Politecnico di Torino
III Facoltà di Ingegneria

Exercises and Homeworks for the course Integrated Systems Architecture

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

Master degree in Electronic Engineering

Group 07

Campagnoli Rafael S260391, Fraval Adrien S276764, Galasso Luigi S267302

January 26, 2020

Contents

CHAPTER 1

Introduction

1.1 Project Description

The purpose of this laboratory resides in the portrayal of the project of a basic pipelined RISCV-Lite processor. After the VHDL description and a simulation through a specific assembler test code, a refinement from RTL (Register Transfer Level) down to synthesis and physical design was performed. The netlists obtained have been subsequently tested to confirm the correct behaviour of the processor. Then a Special Unit was implemented and inserted in the architecture to calculate automatically the absolute value, therefore a new instruction is added and the previous mentioned phases of test, synthesis and Place and Route were performed. In this sense, this report elucidates the description of the essential components needed to assemble the RISCV processor and the main choices made to accomplish the objectives. In addition some reports were generated for both the implementation to better compare the results.

1.2 Specification

1.2.1 RISCV Lite

The specification used for the basic RISCV configuration used is:

- **Pipeline**: The architecture is organized with a five staged pipeline data path and an "hardwired" control unit.
- **Instruction Set**: A set of the 12 basic RISCV-Lite instructions to be implemented.
- **Data Path**: A VHDL description at RT level of a data path capable of execute all of the instructions in the instruction set defined.
- **Control Unit**: A VHDL description at RT level of a control unit capable of organize and manage the correct behavior.
- **Synthesis**: A synthesis of both the control unit and the data path with the generation of the respective performance reports and post synthesis simulation.
- **Physical Design**: The placing and routing of the synthesized design and post placing simulation.

1.2.2 Instruction Set

The instructions set is the RISCV-Lite configuration one, which includes:

- arithmetic: **add, addi, auipc, lui**;
- branches: **beq**;
- loads: **lw**;
- shifts: **srai**;
- logical: **andi, xor**;
- compare: **slt**;
- jump and link: **jal**;
- stores: **sw**.

The instruction set can be divided in 6 different types of instructions that can be implemented: the R-type, the I-type, the S-type, the SB-type, the U-type and the UJ-type instructions. Depending on which is the type of instruction, different groups of bits set the 32 bits word instruction. The possible different fields are the **OPCODE**, the **FUNC3**, the **FUNC7**, the **Register Destination Rd**, the **Operands Rs1, Rs2** and the **Immediate**. In Figure 1.1 the different instructions types formats are reported.

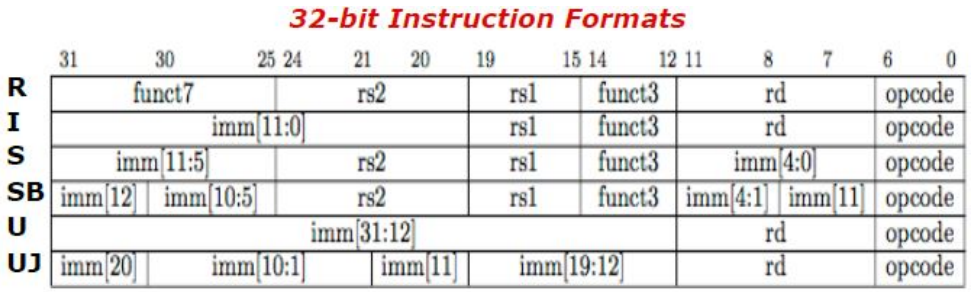

Figure 1.1: Different types of instructions.

CHAPTER 2

Functional Schema

The purpose of this chapter is to describe the schematics of the main blocks of the system and to explain its main way of working and interconnections. In figure 2.1 the functional schema of the all RISCV-Lite is illustrated. Two main blocks represent the core of the microprocessor. In particular there is a **Data Path**, colored black, consisting of 5 stages (FETCH, DECODE, EXECUTION, MEMORY and WRITE BACK) pipelined and a **Control Unit** interconnected as shown in the illustration. The Control Unit is connected with the FETCH UNIT output, i.e. the instruction to be decoded, and the other four stages. Each Data Path stage is then connected with the next one, except for the DECODE UNIT which is connected to the FETCH STAGE to perform JUMP operations if required and the WRITE BACK UNIT which is connected to the Register File of the DECODE UNIT to perform write operations. Additional connections are also exploited for Forwarding operations between DECODE, EXECUTE and WRITE-BACK stages. In the schema are also present two external memory: an **Instruction Memory** and a **Data Memory**.

