operands are equal. This flag is used to evaluate the branch condition for the BEQ instruction. In other words if the two operands are equal then the branch must be taken. This condition will be evaluated in the memory unit.

Forwarding Unit

The **Forwarding Unit** allows to avoid *Data Hazards*, which may occur when we try to read the content of a register which is the destination register of a previous instruction, and it has not been updated yet.
This unit avoids to wait for the write back operation, having a sort of *bypass*. Therefore no NOPs are needed between sequential instructions, so that wastes of time are avoided.
The **Forwarding Unit** generates selection signals for two MUXes, `MuxA` and `MuxB`, one for each input of the ALU.
In this way, operands can be selected from:

- ID stage: no data dependency between two consecutive instructions
- MEM stage: Data Hazard (Read After Write) between two consecutive instructions
- WB stage: Data Hazard (Read After Write) between two instructions among which there is another instruction with no data dependency

Data Hazards can be discovered simply comparing source registers in EX stage with destination register in MEM or WB stage. However, in case of three consecutive instructions that have a data dependency one from each other (all the instructions read and/or write the same register), we have to take into account that the up-to-date value is the one coming from the second instruction. For this reason the comparison between the source register and the destination register in the MEM stage is the one with higher priority. In fact, if source register is equal to both destination register in MEM stage and WB stage, the input of the ALU in EX stage is taken from the MEM stage.

Therefore the **Forwarding Unit** selects the correct input to feed to the ALU:

- Source register in EX stage corresponds to destination register in MEM stage ⇒ selected value from MEM stage
- Source register in EX stage corresponds to destination register of `jump` instruction in WB stage ⇒ selected return address (PC+4) from WB stage
- Source register in EX stage corresponds to destination register in WB stage ⇒ selected value from WB stage

Otherwise, in absence of data dependency the outputs of Register File forwarded from the previous stage are provided to the ALU.

The output signals of the execution unit are:

- forwarded next PC value from previous stage;
- result of the addition of PC + immediate operand;
- control signals for execution unit;
- control signals for memory unit;
- control signals for write back unit;
- ALU result;
- ALU zero flag;
- RS2 register's content;

3.4 Memory Unit

The memory unit is used only in the case of store/load operations. The only element present is the `Data_Memory`.

`Data_Memory`

The data memory is used to store a value or to save the final value of a program, any intermediate result is stored in the register file. The memory has a parallelism of 32 bits and it is described in VHDL as an array of `STD_LOGIC_VECTOR`.
Pointer to the memory is the output of the `ALU`, that in case of `STORE` and `LOAD` instructions provides an address. As previously described for instruction memory, each word takes 4 memory locations. `MemRead` and `MemWrite` signals are used to perform the corresponding operations on the memory.
Since the design of the memory was out of the project, as for the instructions memory, `Data_Memory` has been put in the testbench, as already explained for the `Instruction_Memory`.

Usually, a data memory is based on the SRAM or flash technologies, but we can not describe them in VHDL. For this reason, we though of different alternatives to have a `Data_Memory` resembling the behavior of a real memory:

- having the same VHDL description of a register file in order to read and write;
- using a ROM in which we can only read and checking the RISC-V signals in the cases in which we need to write.

Both of them have pros and cons.
In the first case you have to initialize the registers with the values stated into the code using a series of `lw` instructions at the beginning of the code, having a complete memory, able to read and write.
In the latter case, you store the values that you want in the ROM before the running of the code, therefore you have no need to add `lw` instructions to the code, but you get a read-only-memory, not able to be written.

Since in our case we write in memory just one time (at the end of the code), we decided to use the second approach, defining a ROM and checking if the RISC-V signals are correct in the case of a write operation.

For the size and the parallelism, we decided to use 5 bits for the address (the 5 LSBs of the 32-bit address coming from the RISC-V) with data of 32 bits.

The output signals of the memory unit are:

- ReadData of the `Data_Memory`;
- forwarded next PC value from previous stage;
- forwarded ALU result;
- control signals for write back unit.

3.5 Write Back Unit

This is the last unit. Here, we have simply a multiplexer, whose selector is `MemtoReg`, coming from the control unit. The output of the multiplexers are:

- `11`: output of the data memory;
- `00`: the PC value + 4, in order to store it in the register files;
- `01`: the ALU's result.

