Politecnico di Torino

III Facoltà di Ingegneria

Laboratory 3
RISCV Processor

Master degree in Electronic Engineering

Authors: Group 21

Dilillo Nicola S284963
Moncalvo Stefano S290315
Carrano Lorenzo S281565

GitHub Repository

Contents

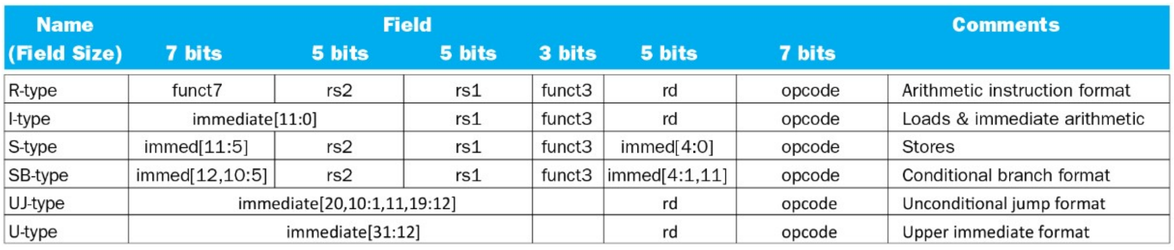
CHAPTER 1

Introduction

The goal of this laboratory is to design and synthesize a RISC-V-lite processor. The ISA of this processor includes:

- R-Type instructions;
- I-Type instructions;
- S-Type instructions;
- SB-Type instructions;
- UJ-Type instructions;
- U-Type instructions.

The format of the instructions is summarized in the following image.



Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Figure 1.1: **Instruction Format**

In particular, the processor is required to support a subset of the RV32I ISA, which corresponds to:

- **Arithmetic:**
  - **add:** R-Type instruction performing the addition between the source registers.
  - **addi:** I-Type instruction performing the addition between a source register and an immediate value.
  - **auipc:** U-Type instruction that adds the content of the program counter with an immediate value.
  - **lui:** U-Type instruction that stores the immediate value in the destination register.

- **Branches:**
  - **beq:** SB-Type instruction used to update the value of the program counter if the content of the two source registers is equal.
- **Loads:**
  - **lw:** I-Type instruction used to read a word from memory, computing the address using the immediate value. The word is stored in the destination register.
- **Shifts:**
  - **srai:** I-Type instruction performing the arithmetic right shift of the content of the source register, by a number of positions corresponding to the immediate value.
- **Logical:**
  - **andi:** I-Type instruction performing the logical AND between the source register and the immediate value.
  - **xor:** R-Type instruction performing the logical XOR between the two source registers.
- **Compare:**
  - **slt:** R-Type instruction that compares the content of the two source registers.
- **Jump and link:**
  - **jal:** J-Type instruction performs a jump to the instruction indicated by the sum between the PC and the immediate value. The value of the PC before the jump, increased by 4, is stored in the destination register.
- **Stores:**
  - **sw:** S-Type instruction that stores the value of the source register in the memory, whose address is computed using the immediate value.

The processor uses a 32-bit parallelism and is composed of 5 pipeline stages, instruction fetch, instruction decode, execute, access memory and write back. The stages will be described in the following chapters, as well as the techniques utilized to avoid hazards and in general guarantee the correct operation of the processor. Finally, the memories are not included in the design, instead they are implemented in the testbench for simulation purposes.

1.1 Schematic

The following image shows the schematic of the RISCV processor.