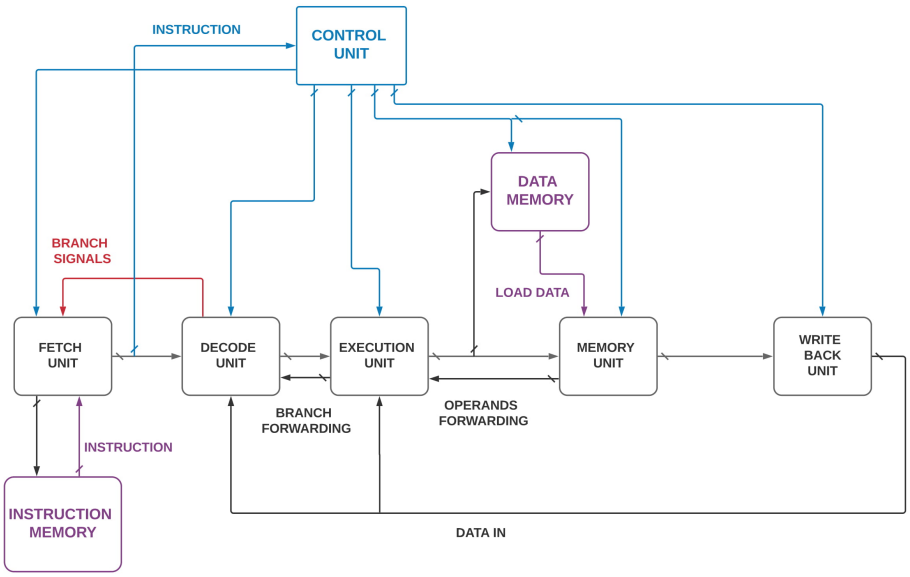

Figure 2.1: Schematic of the complete RISCV.

2.1 Data Path

The Data Path unit is composed by several different blocks. In order to simplify the discussion it is pointed out here that each stage contains a 64-bits pipeline register used to increase the throughput of the execution. By doing that the latency of the whole data path increases and the clock frequency can be increased, limited only by the stage with the highest latency plus the set-up time of the registers. In the following, each block will be examined.

2.1.1 Fetch Unit

The purpose of the fetch unit is to "insert" in the datapath an instruction from the instruction memory. The instruction memory is basically a LUT (Look-Up Table), which contains the programmed code instruction by instruction aligned in its addressable space. The instruction memory is addressed via the PC (Program Counter). Every clock-cycle, the PC is incremented by 4 and fetches the next instruction, which is then sent to the decode stage. The schema is reported in Figure 2.2. It can be noticed from the figure the signals *BRANCH FROM DECODE*, which represents that a branch or a jump must be taken, and the *BRANCH PROGRAM COUNTER*, which represents the program address that should be fetched in the next clock cycle. The *BRANCH FROM DECODE* is the selection bit of the multiplexer whose output is connected to the program counter register and also enters in a OR port with the reset signal to the reset port of the Instruction register. This second configuration is due to the need to implement a pipeline flush in the case of a branch is taken. A signal EN1 is used to enable the registers. The signals that pass to the next stage through registers are the instruction and the PC.

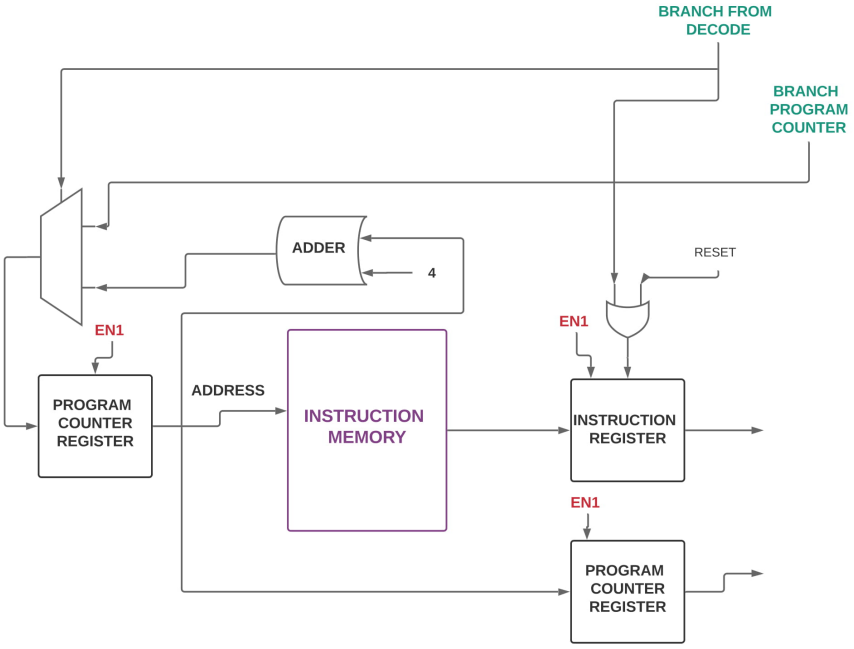

Figure 2.2: Schematic of the Fetch Unit.

2.1.2 Decode Unit

The DECODE UNIT can be observed in the Figure 2.3. It is assembled with the register file, the branch unit, some logic gates, the immediate sign extension unit, the write-back address multiplexer and Branch Forwarding Unit.