The output of the multiplexer is feedback as `WriteData` input of the `register file`.

3.6 Pipeline

A pipelined version of RISC-V was implemented, in order to increase the clock frequency.
All pipeline registers was designed to sample input data on clock falling edge.
In this way, `register file` and pipeline registers sample data on different phases. This allows to have a transparent behaviour of `register file` when the same register has to be written and read at the same time. For this purpose, obviously writing has to be performed before reading. It is possible having a `register file` that samples on clock rising edge (writing) and pipeline registers that sample on clock falling edge, so that when data is sampled from the `register file` output, this value is already updated.
Let's consider an instruction needs to read an operand from a register at the same clock cycle in which the write back operation of another instruction is performed. Since there is a data dependency, it would lead to a Data Hazard.
However, using a "*transparent*" `register file`, it seems to have a register that acts like a simple buffer.

The final datapath is shown in Figure 3. Each of the register is connected to the `RISC_V_in_rst_n` signal, but to not complicate the draw, it is not shown.

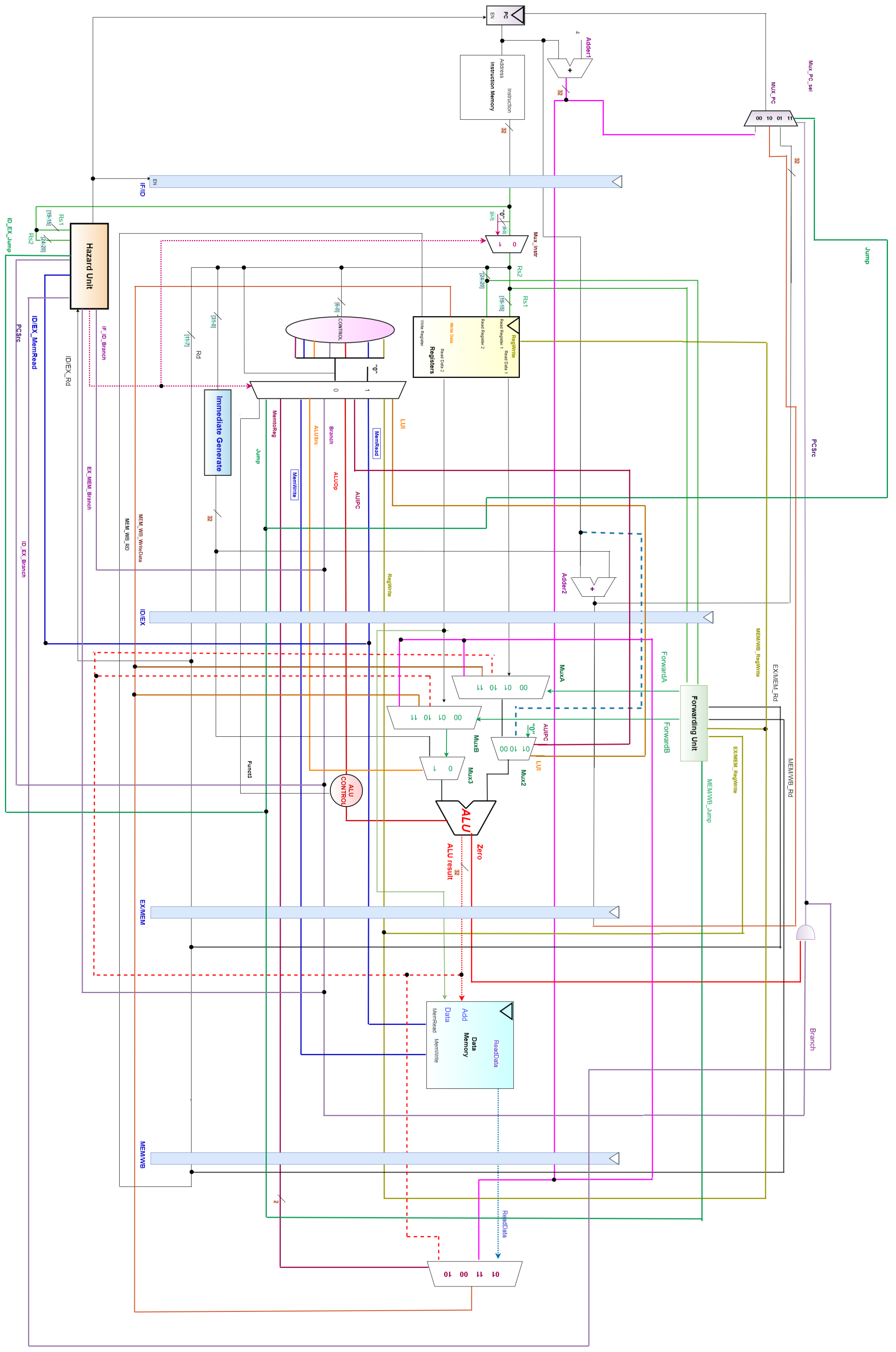

Figure 3: RISC-V

4 Analysis

To verify the correctness of the architecture, an assembly program has been provided. The given code computes the minimum value in a vector of integer numbers: $10, -47, 22 - 3, 15, 27$ and -4.
The code is shown below. The first initialization part has been omitted. Here, `v` and `m` are defined.

```
__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
        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
```

The algorithm is mainly based on loop, where a set of operations are used to compute the **absolute value** of the item. First, there is a check of the content of `x16` register: here, it is stored the number of elements that must be still analysed. The `beq` instructions will return a zero in the case we have analysed all the elements of the vector in input.