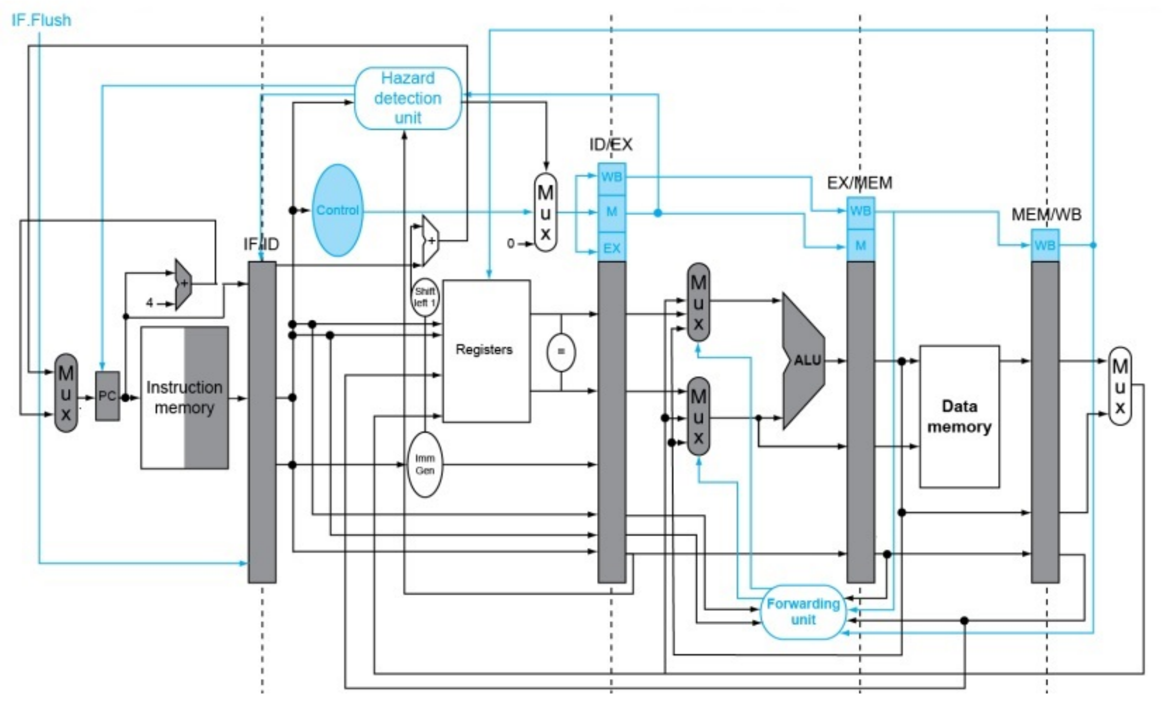


Figure 1.2: **Schematic of the design**

CHAPTER 2

Datapath

2.1 Fetch Stage

During fetch stage, an instruction is read from the instruction register accordingly to the content of the Program Counter, a 32-bit register that stores the address of the next instruction to be fetched. Fetched instructions are then passed to the next stage. Assuming the case of the execution of concurrent instructions, from one instruction to the next one the program counter is incremented by 4. This is because each instruction is encoded on 4 bytes and the instruction memory is byte-addressed, thus we have an offset of 4 bits. In case of branch instructions or jumps, the value of the program counter can be modified in other ways using various offsets and rules depending on the case, as it will discussed later.