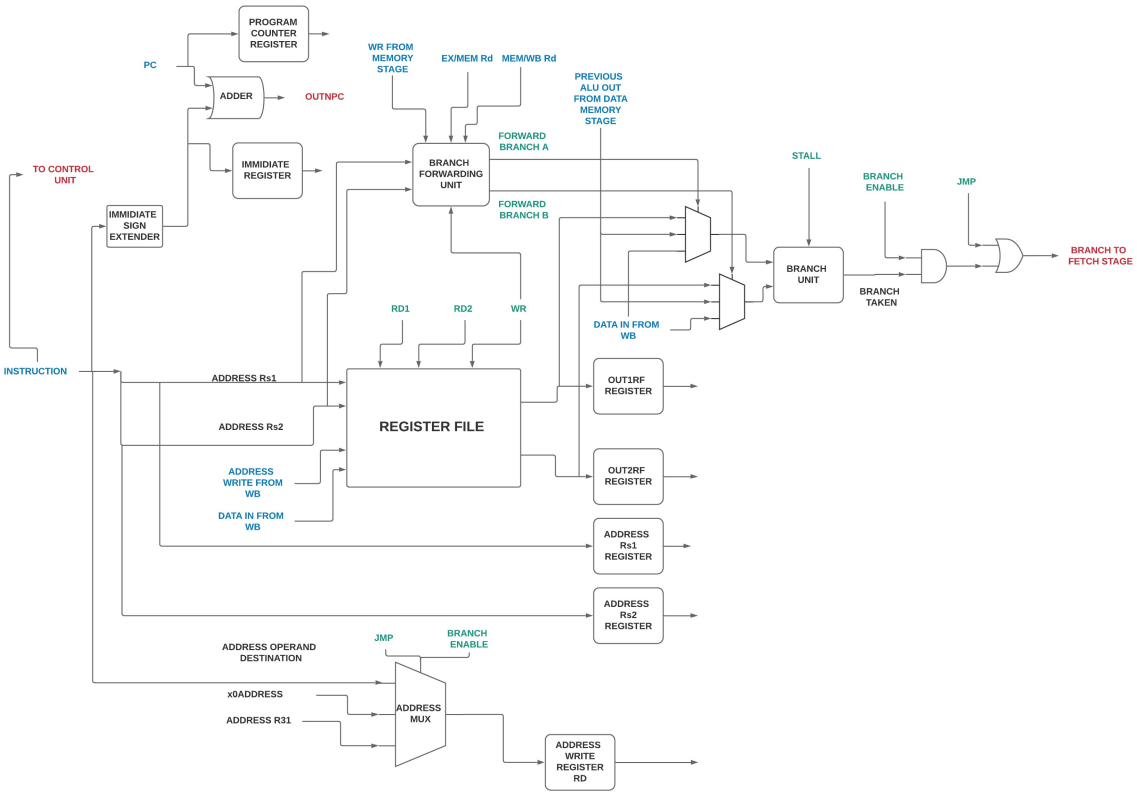

Figure 2.3: Schematic of the Decode Unit.

The Register File(RF) is a 32x64-bit general purpose register. It can be read through two ports concurrently and written through one port. Instruction's operands addresses are used to address the Register File outputs. The read operation is asynchronous, while the write operation is only concluded after one clock cycle; however, if the address to be written is the same of an operand address to be read, the Data in signal is selected for the respective output. The Control signals delivered by the Control Unit to the Register File are in this stage RD1, RD2 and WR. The Branch unit detects if the Branch equal condition is verified or not; if the condition is verified and the BRANCH ENABLE signal from Control Unit is high, through a logic of an AND port and an OR port, the BRANCH TO FETCH signal is sent to the Fetch Unit. Otherwise, in the case of a Jump instruction, the JMP signal from Control Unit is high and the Jump operation has to be done unconditionally. The Stall signal from the Control Unit is raised when one of the operands of the Branch instruction is modified by the direct previous instruction and the Forwarding bypass is not effective, once the result is not yet calculated. That signal acts as an enable signal for the Branch Unit. Then, in order to deliver the correct Program Counter to the Fetch Unit it is implemented an addition of the Program Counter from Fetch stage and the correct 64-bit sign extended immediate, according to the type of instruction, corresponding to the Jump/Branch displacement. As mentioned before, a Branch Forwarding Unit was placed in order to bypass the needed operands in the case of a Data Hazard. It compares the

addresses of EX/MEM register Destination and MEM/WB register Destination, in case of a Write back has to be performed, with the operands of the branch instruction. This component selects through the two signals FORWARD BRANCH A/B to two forwarding multiplexers the BRANCH UNIT operands. As it is shown, it can be observed the presence of an address multiplexer (3 to 1 multiplexer) that is configured by BRANCH ENABLE and JUMP signal from the Control Unit which select the proper address for register destination. If the instruction has its own register destination it is selected otherwise if the JMP signal is high the constant address of register 31 is selected. In other words, in JAL instruction case the address of the operand destination is selected as R31. If, instead a NOP has to be performed the address x0 is chosen as register destination. Seven pipeline registers have been finally inserted. They store respectively: Program Counter, Immediate, OUT1RF (first Register File Out),OUT2RF (second Register File Out), Address Write (Operand destination address) and Operands Addresses Rs1 and Rs2. All the registers are enabled by the EN2 signal from CONTROL UNIT.

