Politecnico di Torino

III Facoltà di Ingegneria

Exercises and Homeworks for the course Integrated Systems Architecture

Master degree in Electrical Engineering

Authors: Group 20

Alessandro Lovesio, Lorenzo Poloni, Simone Pont

January 31, 2021

Many thanks to Prof. Mariagrazia Graziano for providing us with this template.

Contents

Link to our group repository:
`https://github.com/AleLovesio/Laboratory_ISA_Group20.20`
Direct link to the folder of Lab 3 in the same repository:
`https://github.com/AleLovesio/Laboratory_ISA_Group20.20/tree/main/Lab3`

0.1 Introduction

In this third lab the goal is to implement a functioning RISC-V-lite processor which is able to execute a subset of the RISC-V instruction set. In particular:

- add (R-type)
- addi (I-type)
- andi (I-type)
- auipc (U-type)
- beq (SB-type)
- jal (UJ-type)
- lui (U-type)
- lw (I-type)
- slt (R-type)
- srai (I-type)
- sw (S-type)
- xor (R-type)

Instructions are carried out on a classic pipelined Harvard architecture which employs two separate memories (one for data and one for instructions) and is divided into five well-known stages: IF, ID, EX, MEM, WB. While the insertion of pipeline registers drastically increases performance, it also introduces some problems that will be discussed and handled further in the report. In the following chapters three versions are described. Each new version is the enhancement of the previous one as new features are added. For the sake of comparison, all versions will be simulated whereas the synthesis and the physical design are restricted to the last two versions only.

0.2 A simple first RISC-V-lite implementation

The first implemented version of the RISC-V-lite is represented in the following picture:

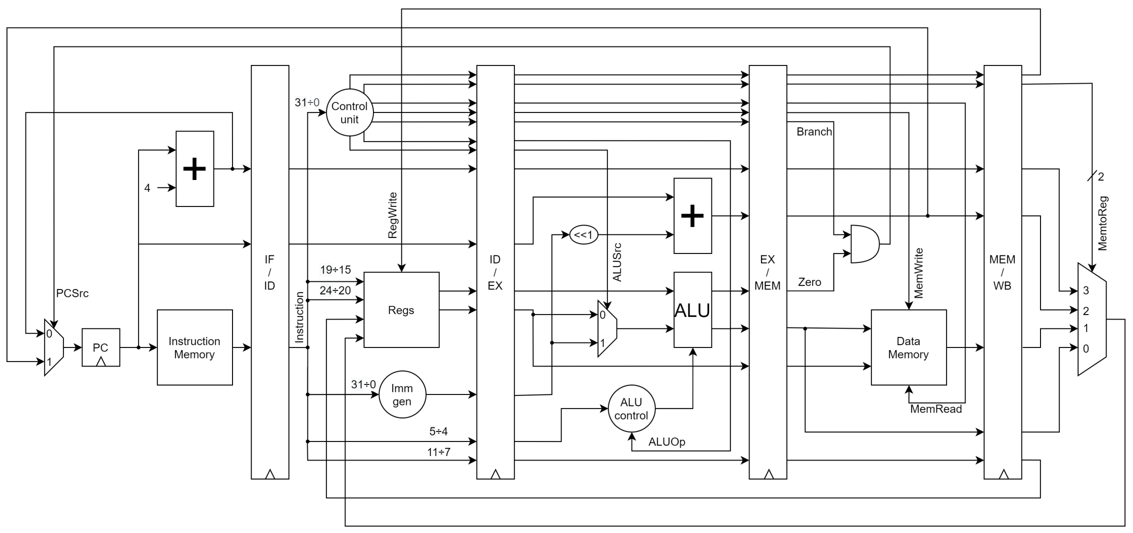


Figure 1: First version of the RISC-V-lite processor

0.2.1 ALU and ALU CU

The Arithmetic Logic Unit (ALU) is one of the most important components in the datapath as it performs almost all the operations required by the instructions during the EX phase. Although the ALU can be used by every instruction, we chose to have a special circuit handle the operations related to the Program Counter (PC) instead of directly connecting it to the ALU. Incremental PC flow is altered by *beq* and *jal*, which are branch instructions. As for the former, the address of the current instruction stored in the PC is increased by the value found in the immediate field shifted to the left by one. The adder used in the process is placed in the EX stage but it is not part of the ALU, which has its own adder. The updated PC goes through the EX/MEM register to the MEM stage where it is sent back to the input of a multiplexer connected to the PC. The value containing the new address is validated by a signal generated by an AND port which takes the branch signal from the CU and the Zero Flag from the ALU. In this case the ALU is used to compare the numbers in input and detect equality. An active zero flag confirms the equivalence and the jump is allowed. The *jal* instruction, instead, has to save the return address (PC+4) in the RF. This is achieved by propagating the signal coming out of another special adder, which increments the PC by four, all the way to the WB stage where the value is saved in a destination register. This instruction does not make use of the ALU. In the hazard-aware implementations (i.e. the following versions), the *beq* instruction is also completely carried out outside the ALU as an equality comparator is inserted directly at the output of the RF during the ID phase. This is done to be aware of hazards earlier in the pipeline so as not to significantly reduce performance. Although it is not a jump instruction, *auipc* accesses the PC, adds the immediate value and saves the result in the RF. Also in this case, the instruction is completed outside the ALU using a special adder.
Inside of the ALU, a lot of operations are managed. For each one of them, a different code is sent in input to let the ALU know what to do. Moreover, some instructions share the same operators. This unit contains: an adder, an AND gate, an XOR gate, a combinational right barrel-shifter and a comparator which checks if input A is less than input B and sets the MSB of the output to 1 if the condition is true (slt instruction). All the operators are described behaviorally. The only output flag,

as stated before, is the zero flag. Overflow is not handled. The output result propagates through the MEM stage where it can be written in the data memory or also through the WB stage to be stored in the RF. The inputs for the ALU come directly from the RF. One of the two is shared with the immediate generation unit by means of a multiplexer.
The ALU opcode is 3 bits long and is generated by the ALU Control Unit, which is driven by the main control unit. It is true that, at first glance, the ALU CU and the main CU could be merged together into a single unit. However, separating them results in less complex circuits which perform better than a single block. The ALU CU takes four input bits of which two coming from the CU and two from the bits 5 and 4 of the instruction opcode. It was found that these bits are enough to encode all the operations. No particular rule or method was employed to map the bits from one CU to the other and finally to the ALU.

0.2.2 Immediate Generator

The immediate generator is a combinational block that takes in input the instruction and computes, if possible, the required immediate to be used by the ALU or for the JUMP operations. It is fairly simple: first the unit identifies the instruction type thanks to the OPCODE and FUNCT3 instruction segments while in parallel every possible immediate form, one for each instruction type, is generated. Then the correct immediate is selected and is multiplexed to the output.

0.2.3 Control Unit

The control unit has a behavior that is very similar to a lookup table. As every control signal is solely dependant on the instruction and it is known beforehand, the current instruction is recognized and the control signals are raised or lowered according to the following table:

	RegWrite	MemtoReg	MemRead	MemWrite	Branch	ALUSrc	ALUOp
add	1	00	0	0	0	0	00
addi	1	00	0	0	0	1	00
auipc	1	10	0	0	0	-	- -
lui	1	00	0	0	0	1	10
beq	0	- -	0	0	1	0	- -
lw	1	01	1	0	0	1	00
srai	1	00	0	0	0	1	10
andi	1	00	0	0	0	1	01
xor	1	00	0	0	0	0	01
slt	1	00	0	0	0	0	11
jal	1	11	0	0	1	-	- -
sw	0	- -	0	1	0	1	10

0.3 Test-bench

The test-bench has the role of generating the clock signal and manage the main memory needed for the correct behavior of the RISC-V. Considering that the generated output is directly written into the data memory, we don't need a "data_sink" or a "checker" unit because the correctness can be easily verified by looking at the memory.
First a reset signal is activated to correctly bring the RISC-V to a known state and then the clock is supplied to the DUT in order to start its computation. During the initial reset phase the memories are initialized with the correct data, for this reason the reset signal remains active until both the memories are ready and a certain time is passed ($\frac{3}{2}T_{CK}$).