In the the condition of the `beq` is not fulfilled, then the content of register `x4` is loaded in `x8`.
First, a shift of 31 position is performed, to obtain the sign of the element (`srai`). Then, a XOR of the sign with the element itself to get the 1's complement if the number is negative and the carry in generation (`andi` between the MSB and 1).
After that, it is performed the addition between the 1's complement and the carry in to get the 2's complement: the absolute value. The result is saved in `x10`.
After absolute value computation, the pointer to the next element is incremented (`andi`) and the item counter is decreased by one (`andi`).
`slt` instruction is used to compare the new absolute value stored in `x10` with the current minimum value in `x13`. If the result is zero, we return to the beginning of the loop (`beq`), otherwise an `add` instruction will load the new minimum value in `x13`. Then, a `jal` will bring us back at the beginning of the loop.

4.1 Simulation

The circuit has been verified on Modelsim.
In Figure 4, is shown the final step of our simulation. Here, all the values of the initial vector has been analysed, the minimum has been found and it is finally written in the `data memory`. As it can be seen from the screenshot, during the MEM stage of the last instruction, `MemWrite` is active, and the value `3` (the minimum absolute value between the one present in the vector) is stored in the memory.
With a clock frequency of 1.45 ns, the final result is written after 190 ns.

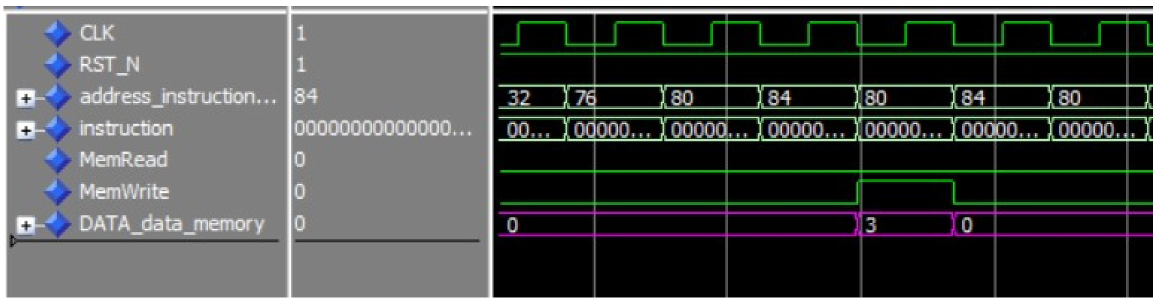

Figure 4: End of simulation

4.2 Synthesis

First of all, a logic synthesis was performed and it was checked that no unwanted latches were present in our design. Then, the design was mapped to the reference library. The synthesis was performed using 0 ns constraint on clock period in order to achieve the best netlist in terms of clock frequency.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} \approx 689\text{MHz}$$