2.1.3 Execution Unit

The execute unit is the main block of the processor. It is where the instruction is executed using the right data and can be observed in the Figure 2.4. It was assembled with the ALU(Arithmetic Logic Unit), five multiplexers, a Forwarding Unit and a set of pipeline registers.

The two inputs of the ALU are selected by the Control Unit control signals MUX1SEL and MUX2SEL and the FORWARD A and FORWARD B from the the FORWARDING UNIT through the multiplexers. The Forwarding Unit was placed in order to bypass the needed operands in the case of a Data Hazard. It compares the addresses of EX/MEM register Destination and MEM/WB register Destination, in case of a Write back has to be performed (WR control signal high in these two stages), with the operands from the DECODE UNIT. This component selects through the two signals FORWARD A/B two forwarding multiplexers the final ALU operands. These 3 to 1 forwarding multiplexer are therefore connected in their input to the Write-Back Out and the Previous ALU out from the Memory stage. The three 2 to 1 Multiplexers are instead used to select the proper operands coming from the decode stage OUT1RF, OUT2RF, PC(Current Program Counter), Immediate and the constant 4 used to perform the JAL instruction and save the Next Program Counter, with respect to the JAL instruction Program Counter , in the Register File location 31. The ALU is able to perform the required RISCV-Lite 64-bits operations. It is configured by the 4-bits ALUCODE signal delivered by the Control Unit. In particular the possible operations are:

- arithmetic: **add**;
- shifts: **srai**;
- logical: **and, xor**;
- compare: **slt**.

The Outputs of this stage are stored in three Pipeline registers corresponding to the ALU's output, the second Register File's output and the address of destination. All the registers are enabled by the EN3 signal from CONTROL UNIT.

2.1.4 Memory Unit

The MEMORY UNIT is composed of three pipeline registers and an external DATA MEMORY.

As it can be observed in the Figure 2.5, there are three inputs. Two of them, i.e. the ALU OUT and the OUT2RF, are used respectively as data input(to be written) and address, for load/store operations of the DATA RAM which is a read and write memory. Memory operations are managed by