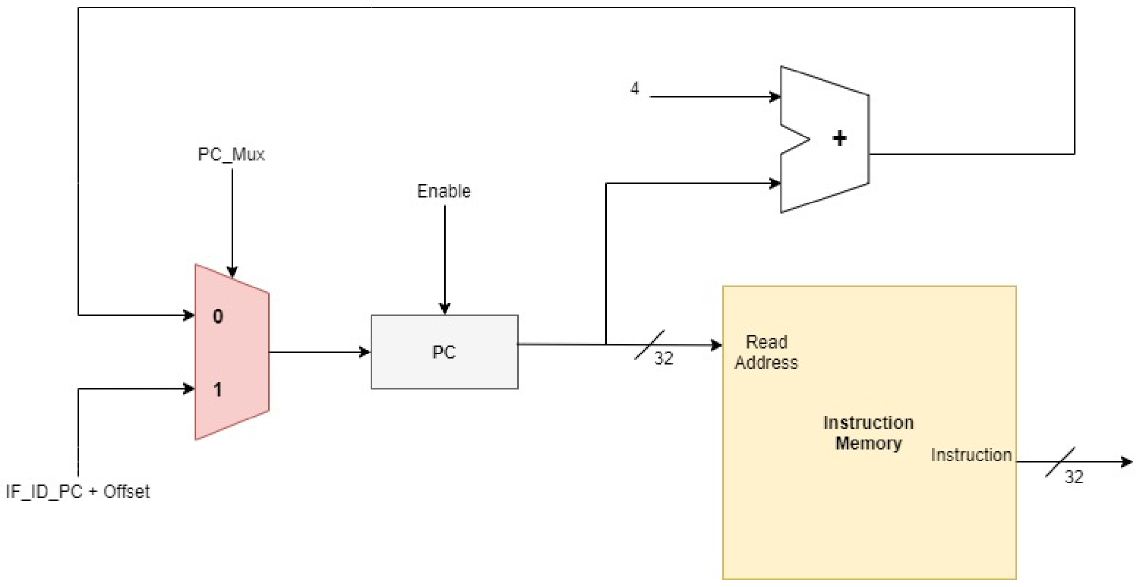


Figure 2.1: **Instruction Format**

2.2 Decode Stage

In decode stage the instruction that has been previously fetched is decoded by means of the bit-fields that compose it. Operands are decoded, be them a content of a register or an immediate value, and the control bits for the other components eventually employed in requested calculation (e.g. multiplexers or the ALU) are prepared to be sent to next stage through pipeline. In this stage stall or nope can be added due to a data hazards or a branch instruction. This behavior is better explained in the dedicated section.

2.2.1 Register File

The register file stores internal runtime values used during calculations. Some registers have specific purposes, such as the stack pointer, the thread pointer, the return address or the register r0, the only one that can't be overwritten and that always stores a value of all zeros; other registers can be used during program execution, and are said to be **general purpose registers**. The register file is synchronous with the falling edge of the clock signal and it is composed by a total of 32 registers, indexed by means of 5-bit address lines. It has been decided to keep the register file always able to provide data in reading, without any enable signal, while a dedicated signal is needed to allow writing operations. An active-high reset signal is present, in order to clear the entire content of the registers.

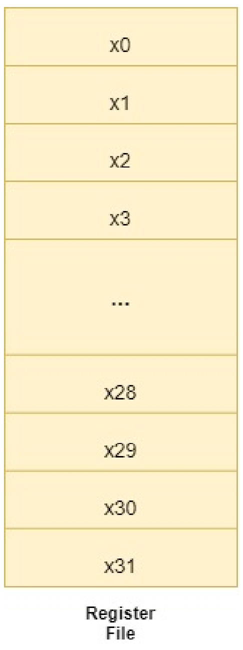


Figure 2.2: **Register File**

Register File is composed of two read data port, able to access two sources at the same time, and one write data port.

2.2.2 Immediate generator

When an instruction containing an immediate occurs this value must be extrapolated, and different opcodes require different ways to compose it. The immediate must be computed on 32 bits.

2.3 Execute Stage

During the execute stage, operands are sent to the ALU in order to perform the required computation. The operand can be provided from different stages of the pipeline and a specific unit is in charge of feeding ALU with the right value.

2.3.1 ALU

The Arithmetic Logic Unit is the computational core of the overall architecture. Since we chose to support 32 bits of parallelism for registers and data management, the ALU has been built with a parallelism of 32 bits too. The Arithmetic Logic Unit has been developed by means of standard arithmetic operations, implemented with behavioral VHDL language, in order to allow the compiler to have a certain freedom during optimization. To better manage constants, sizes and opcodes, all meaningful values have been declared in the file *myTypes.vhd*. The ALU is implemented using a process, in which inputs are employed into the proper computation by means of a control signal on 3 bits. Since the control unit is an hardwired one, this control signal is directly derived from original opcode of the decoded instruction. The operations that ALU can perform are:

- addition, sum operand A and B;
- addition PC, sum 4 to operand A;
- arithmetic shift to right, arithmetic shift operant A to right of position number equal to B;
- absolute function, elaborate the absolute value of operand A and add B to it;
- and, execute AND operation, bit by bit, between operand A and B;
- or, execute OR operation, bit by bit, between operand A and B;
- xor, execute XOR operation, bit by bit, between operand A and B.

2.3.2 Absolute Module

An entity has been written in order to allow the computation of absolute value of an integer register. This module has two input ports and one output ports. The value on the first input port is the one to convert through the absolute function. To achieve this result it sees the last bit of the signal if a zero is detected no operations are performed, while if a 1 is detected all the bits are complemented and is added a one. These are the steps to covert a number, in this case from a negative to a positive value, according absolute function. At the end the second input is summed to the convert one and the result is sent to the output. This component has been inserted into the ALU to increase its operational capacity.

2.3.3 Forwarding Unit

Forwarding Unit is a component with an important purpose: forward a previously computed result to one of the ALU inputs before it is effectively written in register file, in order to resolve a possible Data Hazard, avoiding a stall. The need to forward a certain value in one of the stages following the Execution one is detected by means of comparison between the encodings of the source registers that are passed from Decode stage to the Execute Stage, and the encodings that are actually kept in Memory and Write-back stages. If an equivalence is found, this means that the instruction that has been decoded in previous clock cycle is asking as operand a result that is not yet available in register file, but that has already been computed, thus it is possible to directly wire it to the proper ALU input via additional interconnections couple of multiplexer, as shown in figure.