0.3.1 Memories

In this laboratory the main memories are not included in the RISC-V design and are considered part of the test-bench, for this reason we decided to simplify their management to easily interact with them. In particular, we implemented two different memories, for data and instructions, that are independent and have some differences. The data memory is a byte addressable asynchronous memory that can be both read and written from the outside using the two enable signals *WE* and *RE* (if both are active the read has priority). This memory can be reset but in our case this functionality is not used. The instruction memory is again a byte addressable asynchronous memory but it can be only read from the outside (ROM-like behavior) and has no reset signal. In both cases, even if the two memories are internally byte addressable, the I/O data are on 32 bits, so they occupy 4 consecutive memory locations using the Little Endian approach. To easily interface the memories with the RISC-V, also the address is represented on 32 bit even if the internal addressable locations are only 200 (greatly lower than the maximum addressable ones that are $2^{32} - 1 = 4.294.967.295$). We decided to have a lower number of locations not to work with a big and unused memory. In case of data memory, where the received addresses are affected by an initial offset due to the assembly code, we decided to subtract the correct value from the address before accessing the memory.
Considering that we need an initialization for both the memories, we decided to do it in a more flexible way rather than simply initializing them in the VHDL code. In particular, with the help of the Java executable given to us, we generated two .txt files containing the binary code and data that have to be written and then we initialized the memories by reading these files. This easily let us change the binary code (for example adding some nop instructions) without the need to change the VHDL component but only acting on the external .txt file. The initialization phase is done automatically at the start and it takes no simulation time to complete. When it is done it also asserts a signal that notifies it to the test bench.

0.4 Add-ons: Forwarding and Hazard Units

To improve the performance of our circuit and to automatically detect and solve hazards, we decided to modify the design and add two new units: the Forwarding and the Hazard units. In the following picture the final structure of our design is reported:

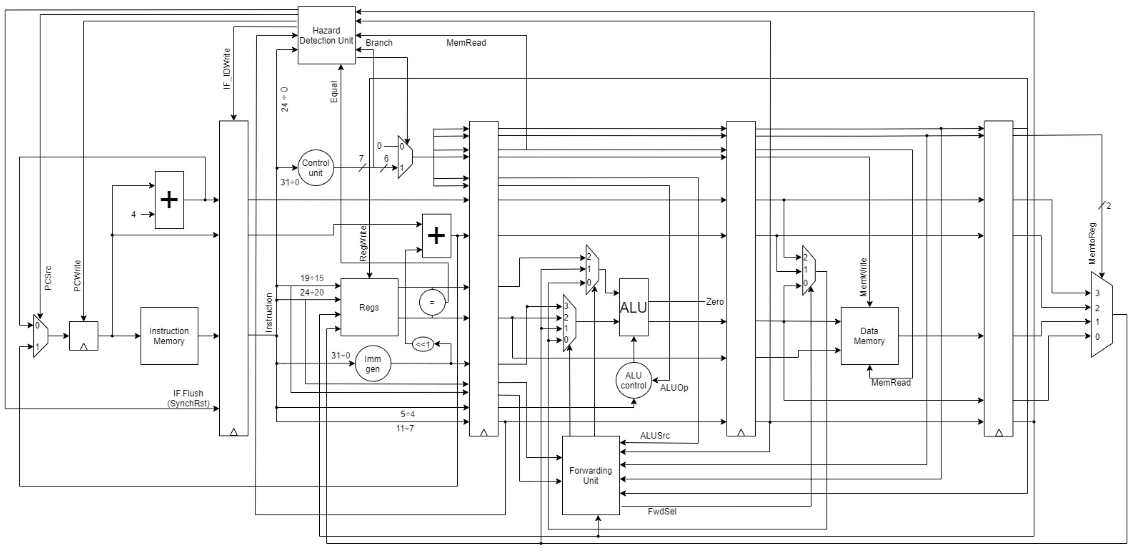


Figure 2: Second version of the RISC-V-lite processor: forwarding and hazard unit added.

0.4.1 Forwarding Unit

The main purpose of this unit is to identify when a RAW data dependency between two consecutive instruction is present and try to solve it by simply connecting the needed result directly to the ALU input without passing through the register file. To obtain this, we can divide the computation into two parts: identify the data dependency and choose the correct data to put in the correct ALU input. The first can be easily performed by following some steps. In particular:

- Checking if one of the previous instructions (on MEM and WB stages) will write to a register.
- Checking if the destination register of that instruction (in EX/MEM or MEM/WB) is equal to one of the two source registers of the instruction in ID/EX;
- Checking if the destination register is not x0.

After that we have to choose the correct data that has to be forwarded. In case of the need to forward the result present in the WB stage, this decision is automatically done by taking the output of the final mux. However, if we need to forward the result at MEM stage we need another mux to choose the correct data between the ALU result, the PC+IMM and the PC+4. This mux is directly controlled by the forwarding unit considering the value of the *MemToReg* signal at that stage.
Finally we have also to properly manage the input muxes of the ALU to work with the wanted data. The two selection signals are obtained considering the identified data dependency and the value of *ALUSrc* using a simple combinational logic.

0.4.2 Hazard Unit

As previously said, a pipelined architecture offers numerous advantages in terms of throughput and power consumption. However, a new problem arises: hazards. If we look at several real-life examples

of codes, we see that programs are rarely fully sequential. In fact, they make use of jump and branch instructions in loops, subroutine calls etc. Moreover, the temporal principle of locality suggests that the program might need to access the same memory element multiple times in a short time interval. These are some of the problems that cause the flow of the pipeline to stop abruptly. We say that a hazard occurred. In simple architectures, hazards cannot normally cause errors but they heavily affect performance as they nullify the advantages of pipeline. Several solutions to limit this problem exist. Some rely on the compiler to rearrange the instruction, while others make use of a circuit that is able to detect such a condition. Some systems use both software and hardware implementations. In our case, a Hazard Unit was designed. This circuit is able to detect hazards by reading specific input signals. The outputs drive registers and muxes in order to either stall (*nop* insertions) or flush (reset registers) the pipeline when necessary. It is important to note that there are three types of hazards. A structural hazard occurs when two (or more) instructions that are already in pipeline need the same resource. Since we are using a Harvard architecture (i.e. two separate memories), this situation will never happen. Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. This kind of hazard is managed by the forwarding unit. However, in presence of a Load-Store data hazard, the bypass cannot be used. This kind of problem arises when an instruction which reads a value from a register follows a load from memory instruction (*lw*) that writes a value in the same register. This happens because the word will be available after the MEM stage rather than the EX stage. When *lw* is in the MEM stage the following instruction will be in the EX stage and will expect the register to contain a valid number. In this situation, we have no other choice but to stall the pipeline. This is done by inserting a *nop* instruction between the *lw* and the next instruction. To detect the problem and correct it, the Hazard Unit identifies the *lw* instruction by the *MemRead* signal from the CU and checks that its destination register matches one of the source registers of the next instruction. In case of hazard, the PC and the IF/ID register are disabled and the control signals from the CU to the following stages are set to 0.
Control hazards occur when branch or jump instructions are present. In order to understand whether to jump or not, we are supposed to wait until the EX phase. In the meanwhile the pipeline has brought instructions into the pipeline that must subsequently be discarded. In this situation, the Hazard Unit has to flush the pipeline. Normally, a penalty of clock cycles after the jump would be required if the branch is taken. This is often unacceptable. To reduce the branch delay, the branch decision is taken earlier in the pipeline, in the ID stage, to be precise. Thanks to this solution only one clock cycle is lost. To deal with the problem, the Hazard Unit detects the presence of *beq* or *jal* instructions. In case of *beq*, it also checks that the conditions for branch are verified. If a control hazard occurs, the IF/ID register is reset and the mux at the input of the PC is switched to the value equal to the new address.
There is one last case that we have to consider to avoid hazard-related issues: when we have a *beq* instruction there is the possibility that one of the two source registers (or even both in some very unusual cases) need to be updated by the previous instructions that are not yet completed due to the pipelined structure. This RAW hazard can lead to wrong branch decisions thus creating quite a few troubles. The problem is present only for the *beq* instruction because we moved the branch decision to the decode stage and so the forwarding unit is no more useful to solve the data dependency. For this reason, we decided to implement a simple circuit that can identify this situation and stall the fetch and decode stages until the dependency is solved. This behavior is similar to the insertion of some *nop* operations before the *beq* instruction to correctly execute it, so it implies a loss on performances. However, considering that the implementation is very simple, we decided to use this solution that modifies only a little the hazard unit and the complete RISC_V structure. The steps to follow in order to solve the problem are:

- Identify the presence of a *beq* instruction on the decode stage.
- Identify the presence of the RAW data dependency between the destination register of the

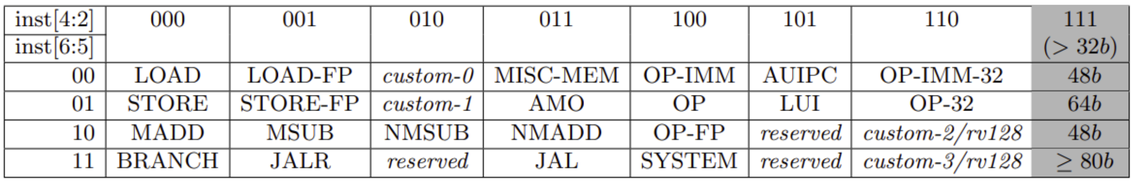
instructions in EX, MEM or WB stages and at least one of the source registers of the *beq*.

- In case of the two previous points are verified stop the fetch and decode stages while the other parts of the pipeline execute normally.
- When the RAW data dependency is no more present, restart the fetch and decode stages returning to the normal execution.

To stop the fetch and decode we use the same signals of the hazard unit described before. However in this case we limit to stop the PC update and the IF/ID register sampling without resetting them. The *nop* bubble inserted in the following stages is obtained acting on the control signal mux, as described before for the case of data hazard.

0.5 Definitive Version with Modulus Unit

Finally, a new unit dedicated to the execution of a new special instruction, the absolute value instruction, was implemented. The newly implemented instruction, denominated *ABS*, has been added to the instruction set with the opcode *"0001011"* and funct3 code *"000"*. This opcode was chosen as it reserved specifically for custom instructions, quoting the specifications at Chapter 25 (RV32/64G Instruction Set Listings): *Major opcodes marked as custom-0 and custom-1 will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format.* In particular we have chose the custom-0 opcode defined in the following table:



inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80*b*

Table 25.1: RISC-V base opcode map, inst[1:0]=11

Figure 3: Table with the possible opcodes of the RISC-V

To implement the new instruction, the previous version with the Hazard and Forwarding Units was modified in order to introduce into the RISC-V an unit additional to the ALU called SFU (Special Functions Unit). This unit can be used to compute the absolute value of a number in input, so this allows for a much faster execution of the given algorithm. Here is the schematic of the updated RISC-V:

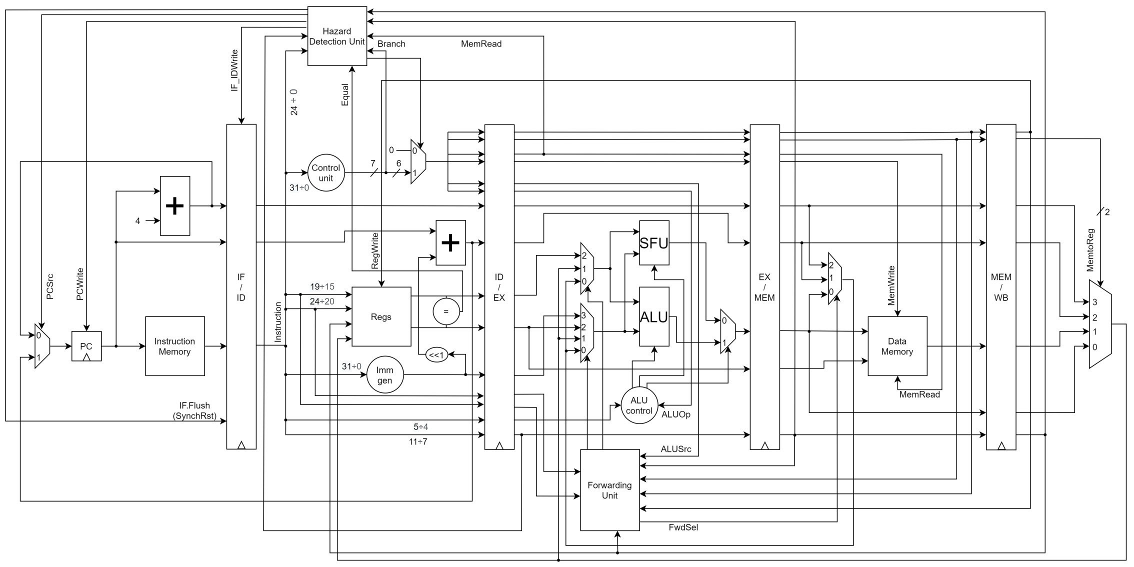
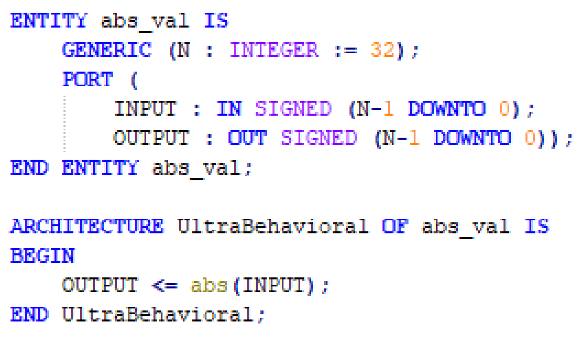


Figure 4: Final version of the RISC-V-lite processor

As it can be seen, the SFU takes the same inputs as the ALU while its output is multiplexed with ALU output so that only the correct one is forwarded. New control signals have been added to control both the output MUX as well as the SFU itself via the SFUOP signal. To control these signals, both the ALU_CU and the Control Unit have been updated so that the new operation can be correctly performed. The control signals generated by the Control Unit are now as follows, with the row relative to the new instruction highlighted in green:

	RegWrite	MemtoReg	MemRead	MemWrite	Branch	ALUSrc	ALUOp
add	1	00	0	0	0	0	00
addi	1	00	0	0	0	1	00
auipc	1	10	0	0	0	-	- -
lui	1	00	0	0	0	1	10
beq	0	- -	0	0	1	0	- -
lw	1	01	1	0	0	1	00
srai	1	00	0	0	0	1	10
andi	1	00	0	0	0	1	01
xor	1	00	0	0	0	0	01
slt	1	00	0	0	0	0	11
jal	1	11	0	0	1	-	- -
sw	0	- -	0	1	0	1	10
abs	1	00	0	0	0	1	01