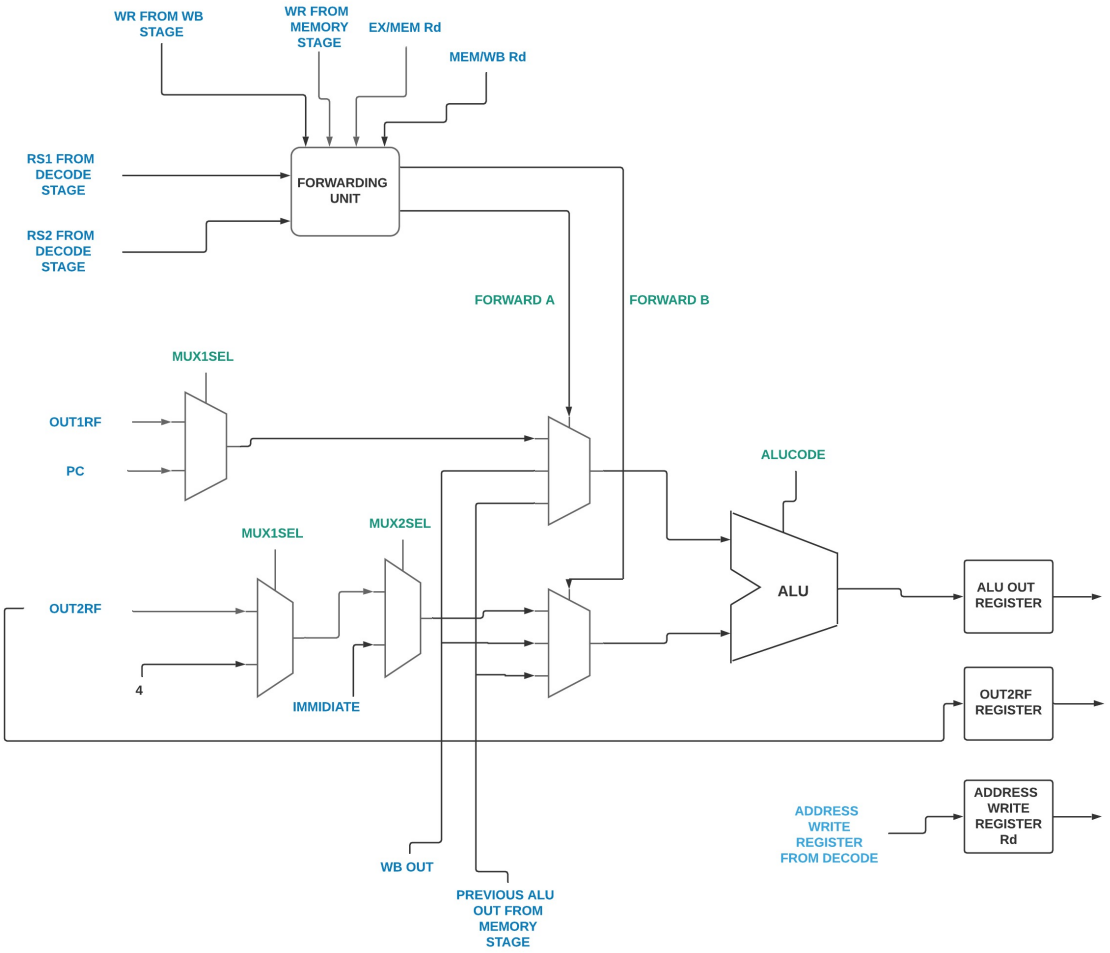

Figure 2.4: Schematic of the Execution Unit.

two control signals from the Control Unit: DATA MEMORY ENABLE (which activates the memory) and READ NOT WRITE (which configures the operating mode of the RAM). The output of Memory is stored in a LMD (Loaded Memory Data) register, whereas the ALU OUT register stores the ALU output and the ADDRESS WRITE register delays, of one clock cycle, the writing operation in Register File address pointed by the ADDRESS WRITE FROM EXECUTE. All the registers are enabled by the EN3 signal from CONTROL UNIT.

2.1.5 Writeback Unit

The WRITE-BACK unit is composed of a simple multiplexer 2 to 1.

As it can be noticed in the Figure 2.6 the two inputs of the write-back multiplexer (the ALU OUT and the LOAD DATA from MEMORY STAGE) can be selected as data to be written in Register File in DECODE UNIT through a SELECT WB signal from the CONTROL UNIT.

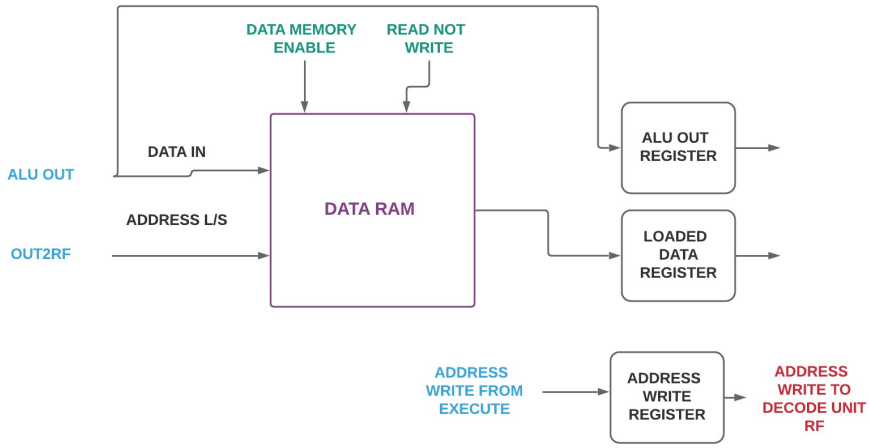

Figure 2.5: Schematic of the Memory Unit.

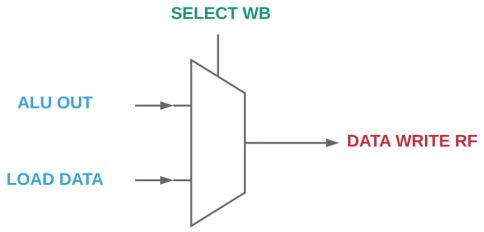

Figure 2.6: Schematic of the Write Back Unit.

2.2 Control Unit

The implementation chosen for the CU (CONTROL UNIT) is the HARDWIRED one. As it is illustrated in Figure 2.7, the INSTRUCTION taken as input is decoded by its two major parts, i.e. the OPCODE and the FUNCTION3, which determine through a LUT(Look Up Table) a proper Control Word and the correspondent ALUCODE. In that Control Unit is also present an Hazard Detection Unit able to prevent the Load-Use Hazard. It requires 5 input: the ID/EX address of register Destination Rd to be compared with the IF/ID addresses of registers Operands Rs1 and Rs2 in case a memory read has to be performed (ID/EX MEMORY READ=1) producing the BUBBLE signal and in case a Branch is decode (ID/EX BRANCH ENABLE=1) raising the STALL signal. Then if BUBBLE OR STALL is high the FETCH UNIT components are flushed(ENABLE1=0). In addition in case the STALL is high the Branch Unit is enabled. The BUBBLE signal is required to insert a bubble (setting all the Control Word bits to 0) in the pipeline and to do that the proper CONTROL WORD and ALUCODE is selected through two multiplexers. Some registers are then

exploited to store them and allow the pipeline to work in the correct way. The 13-bits Control Word is then delivered to the different stages, delayed by the right number of CLOCK periods, for the correspondent Control Word bits. For what concerns the 4-bits ALUCODE, it is first calculated during the Decode Phase, by the CU and then delayed by one CLOCK cycle before properly reach the ALU in the EXECUTION UNIT.