Forwarding for multiplexer A is based also on the possibility to pick the value of Program Counter, it takes care about *ALUSrc_PC* signal. While the output of multiplexer B is the input of another one that chooses between the source register and the immediate value, the decision is taken through the control signal *ALUSrc*.

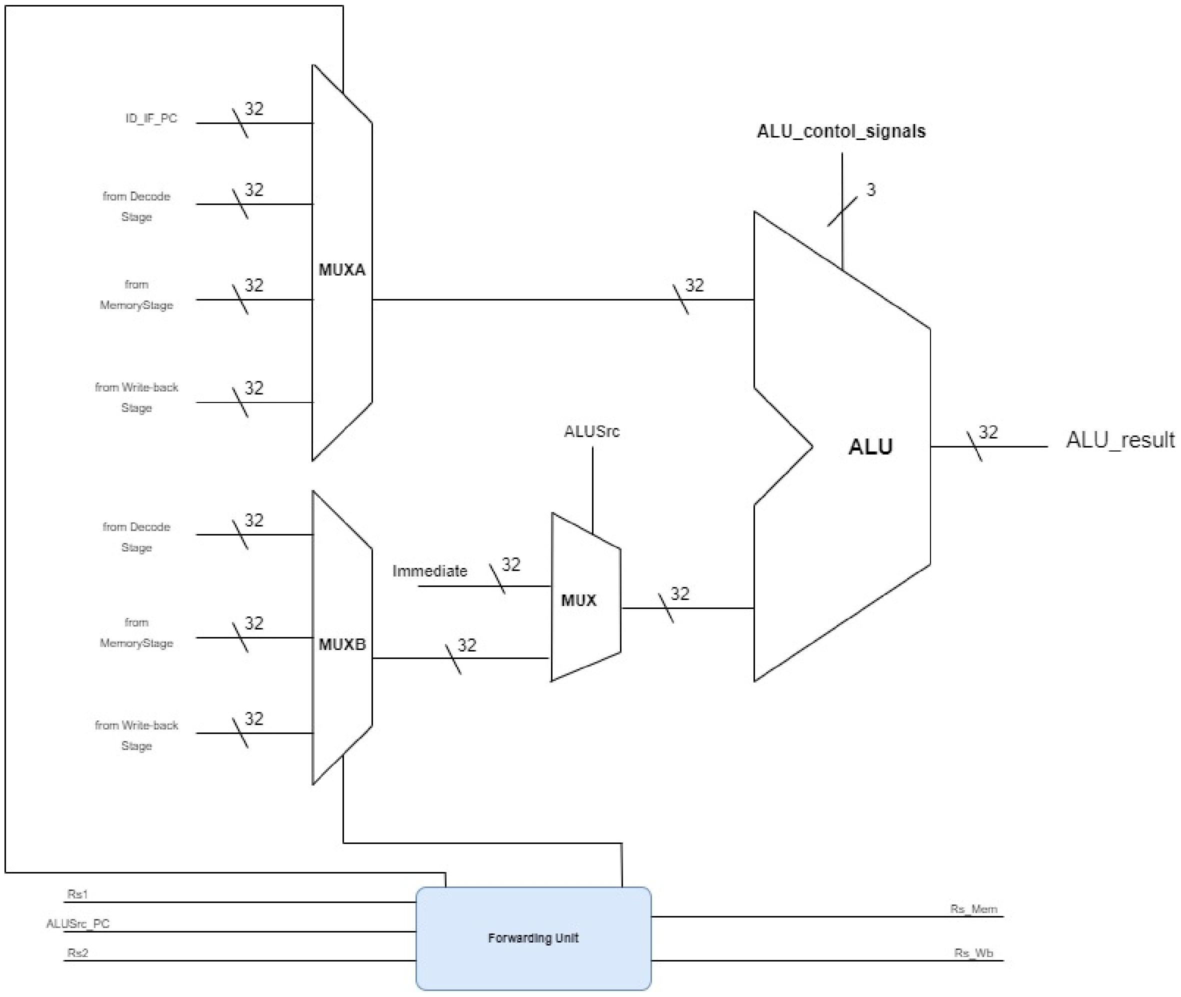


Figure 2.3: **ALU with Forwarding Unit**

2.4 Memory Stage

Memory stage is the step of the pipeline in which results are loaded/stored in the main memory. The memory is byte addressed and characterized to be Little Endian, i.e. the least significant byte is stored at a least address than the most significant one. The main memory is involved in load and store operations, that directly invoke it. It has a crucial role during when composite data need to be stored to guarantee the program correctness. Since RISCV does not require stored words to be aligned in memory, their management is simpler and requires less hardware resources.