The unit, even though it is only able to perform the absolute value operation, was structured very similarly to the ALU, with two input operands, a SFUOP signal to be used as opcode, an Output and a zero signal. While the second input operand and the zero signals are not used and the opcode signal is redundant as there is only a single operation that can be performed, all these signal were implemented to keep the unit more generic and more easily upgradable.
The actual combinatorial element that computes the absolute value of the operand in input has been implemented in three different ways. The different architectures have been tested and synthetized seprately to verify both their functionality and their performance.
The first implemented version, called *UltraBehavioral*, takes advantage of the ieee library function *abs()*, which computes the absolute value of the value provided in input. The corresponding VHDL code is the following:



```
ENTITY abs_val IS
    GENERIC (N : INTEGER := 32);
    PORT (
        INPUT : IN SIGNED (N-1 DOWNTO 0);
        OUTPUT : OUT SIGNED (N-1 DOWNTO 0));
END ENTITY abs_val;

ARCHITECTURE UltraBehavioral OF abs_val IS
BEGIN
    OUTPUT <= abs(INPUT);
END UltraBehavioral;
```

Figure 5: VHDL of the first version implemented with the abs macro

The second implemented version, called *Behavioral*, is the one in the following picture and it consists of a simple circuit used to change the sign of the operand (NOT gates and incrementer) which takes advantage of the two's complement notation of the signed number plus a multiplexer used to choose between the operand and the sign inverted operand.

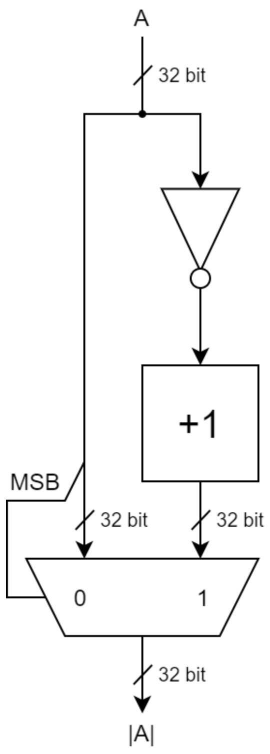
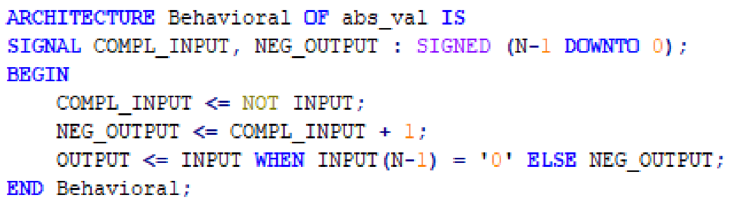


Figure 6: Second implemented version

The corresponding VHDL is the one reported in the following picture:



```
ARCHITECTURE Behavioral OF abs_val IS
SIGNAL COMPL_INPUT, NEG_OUTPUT : SIGNED (N-1 DOWNTO 0);
BEGIN
    COMPL_INPUT <= NOT INPUT;
    NEG_OUTPUT <= COMPL_INPUT + 1;
    OUTPUT <= INPUT WHEN INPUT(N-1) = '0' ELSE NEG_OUTPUT;
END Behavioral;
```

Figure 7: VHDL of the second implemented version

After the synthesis performed to evaluate the circuit, it has been discovered that Synopsys actually compiles the described circuit into the following one by implementing a Design Ware provided subtractor:

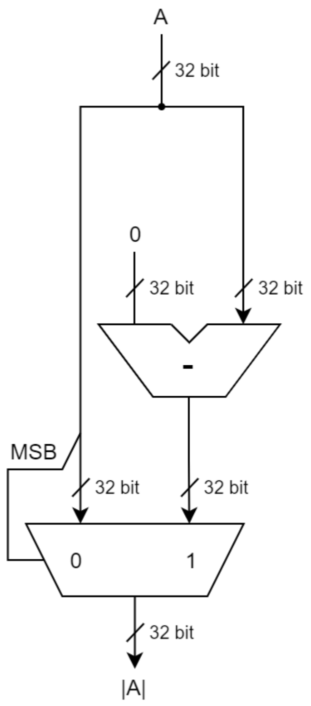


Figure 8: Second implemented version, compiled by Synopsys

It has also been noticed that this second version, once synthesized, is exactly identical to the first version. We speculate that the IEEE function was implemented in the same way as we did or, at least, in a very similar manner. Finally, a third version, called *Behavioral_codelike*, has been implemented by resorting to the technique adopted by the provided assembly code. The schematic of this architecture is represented in the following image:

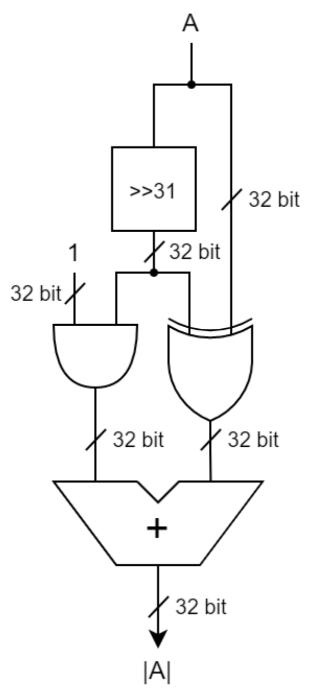


Figure 9: Third implemented version

The VHDL implementation of the circuit is reported in figure 10:

```
ARCHITECTURE Behavioral_codelike OF abs_val IS
SIGNAL COMPL_INPUT, MSB_CIN, SIGN_VECT : SIGNED (N-1 DOWNTO 0);
BEGIN
    SIGN_VECT <= SHIFT_RIGHT(INPUT, 31);
    COMPL_INPUT <= INPUT XOR SIGN_VECT;
    MSB_CIN <= TO_SIGNED(1, 32) AND SIGN_VECT;
    OUTPUT <= COMPL_INPUT + MSB_CIN;
END Behavioral_codelike;
```

Figure 10: VHDL of the third implemented version

As with the second version, Synopsys optimized the circuit and implemented the architecture differently than described. In particular the right shift was implemented just by repeating 32 times the MSB while the AND was simplified as on term was always a constant. The final result is reported next:

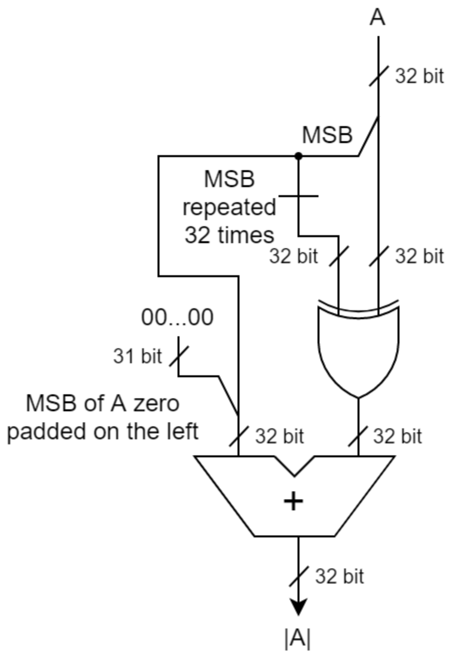


Figure 11: Third implemented version, compiled by Synopsys

After performing the synthesis with *Synopsys* a simulation was performed with *Modelsim* with a set of a few limit case values and another 100 random input values to verify the functionality and also to obtain the activity data of the internal nets to evaluate power consumption. The final Area, Timing and Power consumption results are summarized in the following table:

Version	Area occupation	Critical path delay	Power consumption
Ver. 1 "UltraBehavioral"	$224.238\mu m^2$	$1.83ns$	$45.809\mu W$
Ver. 2 "Behavioral"	$224.238\mu m^2$	$1.83ns$	$45.809\mu W$
Ver. 3 "Behavioral_codelike"	$185.136\mu m^2$	$2.00ns$	$38.165\mu W$

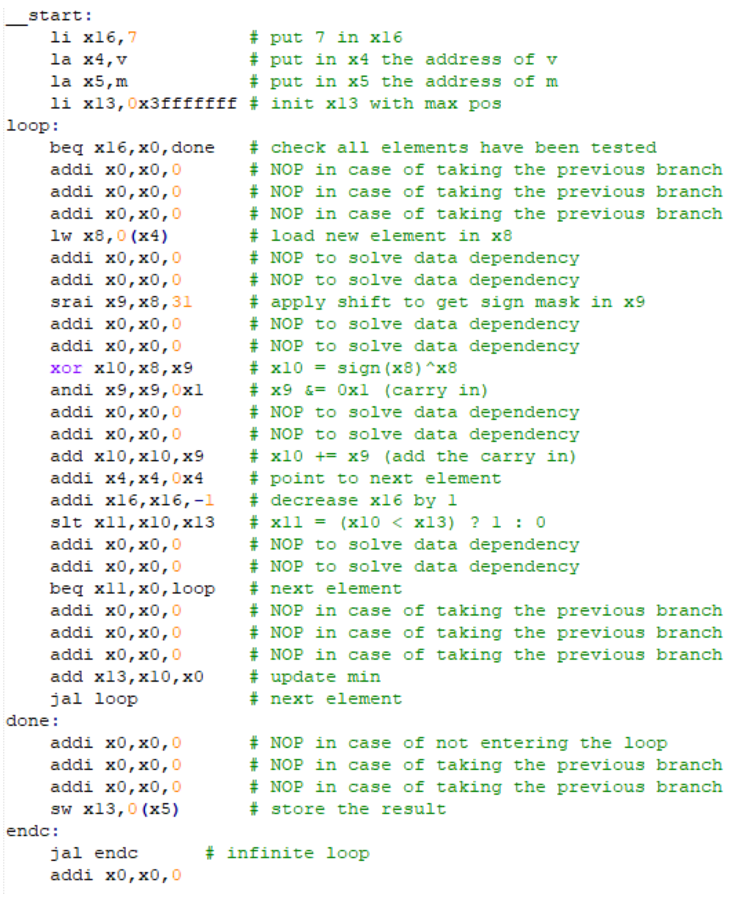
We have picked the third version for the final RISC-V implementation for three reasons: lower area occupation, lower power consumption and negligible critical path difference.
In the next section we will see how the assembly code has been modified to account for the new instruction. The change has resulted in three less instructions to be executed at every cycle, or a 25% reduction of needed instructions each cycle. This can lead to very consistent performance improvement for a large dataset.

0.6 Simulation

In the previous sections we described the different components of the RISC-V, now we have to consider the evolution of the design starting from the first simple version and reaching to the final one. At each step we performed simulations to verify the correct behavior of our project before implementing new features and optimizations. We will talk about three versions of the RISC-V design, all correctly working but with some important differences in terms of performances and internal structure. Before starting to talk about the first one we need to underline that, during the design evolution, we passed through additional different versions that are in the middle between these three. However, the differences were often very small and so we will analyze better only the three most important.

0.6.1 First Version

The first version of the RISC-V we implemented is the simplest but also the least efficient one. In this solution, the Hazard and Forwarding units are missing and the absolute value calculation is not done using the special dedicated unit. The scheme of this first version is the one reported in the first part of the report by figure 1.
Considering that both the Forwarding and the Hazard unit are missing we need to manually add some *nop* instructions (where needed) to avoid hazards. To do this we acted on the given assembly code to obtain a final set of instruction with no possible hazards, the modified code is the following (remembering that in RISC-V architecture *nop* instruction are represented as *addi x0,x0,0*):



```
__start:
    li x16,7          # put 7 in x16
    la x4,v           # put in x4 the address of v
    la x5,m           # put in x5 the address of m
    li x13,0x3fffffff # init x13 with max pos
loop:
    beq x16,x0,done   # check all elements have been tested
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    lw x8,0(x4)       # load new element in x8
    addi x0,x0,0      # NOP to solve data dependency
    addi x0,x0,0      # NOP to solve data dependency
    srai x9,x8,31     # apply shift to get sign mask in x9
    addi x0,x0,0      # NOP to solve data dependency
    addi x0,x0,0      # NOP to solve data dependency
    xor x10,x8,x9     # x10 = sign(x8)^x8
    andi x9,x9,0x1    # x9 &= 0x1 (carry in)
    addi x0,x0,0      # NOP to solve data dependency
    addi x0,x0,0      # NOP to solve data dependency
    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
    addi x0,x0,0      # NOP to solve data dependency
    addi x0,x0,0      # NOP to solve data dependency
    beq x11,x0,loop   # next element
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    add x13,x10,x0    # update min
    jal loop          # next element
done:
    addi x0,x0,0      # NOP in case of not entering the loop
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    sw x13,0(x5)      # store the result
endc:
    jal endc      # infinite loop
    addi x0,x0,0
```

Figure 12: First version of the assembly code, with all the needed *nop* bubbles

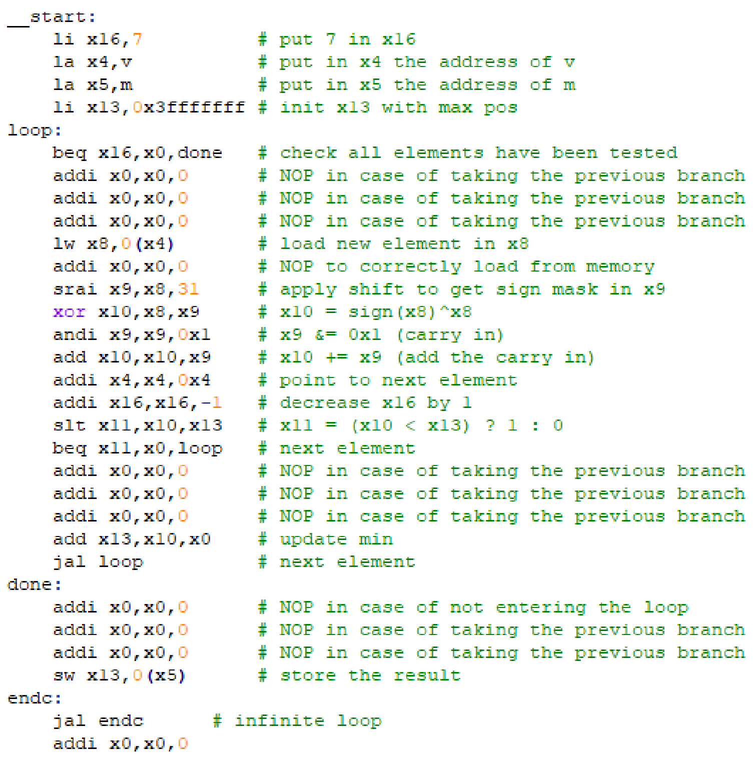
We have also to precise that for the three instructions used to save data memory addresses (1 *li* and 2 *la* instructions) we had to act directly on the generated binary code to insert the needed *nop* bubbles. The reason is that in those cases, the single assembly instruction is divided into two different RISC-V ones with an internal data dependency, in particular we have that:

- *li* instruction becomes the sequence of *lui* and *addi*.
- *la* instruction becomes the sequence of *aiupc* and *addi*.