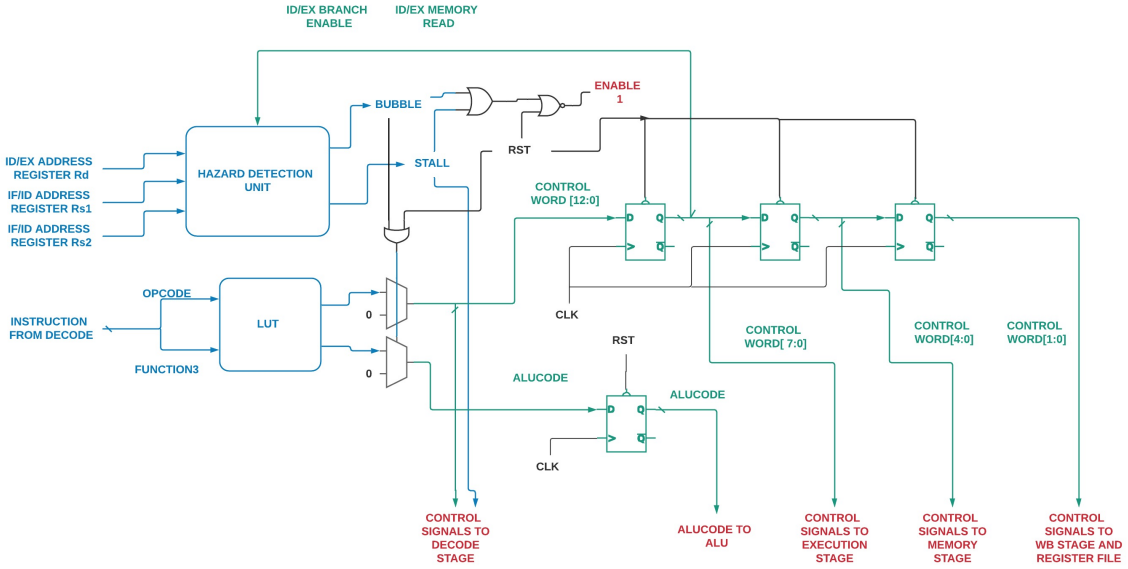

Figure 2.7: Schematic of the Control Unit.

Control Word

In the instruction decode stage, the instruction contained in the instruction register is sent to the hardwired control unit, which then translates it in a control word. The control word used contains the 14 bits needed for the correct functionality of the datapath and the 4 bits ALUCODE required for the ALU correct behavior. The control word can be described, from MSB to LSB, as:

- **EN1**: Enable signal for the registers in the fetch stage;
- **EN2**: Enable signal for the registers in the decode stage;
- **RD1**: Enable signal for the read port 1 of the RF;
- **RD2**: Enable signal for the read port 2 of the RF;
- **JMP**: Unconditional Jump signal;
- **BRANCHenable**: Enable signal for the output of the branch unit;
- **EN3**: Enable signal for the registers in the execute stage;
- **MUX1SEL**: selection bit used for the selection of the ALU inputs;
- **MUX2SEL**: selection bit used for the selection of the ALU inputs;
- **ALUCODE**: 4-bit code to select the operation that must be performed by the ALU;

- **EN4**: Enable signal for the registers in the memory stage;
- **MemoryEnable**: Enable signal for the operation of the memory;
- **ReadNotWrite**: Signal to the define which operation should be executed by the memory;
- **SelWB**: Selection signal for the multiplexer that selects the output of the ALU or the Output of the memory;
- **WR**: Enable signal to write-back into the register file.

2.3 Memories

The memories present in the project are the Instruction Memory and the Data Memory. It should be pointed out here that they are outside the real Data Path and for this reason they were put inside the Test Bench and therefore they were not synthesized.

Instruction Memory

The instruction memory is a ROM (Read Only Memory). Its purpose is to store the code of the program in the form of bit words, which are later decoded in the control unit in the decode stage. For this project it was realized a component which basically takes an assembly program (test.asm.mem) previously converted to hexadecimal and aligns it in a LUT. The memory is then addressed by an index of 4, according to the RADIX-4 allignement used by the compiler's assembly code.

Data Memory

The data memory is a external memory used for load and store. The one that is described in VHDL is very similar to the register file, for simulation purposes. It's functionality can be described as a memory where one can read or write some data at a given address. It is controlled by an enable signal and a read-not-write signal that selects the type of operation to be done. Both signals come from the control unit. For simulation purposes the offset used to fill the memory used is the one imposed by the compiler and in addition it was realized word addressable, i.e. it is possible to read one word at a time (4 bytes). To instantiate the correct initial stored values a file data.mem was used.

2.4 Special Unit

In the following stage of the laboratory, an optimization for the system was proposed. A Special Unit was inserted, able to perform the absolute value of a given signed operand. The correspondent instruction to be used added is **abs**. The insertion of a special unit for performing the absolute operation could reduce the code size and therefore optimize the execution time of the code. In particular, as it is possible to observe by the assembly code reported in the Appendix the **abs** instruction replaces four instructions (commented in the reported assembly code with the symbol #) with an advantage in terms of code length and execution duration. The Special Unit was described and inserted as a component of the ALU and a proper ALUCODE was also added. This Special Unit receives as an input the first ALU operand and perform a sign check of the MSB (Sign Bit):

- if the MSB is 0 the operand is positive and it is directly output;
- if the MSB is 1 the operand is negative and the operand is first complemented, i.e. a bitwise XOR with 1 is performed, and then a 1 is added to generate the absolute value.