2.5 Write Back Stage

During Write-back stage, the register file is accessed in order to store the result of the previously performed calculation inside the destination register, determined during the past decoding phase of the instruction that has generated this value. As previously clarified, the register file needs in this case that the relative write-enable signal is set in order to be able to accept a new value inside one of its locations, and the control unit is in charge of properly drive that enable signal.

CHAPTER 3

Control Unit

The control unit is the entity that regulates the control signals of the pipeline and it has been designed hardwired. It receives as input the OPCODE, as well as the FUNC3 and FUNC7 when present. According to those values it generates the correct control signals for the components present in the datapath. The signals are then pipelined in order to arrive to the corresponding component exactly when they are needed. Specifically the OPCODE specifies what kind of instruction has to be executed, while the FUNC codes are used to control the operation performed by the ALU.

In case of stall the control unit sees nothing and external component switch the value of control signal.

The control signals are:

- branch, in case of conditional branch instruction, wait for the computation of the condition before performing or not the jump;
- branch_j, in case of unconditional instruction, jump always;
- ALUSrc, to choose between value of source register two and immediate;
- ALUSrc_PC, to choose between value of source register one and program counter;
- ALU_control_signals, to choose the ALU operation;
- MemWrite, to write on data memory;
- MemRead, to read from data memory;
- RegWrite, to write in register file;
- MemToReg, to decide if write value back from read data memory or from ALU result.

CHAPTER 4

Hazards

4.1 Data Hazards Unit

The data Hazard Unit detects the need of a value that has not been read from data memory yet. This problem is detected in the Decode stage, when one of the two source register is the destination of a load instruction in the previous cycle. To avoid this problem a stall is inserted in the pipeline to wait for the load of the data from memory.

To insert the stall:

- The program counter is not updated;
- The IF/ID pipeline is not updated and keeps storing the previous instruction;
- All control signals are set to zero, regardless of the Control Unit, using a multiplexer.

The stall lasts only one clock cycle.

After reading the value from memory the forwarding unit will place it in the proper position in Execution stage.

4.2 Branch Detection

The branches are detected in Decode stage, where a dedicated adder will calculate always the new value where to jump. Two situations occur:

- unconditional branch, the jump is always taken and PC is always upgrade with the offset value;
- conditional branch, before proceeding with jump a condition must be verified, for this Laboratory exists only the possibility of the equivalence of two content registers. It takes care also about the fact that some register value could be forwarded from other stages.

If a branch is not taken nothing happens and the following instruction will load like usual. In case of branch taken the instruction before is flushed and substitute with a NOP, in the next clock cycle, the instruction loaded will be the one reached through the new offset program counter, calculated with the previous jump instruction.

NOP instruction is the following one: *addi x0, x0, 0.*

CHAPTER 5

Testbench

5.1 Memory component

For the testbench two more components have been written to mimic RISCV memories. Both of them work on the negative edge of the clock and only the needed cells have been allocated in order to speed up the simulation.

For data memory the data_memory entity has been used. It is a memory with one single address used for writing and reading, one input port for data to be written and one output port for data to be read. Two different signal activate the writing or the reading on the memory.

For instruction memory the instruction_memory entity has been used. It has one input port for address and one output port for read instruction.

5.2 ASM code

For testbench an assembly code has been written that finds the minimum absolute value in a set of data, with cardinality six. Through a simulator it has been tested and the binary instructions code has been stored in instruction memory while the data in data memory, the two previous entities, both in the right address.

The following set of instructions is used to calculate the absolute value.

```
lw x8,0(x4)
srai x9,x8,31
xor x10,x8,x9
andi x9,x9,0x1
add x10,x10,x9
```

If the simulation discovers a right behavior at the end, while the last instruction performed an infinite loop on itself, it will be possible to see in the memory, at position seven, the value 3 that corresponds to the minimum absolute value.