After having correctly inserted all the needed *nop* bubbles, we simulated the complete RISC-V using the test-bench described before and we verified its correct behavior. This first basic version was used only to check the correctness of the general RISC-V structure before starting the optimizations and so it wasn't synthesized nor placed.

0.6.2 Second Version

The passage from the first to the second version was done with multiple little steps, adding time by time only small modifications to the circuit in order to obtain an improvement in performance. The first major change was the addition of the forwarding unit that allowed us to remove some *nop* bubbles needed to solve RAW data dependencies. The starting assembly code was modified and shortened, leading to a decrease in time wasted and so to an increase in performance. The modified assembly is reported in the following figure (we have yet to consider the presence of some *nop* inside the *li* and *la* instructions, not reported in this code):

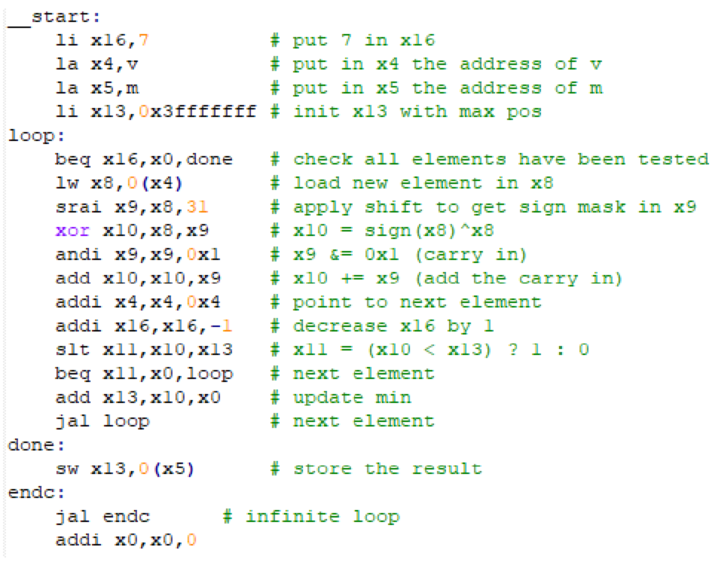


```
__start:
    li x16,7          # put 7 in x16
    la x4,v           # put in x4 the address of v
    la x5,m           # put in x5 the address of m
    li x13,0x3fffffff # init x13 with max pos
loop:
    beq x16,x0,done   # check all elements have been tested
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    lw x8,0(x4)       # load new element in x8
    addi x0,x0,0      # NOP to correctly load from memory
    srai x9,x8,31     # apply shift to get sign mask in x9
    xor x10,x8,x9     # x10 = sign(x8)^x8
    andi x9,x9,0x1    # x9 &= 0x1 (carry in)
    add x10,x10,x9    # x10 += x9 (add the carry in)
    addi x4,x4,0x4    # point to next element
    addi x16,x16,-1   # decrease x16 by 1
    slt x11,x10,x13   # x11 = (x10 < x13) ? 1 : 0
    beq x11,x0,loop   # next element
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    add x13,x10,x0    # update min
    jal loop          # next element
done:
    addi x0,x0,0      # NOP in case of not entering the loop
    addi x0,x0,0      # NOP in case of taking the previous branch
    addi x0,x0,0      # NOP in case of taking the previous branch
    sw x13,0(x5)      # store the result
endc:
    jal endc      # infinite loop
    addi x0,x0,0
```

Figure 13: Second version of the assembly code

After having verified the correct behavior of this intermediate version we decided to add also the hazard unit to our design. The first implementation of this unit lets us solve some control and data hazards giving us the opportunity to remove more *nop* bubbles and improve again performances.

However this implementation was not able to properly manage the *beq* instruction under certain specific conditions, so we decided to improve it as already described in the end of the dedicated section. We finally obtained the complete working hazard unit that jointly with the previous added forwarding unit let us remove all the *nop* bubbles. At this point we modified again the assembly code deleting all the *nop* instructions that we have inserted initially and the obtained result is the one reported in the following (in this case also the *li* and *la* instructions have no internal *nop*):



```
__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
```

Figure 14: Final version of the assembly code, without *nop*

The final structure obtained for this second version is the one reported in figure 2. In this case, after the simulation to verify its correct behavior, we synthesized and placed the circuit obtaining the results that are reported in the following section.

0.6.3 Third Version

This last implementation uses the final structure of the second version and adds to it the special unit created to calculate the absolute value. The changes are present mainly in the execution stage where the Modulus unit is added in parallel to the ALU. The final circuit is the one reported in figure 4 that we have described before.
To correctly use this new version of the design we need to change again the assembly code substituting the set of instructions previously needed to calculate the absolute value with the new added single instruction. The final result is the following:

```
__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,x8,0      # SPECIAL INSTRUCTION to directly compute the absolute value
    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
```

Figure 15: Assembly code considering the presence of the Modulus unit

We can notice that the assembly code sees another big reduction and this leads again to performance improvements. The cost of this is the added special unit that will occupy more area and consume power.
Using the assembly we proceeded as usual to obtain the binary code and we perform the simulation of this last version of the RISC-V. Once we checked its correct behavior we synthesized and placed it in order to compare it with the second version of our design, where no special unit is present.

0.6.4 Assembly code comparison

Before starting to analyze the results obtained with the synthesis and placing we can briefly compare the assembly of the three different versions. For now we compare only the assembly code size but this gives us an idea of the evolution of our design and of the improvements we obtained. The comparison is reported in the following table (also the *nop* "hidden" inside the *li* and *la* instructions are considered):

RISC-V version	Number of useful instructions	Number of *nop*	Total number of instructions
Version 1	21	24	45
Version 2 (only forwarding)	21	17	38
Version 2 (final version)	21	1	22
Version 3	18	1	19

Table 1: Number of instructions of the assembly code.

From this table we can see how the total number of instructions lowers in each new version and this leads to better performance. In particular we have that in the first two steps the decrease is due only to the *nop* reduction while in the last one (when we pass from version 2 to version 3) the lowering is due to the reduction of "useful" instructions thanks to the presence of the special Modulus unit. In all cases a lowering of the total number of instruction should bring a faster execution of the complete algorithm and so better overall performances. However we need to consider that these reductions will not correspond perfectly to the execution time reductions but give us only an idea of the obtained improvements.

0.7 Synthesis and Place & Route

In the last part, we instructed the design compiler to synthesize our circuit. After that, the physical design was performed. As usual, the tools used are *Synopsys* and *Innovus*. These two steps are common to the two versions of our architecture.

0.7.1 Synthesis

As for the synthesis, a script was written. As always, the compiler first analyzes the VHDL entities, elaborates the top entity and starts the compilation using the provided clock frequency. The files produced are: a verilog netlist of the flattened design, the design constraints file (SDC) and the delays file (SDF). Moreover, some important reports are generated. The output of the elaborate command is redirected to a file so as to check that no latch has been synthesized, as only flip-flops are allowed. The compile log has also been examined in search of errors and warnings. In addition to that, reports on timing and area have been printed on two different files. The results are:

RISC-V version	Total Area [μm^2]	Slack (20 ns CLK period)
Version 2	15374	8.95
Version 3	15554	8.84 (rising edge)

By reading through the file, it appears that the most space is covered by sequential logic (i.e. memories and registers), as expected. Lastly, in order to emulate practical situations, realistic features have been set for the design. As a result, from the previous labs, we included in the script the commands that set a clock uncertainty, input and output delays and a load.
After the synthesis, the verilog netlist was simulated to verify the equivalence of the designs before and after the process.