The new RISCV-Lite system was then simulated, synthesized, placed and routed using the same procedures as the RISCV-Lite version.

CHAPTER 3

Implementation

After the correct assembly of the complete processor system, a simulation was performed using Modelsim to verify the correct behaviour of the described RISCV processors. After the simulation phase, a synthesis using the software Synopsys Design Vision followed by a physical design using the software Innovus were performed. After both the Synthesis and the Place&Route further simulations were performed in order to prove the correctness of these steps. The purpose of this chapter is to describe the different phases performed for both the implementations (with and without the special unit) and comment the results obtained.

3.1 Simulation

The simulation performed was the execution of the given assembly test program (minv-rv.s) and the verification of the data memory at the end of the program. The given test program functionality is evaluate a vector of integers and writing in the memory the one with the lowest absolute value.

To verify if the processor could correctly execute the program, a testbench containing both the data memory with the program input data and the instruction memory with the machine code of the program was created. To translate the assembly program into hexadecimal machine code, the RARS-RISC-V Assembler and Runtime Simulator was used. The set offset for the memories in the RARS software were 0x00000 for the instruction memory and 0x02000 for the data memory, so the testbench was configured accordingly.

If the processor could execute the program, choose among the numbers the lowest absolute value and put it in the data memory in the correct address, the processor would be considered valid.

In the case of the Special RISCV, the test program was reduced in a few lines as the new special instruction -abs- could be inserted. The test codes for the simulations can be found in the Appendix A. In particular it's possible to observe that the special-unit used code provides the usage of just one instruction **abs** in place of 4 instructions in the code commented. This reduction of the code length allows to have a smaller execution time in the case of the **abs** insertion. Anyway both the implementation work as expected.

3.2 Synthesis

3.2.1 Description

In the Synthesis script the entire systems, apart from the memories, were analyzed from bottom to top level. An elaboration and a compilation with no map effort were performed and a timing report

was generated using **Synopsys Design Vision**. The target library used was nangate45. In order to find the maximum frequency of the system, a clock constraint of 0 was set in order to verify what was the lowest clock period that the design compiler could achieve. After that, the clock constraint was set to 4 times the previously achieved minimum period and the design was compiled again. Timing and area reports of the final design were generated. For later use for verification, the script also saved a verilog file for the netlist, a .sdf file for simulation and a .sdc file for the place and routing. The verilog netlists of the two implementation obtained were then, as anticipated, tested again confirming the same behavior of the functional simulations.

3.2.2 Results and Comparison

The critical path delays achieved setting the clock period to 0 for the two implementations are reported in Table 3.1.

	Critical Path Delay
RISCV-Lite	1.42 ns
RISCV-Special Unit	1.42 ns

Table 3.1: Minimum critical path delays achieved in the RISCV synthesis.

Both the implementation obtained during synthesis phase a maximum critical path delay of 1.42 ns. As the timing reports already considers the set-up time, it can be calculated the maximum clock frequency, which should be the inverse of the "worst-path time" of the processor. In this project, it was possible to achieve a maximum clock frequency of approximately 700 MHz. The subsequent step of the synthesis provides the usage of four times the critical path delay as a clock period. Therefore a new synthesis was run in both the cases and the results of the power and area reports of the RISCV processor can be observed by the Table 3.2 and 3.3.

	Internal Power	Switching Power	Leakage Power	Total Power
RISCV-Lite	1.1900e+03 μW	236.8709 μW	6.6848e+05 nW	2.0953e+03 μW
RISCV-Special Unit	1.1739e+03 μW	208.0577 μW	6.7199e+05 nW	2.0539e+03 μW

Table 3.2: Power results in the final synthesis of the RISCV.

	Total Cell Area
RISCV-Lite	32299.848256 μm^2
RISCV-Special Unit	32592.448252 μm^2

Table 3.3: Area results in the final synthesis of the RISCV.

Comparing the results obtained it is possible to assert that the total Power consumption estimation is roughly the same for the two implementation. This not very relevant variation between them is due to the Internal Power and the Switching Power slightly smaller in the case of the RISCV-Lite with the internal Special Unit. For what concern the Area it can be said that the the RISCV-Lite with Special Unit occupies just 300 μm^2 more then the base Lite implementation.

3.3 Physical Design

For the physical design, the standard procedure was followed. The verilog file generated in the synthesis part, i.e. the one synthesized and tested at the maximum frequency divided by 4, was used

for place and route using the software **Innovus**, through an optimization for routing. The results of the placing and routing of the two implementations can be observed in the Figure 3.1 and 3.2 showing the screen dumps generated. The post Routing analysis results are reported in the following Table 3.4.