In terms of **area**, the circuit we obtain has a cell area equal to 17433.64 μm^2, which is divided into:

- Combinational area: 8980.16 μm^2
- Buf/Inv area: 1788.32 μm^2
- Non-combinational area: 8453.48 μm^2

Once the clock period was found, also the performances in terms of power can be evaluated:

- Total Dynamic Power: 9.37 mW
- Cell Leakage Power: 320.90 µW

Starting from this synthesized netlist it is possible to extract the Verilog netlist. For this purpose, the whole design was flattened in order to have in only one Verilog file the description of all the designed cells.

4.3 Netlist generation

Starting from this synthesized netlist it is possible to extract the Verilog netlist. For this purpose, the whole design was flattened in order to have in only one Verilog file the description of all the designed cells.
At this point, it is possible to simulate the obtained netlist, in order to verify if it still behaves like the RTL description previously simulated. The same testbench as before was employed in Modelsim simulation, but in this case the clock period was replaced with the minimum one, previously estimated (1.45 ns).
The obtained waveforms are the same shown in Figure 4, so the synthesized netlist still behaves correctly.

4.4 Place & Route

We finally perform the physical layout of the circuit.
Innovus considers not only the capacitance of each block of the design as done in the previous steps, but also parasitic resistances and capacitances due to the interconnections are used to evaluate each power consumption contribution.
The most important steps are:

- **Placement**: the cells that compose the operator are placed;
- **CTS optimization**: try to optimize the design in order to satisfy timing constraints;
- **Route**: it generates the connections among the cells;
- **PostRoute optimization**: it tries to optimize the design.

The first step is to import our synthesized design, thanks to the Verilog netlist extracted in subsection 4.3. Then, we have to structure the **floorplan** and **power rings** all around the core. They will be necessary to distribute VDD and GND to the whole die.
Then **placement** can take place. This process takes some time. It is also carried out more than once to improve the outcome and to minimize the area and wiring costs, reducing consequently the overall power consumption.
The placement operation, in principle, employs an **iterative algorithm** that divides the available area into smaller regions whose boundaries represent constraints which dictate where cells can be placed.
For the standard cell placement, metal layers from 1 to 8 are used. Clock nets are also positioned.
The first optimization is a **post-CTS** optimization (checking for DRVs and setup and hold times, area and power optimization, congestion reduction).
The actual routing takes place, by using the **NanoRoute algorithm**. A **second optimization is performed following routing**: setup and hold times are verified.
After this second improvement, what's left to do is place the **filler cells**, and then the place & route of the design is complete.
Result of Innovus synthesis in shown in Figure 5.

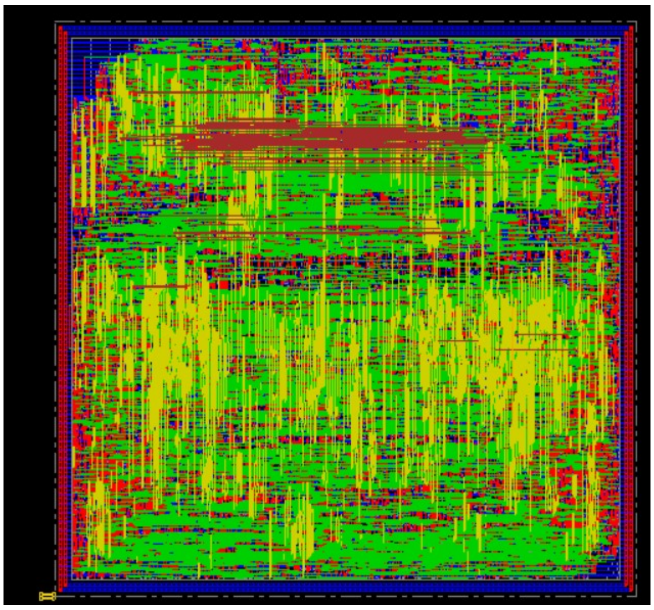

Figure 5: Layout of RISC V

In order to analyze timing and integrity of our circuit, **parasitics** (i.e. resistances and capacitances) **for each metal wire are extracted**.