0.7.2 Place & Route

The last step is the physical design. It was accomplished by following the same steps from the previous labs already described in detail in the Laboratory 1 report. These steps are: design import, floorplan structuring, power rings insertion, standard cell power routing, placement, post CTS optimization, filler placement, post routing optimization, parasitic extraction, timing analysis, design analysis and verification. For the second version of the RISC-V architecture, this part was performed using the *Innovus GUI*. For the third part instead, a script was written. The script contained the same commands used for the previous design which are normally logged by *Innovus* in a .cmd file. After the procedure, the final design has been again simulated to verify its correctness.

The obtained designs are presented in the following pictures, showing the post-route layouts of the two developed versions of the RISC-V processor:

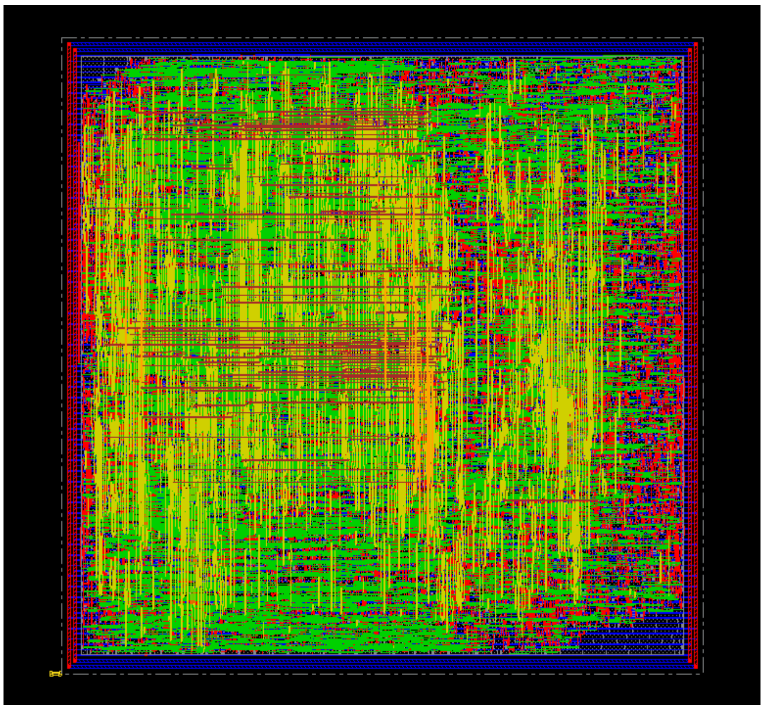


Figure 16: Screen capture of the placed and routed second version of the RISC-V processor.

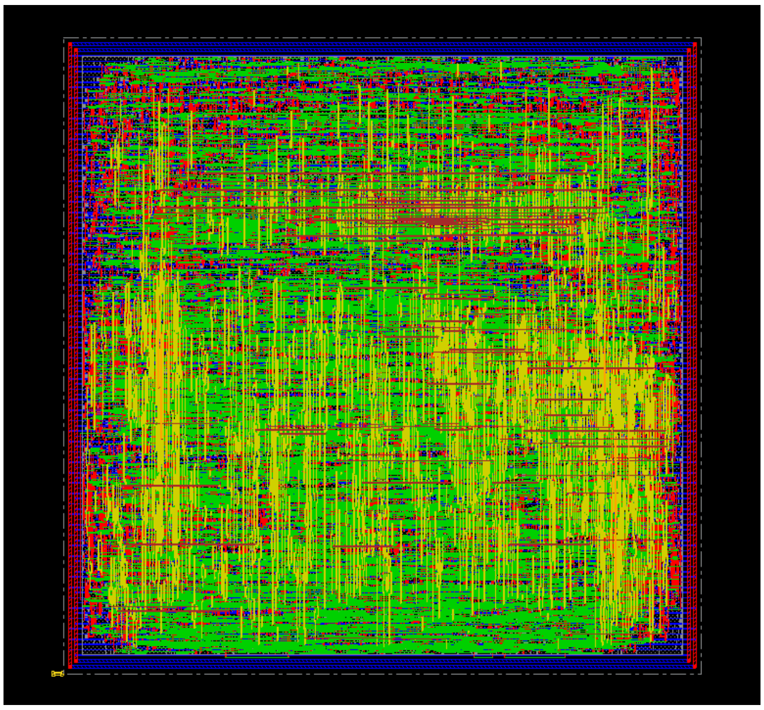


Figure 17: Screen capture of the placed and routed third version of the RISC-V processor.

The next tables show a quantitative overview on the differences between the two architectures.

	Second Version	Third Version
Leaf Cells	20137	20265
Nets	7564	7664
Standard Cells	6345	6482
Gates	18556	18780
Total StdCell Length	10.42 mm	10.55 mm
Total StdCell Area	14594 μm^2	14774 μm^2
Total Allocated Area	25624 μm^2	25940 μm^2
Design Density	57.79 %	57.77 %
Pin Density	29.64 %	29.73 %
Worst Slack	6.880 ns	6.887 ns

Filler Cells Drive Strength	Second Version	Third Version
FILLCELL_X1	8810	8684
FILLCELL_X2	0	0
FILLCELL_X4	2479	2540
FILLCELL_X8	2334	2358
FILLCELL_X16	134	185
FILLCELL_X32	35	16
Total	13792	13783

Table 2: Number of filler cells needed for each design

Second Version

	Wire Length
Layer M1	5436 μm
Layer M2	42584 μm
Layer M3	47319 μm
Layer M4	18026 μm
Layer M5	3228 μm
Layer M6	635 μm
Layer M7	0 μm
Layer M8	0 μm
Clock Nets	5354 μm
Total	117228 μm

	Number Of Vias
M1	29180
M2	19642
M3	2506
M4	205
M5	36
Total	51569

Table 3: Clock Nets length was estimated during eGR (early global routing)

Setup mode	all	reg2reg	default
WNS (ns)	6.881	8.684	6.881
TNS (ns)	0.000	0.000	0.000
Violating Paths	0	0	0
All Paths	3320	1611	3320

Table 4: WNS: Worst Negative Slack, TNS: Total Negative Slack

Hold mode	all	reg2reg	default
WNS (ns)	0.005	0.005	0.000
TNS (ns)	0.000	0.000	0.000
Violating Paths	0	0	0
All Paths	1611	1611	0

Table 5: WNS: Worst Negative Slack, TNS: Total Negative Slack

Third Version

	Wire Length
Layer M1	5398 μm
Layer M2	40587 μm
Layer M3	50109 μm
Layer M4	15246 μm
Layer M5	2205 μm
Layer M6	233 μm
Layer M7	0 μm
Layer M8	0 μm
Clock Nets	5248 μm
Total	113779 μm

	Number Of Vias
M1	29560
M2	19598
M3	2504
M4	172
M5	16
Total	51850

Table 6: Clock Nets length was estimated during eGR (early global routing)

Setup mode	all	reg2reg	default
WNS (ns)	6.888	8.680	6.888
TNS (ns)	0.000	0.000	0.000
Violating Paths	0	0	0
All Paths	3320	1611	3320

Table 7: WNS: Worst Negative Slack, TNS: Total Negative Slack

Hold mode	all	reg2reg	default
WNS (ns)	0.006	0.006	0.000
TNS (ns)	0.000	0.000	0.000
Violating Paths	0	0	0
All Paths	1611	1611	0