5.3 ABS function

The abs operation that has been created it's not a real instruction, it can't be compiled and doesn't exist an absolute instruction for integer registers. In the previous version of asm code the absolute value was calculated through different operations while now they are now substituted in the following way.

```
lw  x8,0(x4)
abs  x10,x8,0
```

For this reason, after removing the four instruction that calculated the absolute value of a register, during ASM simulation a NOP instruction has been added to calculate the offset values used from the jumps. While, in the instruction memory, the NOP instruction has been substituted from the abs instruction, codified manually, bit by bit, to achieve the right behavior. At the end the result must be the same as in the previous version.

5.4 Conclusion

The first simulation lasts 101 clock cycles the second one lasts only 83.

CHAPTER 6

Synthesis and Place & Route

6.1 Shynthesis

The two different designs have been synthesized using Synopsys Design Compiler, in order to obtain the maximum working frequency and an area estimation of the circuits. The following table summarizes the obtained results.

	RISCV	**Modified RISCV**
Min. Period	3.55 ns	3.6 ns
Max. Frequency	281.7 MHz	277.8 MHz
Area	14006.23 μm^2	14802.1 μm^2

Table 6.1: **Design Compiler Reports**

As expected, the modified design with the added component occupies more area. Moreover, since the added hardware is placed along the critical path, the maximum frequency is also slightly lower; however this is compensated by the reduced time needed for the computation of the instruction.

The report of the *elaborate* command lists the inferred memory devices inserted in the netlist. Every element was checked to avoid the insertion of latch instead of flip-flops. Finally, both synthesized netlists were simulated with Modelsim to confirm they still behaved correctly.

6.2 Place & Route

After the synthesis of the circuits, place & route was performed using Cadence Innovus. This requires to perform the following steps:

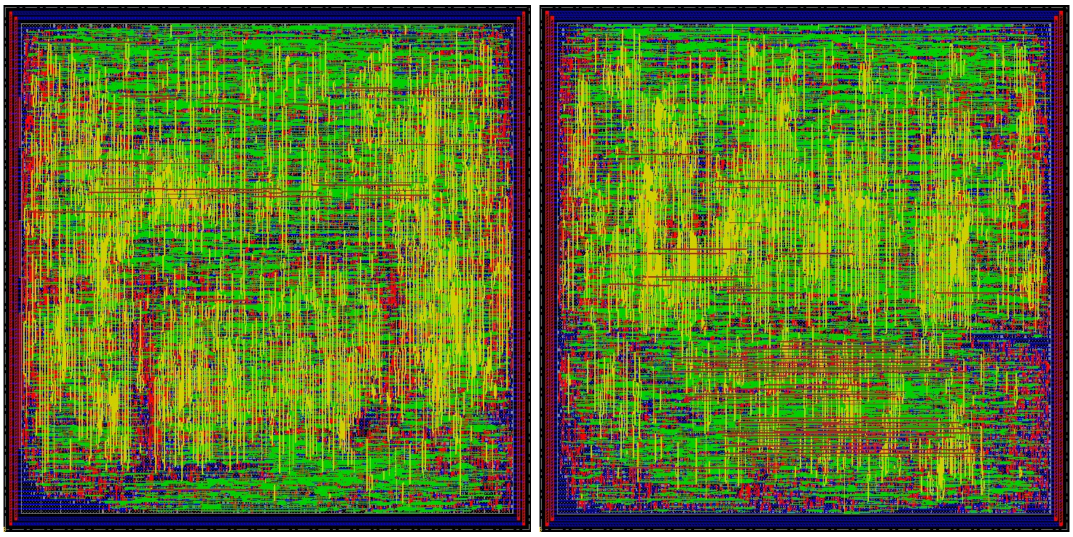
- Importing the design;
- Floorplanning;
- Power planning and routing;
- Cell placing;
- Signal routing;
- Timing and design analysis.

The procedure returned no violations in connections and geometry. The resulting netlist was saved together with the final gate count.

	RISCV	**Modified RISCV**
Area	13533.0 μm^2	14133.6 μm^2

Table 6.2: **Innovus Reports**

Innovus was able to optimize the netlists and save a noticeable amount of area on both designs, with respect to the value obtained with Synopsys. Also in this case, the produced netlists were simulated to verify the compliance with the original design.



(a) RISCV (b) Modified RISCV

Figure 6.1: **Place and route of the two designs**