Inserting a clock period of 3 ns, from timing reports, **no negative slack times have been recorded**.
The final step consists in looking for **geometry and connectivity violations** and none have been spotted.

Innovus consider not only the capacitance of each block of the design as done in the previous step, but also parasitics resistance and capacitance due to the interconnections are used to evaluate each power consumption contribution.
After performing placing and routing operations, we evaluate again power consumption of our lite-RISC V.
We obtain the result shown in Table 7.

Power	Value [mW]	Percentage
Total Internal Power	3.71803097	67.0813%
Total Switching Power	1.53142255	27.6302%
Total Leakage Power	0.29312351	5.2886%
Total Power	5.54257701	

Table 7: Total Power Consumption after Place & Route

The clock used for this synthesis is different with respect to the one used in 4.2, therefore no comparison between can be performed.

5 Advanced Architecture

Starting from the initial architecture, a new instruction must be introduced in the instruction set to perform the **absolute value computation**.
The already implemented ALU must be slightly modified.
Since the RV32 processor does not support the direct computation of the absolute value, a new instruction is created to support that.

The main difference with the previous architecture is the latency cost: with the standard processor four instructions were necessary to compute the absolute value, hence the latency cost corresponds to nine clock cycles. In this advanced architecture the latency cost is given by one instruction completion (5 clock cycles). The new instruction is shown in Table 8.

funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
0000000	00000	01000	111	01010	0110011

Table 8: Abs instruction

The new instruction takes the vector element stored in `x8` from both ports and stores the result of the operation in `x10`. From the opcode the control unit recognizes an R-type instruction, hence the generated ALUOp is 10 and the ALUcontrol block decodes the 1011 sequence as 0101 for the ALU. The ALU block when 1010 is received perform the absolute value.

Being the instruction set of the RISC already defined and since `abs` is not present, a new instruction has been introduced in our implementation of the instruction memory, which is `ABS, X10,X8,X8`, to see if the new HW works correctly, but we are aware of the fact that in a common RV32I it will be not executed.
This instruction substitutes the four lines which compute the absolute value, namely:

1. `SRAI x9,x8,31;`

2. XOR x10,x8,x9;
3. ANDI x9,x9,0x1;
4. ADD x10,x10,x9;

So the assembly code, supposing to have an additional instruction to compute the **absolute value** (abs) becomes:

```
__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
        abs x10, x8, x8     # x10 = abs(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
        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
```

5.1 Simulation

The circuit has been verified on Modelsim.
In Figure 4, is shown the final step of our simulation. Here, all the values of the initial vector has been analysed, the minimum has been found and it is finally written in the data memory. As it can be seen from the screenshot, during the MEM stage of the last instruction, MemWrite is active, and the value 3 (the minimum absolute value between the one present in the vector) is stored in the memory.
The obtained waveforms are the same shown in Figure 4, so the synthesized netlist still behaves correctly. Differently from Figure 4, since the abs is introduced, the number of cycle needed is lower. With a clock frequency of 1.45 ns, the final result is written after 160 ns. The final result is shown in Figure 6.

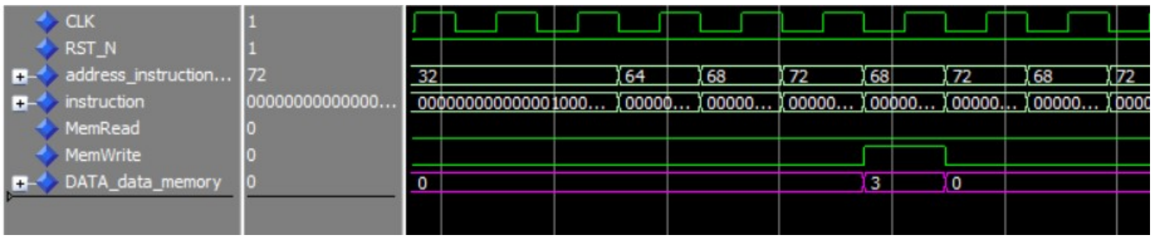

Figure 6: End of simulation

5.2 Synthesis

First of all, a logic synthesis was performed and it was checked that no unwanted latches were present in our design. Then, the design was mapped to the reference library. The synthesis was performed using 0 ns constraint on clock period in order to achieve the best netlist in terms of clock frequency.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} \approx 689\text{MHz}$$