	Gate Count	Area
RISCV-Lite	38821	16584 μm^2
RISCV-Special Unit	39365	16887 μm^2

Table 3.4: Gate Count and Area of the RISC-Lite obtained in the physical design.

As previously done after the synthesis, in this case too, a further simulation post Place and Route was performed in order to confirm the correct behaviour of the systems. A .vcd file was also generated in order to estimate the power consumption on the routed system. The power analysis results can be observed in Table 3.5.

	Internal Power	Switching Power	Leakage Power	Total Power
RISCV-Lite	4.37706634 mW	1.85955997 mW	0.59952473 mW	6.83615104 mW
RISCV-Special Unit	4.31570000 mW	1.83187757 mW	0.60763687 mW	6.75521444 mW

Table 3.5: Power results of the RISC-Lite obtained in the physical design.

As observed in the post-synthesis analysis the simple RISCV-Lite shows a smaller Area occupation and a bigger power consumption with respect to the RISCV-Lite with Special Unit. This can be traced back to a longer test in terms of code length. On the other hand the Area is slightly different due to the insertion of the Special Unit.

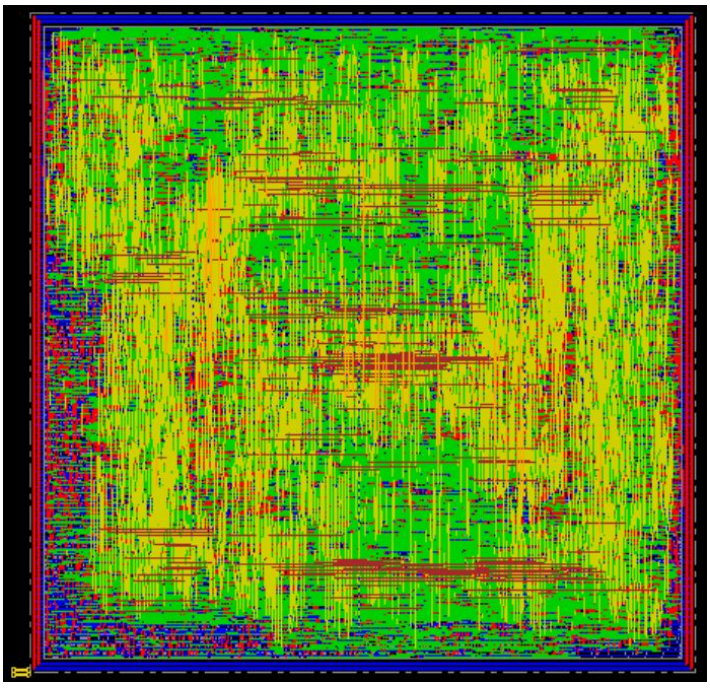

Figure 3.1: Screen Dump of the physical implementation of the RISCV-Lite

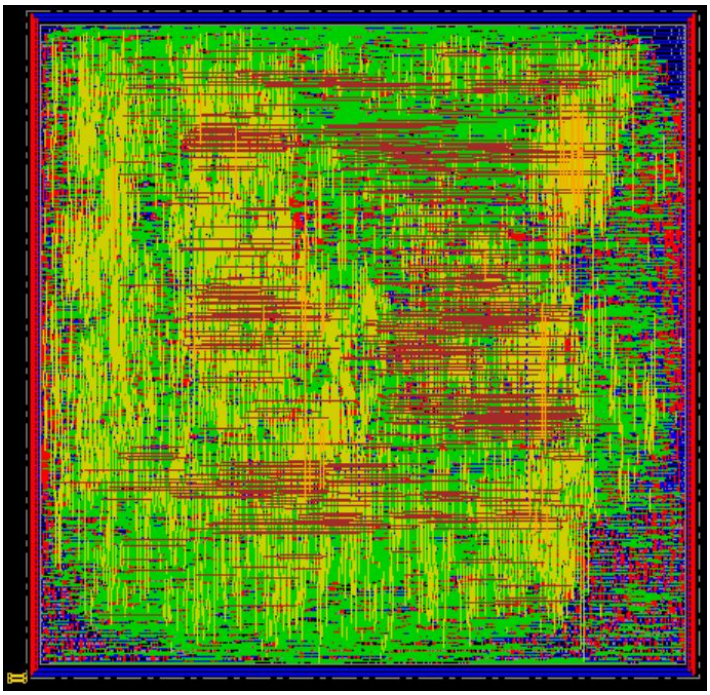

Figure 3.2: Screen Dump of the physical implementation of the RISCV-Lite with Special Unit

APPENDIX A

Assembly code

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

        .text
        .align  2
        .globl  __start
__start:
        li x16,7            # put 7 in x16
        la x4,v             # put in x4 the address of v
        la x5,m             # put in x5 the address of m
        li x13,0x3fffffff # init x13 with max pos
loop:
        beq x16,x0,done     # check all elements have been tested
        lw x8,0(x4)         # load new element in x8
        #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)
        abs x10,x8,x0
```

```
        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
```