Table 8: WNS: Worst Negative Slack, TNS: Total Negative Slack

0.8 Conclusions

We can now briefly summarize the obtained results and compare them to have a view of the design general evolution. First we can consider the effective changes and improvements obtained during the simulations of the three different versions. To do this we will take as reference point the simulation instant in which the final result is written in the memory, considering that time as the end of the calculations. Using this assumption and working always with the same clock period of $10ns$, we can compare the different solutions by means of the execution time. In the following pictures the simulations of the three versions are reported:

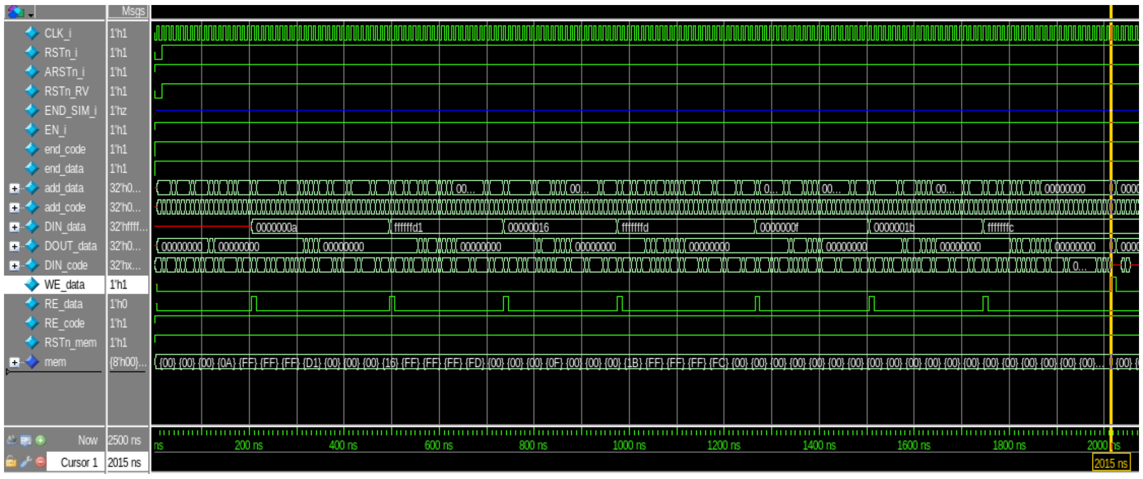


Figure 18: Simulation of the first version where all the *nop* bubbles are present in the assembly code.

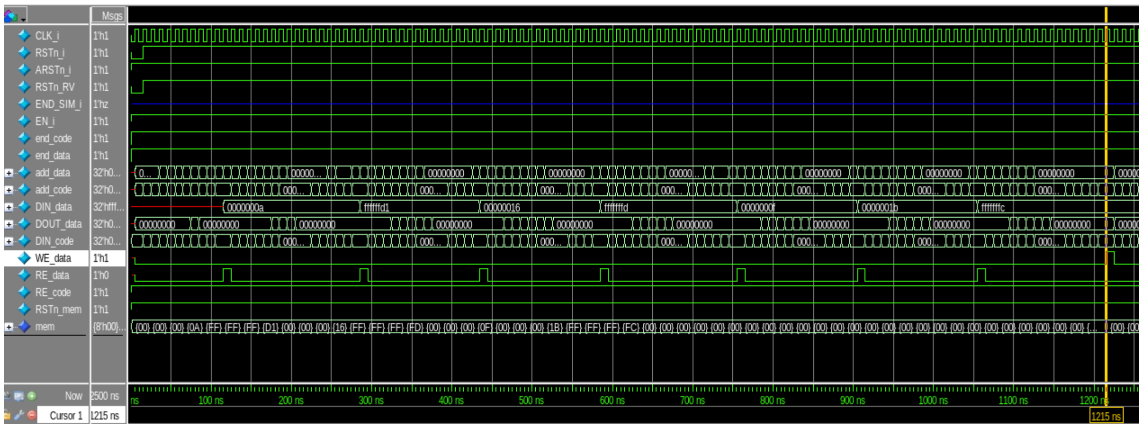


Figure 19: Simulation of the second version where no more *nop* bubbles are present in the assembly code.

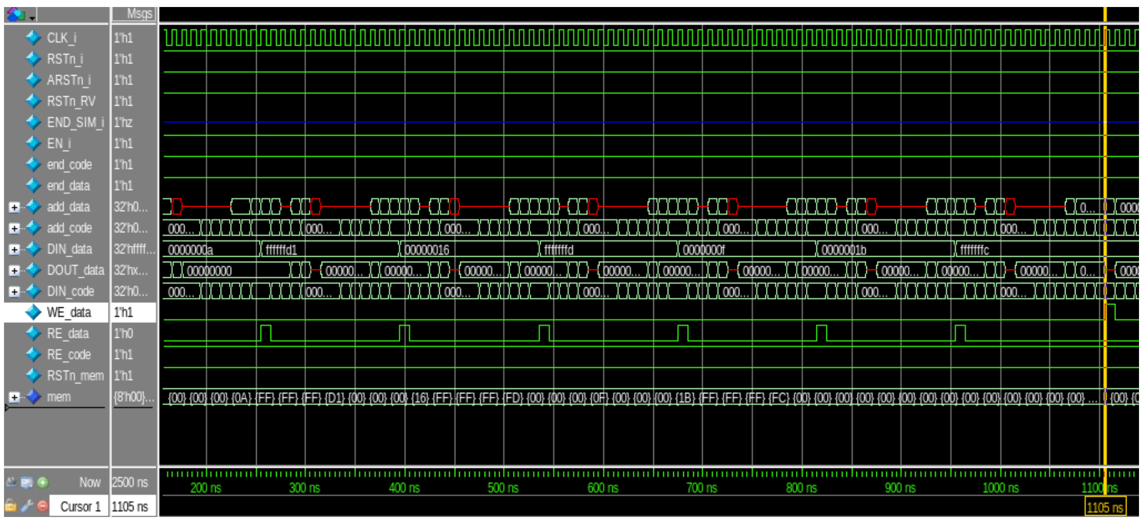


Figure 20: Simulation of the third version where the special unit for absolute value calculation is present.

Thanks to these simulations we have found the execution time for each implemented solution and in particular we have obtained:

Design version	Execution Time
1	$2015ns$
2	$1215ns$
3	$1105ns$

We can notice the improvements obtained by the different solutions and in particular the speed-up reached by the final version where we have almost halved the execution time. This is due to the fact that by using only one instruction to calculate the absolute value we can reduce by a lot the clock cycles needed to complete the execution. However, if we compare the simulation results with the reduction of the assembly code reported on table 1, we can see that the final results are lower than the expected ones. This is principally due to two factors:

- Even if the second and the third version of the design have both the Hazard and the Forwarding units not all the original *nop* bubbles are completely optimized. In fact, there are always some internal stalls that are needed to correct the behavior but cause a decrease in performance.
- In the table we consider only the number of instructions written in the assembly code and not the ones really executed. We have to consider that the code has one cycle and some conditional branches so the final number of executed instructions doesn't correspond to the one on the table but it will be higher.

We can say that the table gives us only an idea of the code reduction obtained by the different solutions but doesn't represent the true improvements obtained for the execution time: these can be obtained in a correct way only using simulations.
The results obtained after the synthesis and the place and route process have the same behavior of the ones previously reported. We obtained in both cases a reduction of the execution time, passing from the second to the third version that is very similar to the one described before. This allowed us to confirm the advantages obtained during the evolution of the design and to identify the last version as the best one in terms of performances.

In conclusion we have to say that in this laboratory we optimized the RISC-V architecture but we did not improve the given assembly code. To reach better performances we could also act on it and reorder some instructions to avoid pipeline stall. However, we decided to limit our work to the architecture itself so that all the obtained improvements are not dependent on the assembly code and are more general.