The clock frequency does not change compared to the one of the RISC-V without `abs` instruction. It is related to the fact that the critical path does not depend on the ALU.
In terms of **area**, the circuit we obtain has a cell area equal to 17808.97 µm^2, which is divided into:

- Combinational area: 9355.49 µm^2
- Buf/Inv area: 1922.12 µm^2
- Non-combinational area: 8453.48 µm^2

As we could expect, the combinational area increases, since a new functional unit was join to the ALU.
Once the clock period was found, also the performances in terms of power can be evaluated:

- Total Dynamic Power: 9.39 mW
- Cell Leakage Power: 331.39 µW

Starting from this synthesized netlist it is possible to extract the Verilog netlist. For this purpose, the whole design was flattened in order to have in only one Verilog file the description of all the designed cells.

5.3 Netlist generation

Starting from this synthesized netlist it is possible to extract the Verilog netlist. For this purpose, the whole design was flattened in order to have in only one Verilog file the description of all the designed cells.
At this point, it is possible to simulate the obtained netlist, in order to verify if it still behaves like the RTL description previously simulated. The same testbench as before was employed in Modelsim simulation, but in this case the clock period was replaced with the minimum one, previously estimated (**1.45** ns). The obtained waveforms are the same shown in Figure 6, so the synthesized netlist still behaves correctly.

5.4 Place & Route

We finally perform the physical layout of the circuit. The same consideration done in subsection 4.4 are valid. Result of Innovus synthesis in shown in Figure 7.

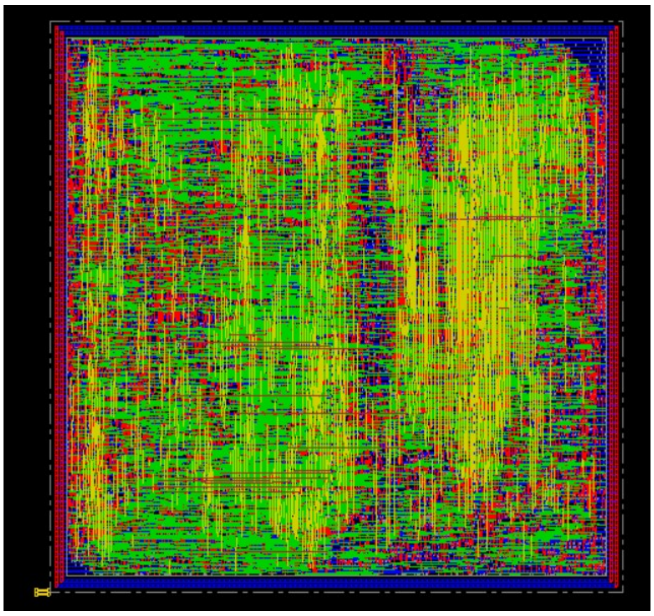

Figure 7: Layout of the RISC-V (with ABS)

Again, in order to analyze timing and integrity of our circuit, **parasitics** (i.e. resistances and capacitances) **for each metal wire are extracted**.
Inserting a clock period of 3 ns, from timing reports, **no negative slack times have been recorded**.
The final step consists in looking for **geometry and connectivity violations** and none have been spotted. Innovus consider not only the capacitance of each block of the design as done in the previous step, but also parasitics resistance and capacitance due to the interconnections are used to evaluate each power consumption contribution.
After performing placing and routing operations, we evaluate again power consumption of our lite-RISC V.
We obtain the result shown in Table 9.

Power	Value [mW]	Percentage
Total Internal Power	3.75551050	66.5075%
Total Switching Power	1.59143628	28.1832%
Total Leakage Power	0.29980437	5.3093%
Total Power	5.64675113	

Table 9: Total Power Consumption after Place & Route

The clock used for this synthesis is different with respect to the one used in 5.2, therefore no comparison between can be performed.