Politecnico di Torino
III Facoltà di Ingegneria

Integrated systems architecture

Laboratory 3

Laurea Magistrale in Ingegneria Elettronica
Orientamento: Sistemi Elettronici

Group n. 9

Authors:

Favero Simone S270686
Micelli Federico S270456
Spanna Francesca S278040

Repository Github: github.com/federico-micelli/ISA_2020_Group_09_Lab_3

Index

RISC-V lite processor

1.1 Introduction

The purpose of this laboratory is to design the architecture of a RISC-V processor. In particular, a lite version that implements the following subset of the RV32I instruction set has been realized.

- **ADD**: addition between the content of two internal registers
- **ADDI**: addition between a value stored in an internal register and the immediate field
- **AUIPC**: addition between the current PC content and an offset represented by the immediate field
- **LUI**: a numeric value generated by the immediate field is stored in one of the internal registers
- **BEQ**: equality comparison between the content of two internal registers: if the condition is verified, then a branch is taken. The target address is generated in a relative way, through an addition between the current PC value and the immediate field
- **LW**: the content of a word located in the data memory is loaded into an internal register. The target address is computed through the immediate value
- **SRAI**: a sign-extended right shift operation is applied on the content of a given register. The amount of shift positions is stated by the immediate field
- **ANDI**: logical AND operation between a given register and the immediate field
- **XOR**: bitwise XOR operation between the content of two registers
- **SLT**: numeric comparison between values contained in two internal registers. The logic result of the operation is then stored in the register file
- **JAL**: unconditional jump to a target address generated through the immediate field. The next PC value, before performing the jump, is stored in the register file
- **SW**: the content of an internal register is stored into a proper data memory location. The target address is evaluated through the immediate field

As a typical RISC-V processor, the designed register file contains 32 locations where data are expressed with a parallelism of 32 bits.

Since the discussed processor relies on a Harvard architecture, instructions and data memories are separated entities.
All the memories are meant to be byte addressed. Therefore, taking into account the internal 32 bits parallelism, two adjacent locations are separated by 4 bytes indexing distance.
It is important to highlight that the implemented architecture is not supposed to include instructions and data memories, since these ones are requested to be external entities defined in the testbench scenario.

In order to improve processor performances, five pipeline stages have been introduced to separate the fundamental processing steps of an instruction.

- **IF**: instruction fetch
- **ID**: instruction decode
- **EX**: execution
- **MEM**: data memory access
- **WB**: write back

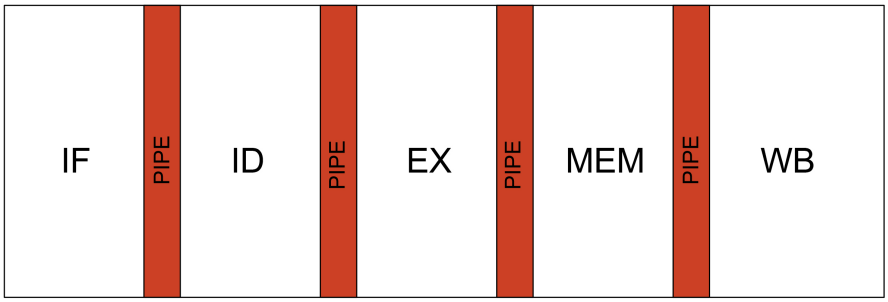

Figure 1.1: Pipeline stages

The stages listed above will be discussed more deeply in the following sections.
The correct synchronicity for operations execution is guaranteed by delaying in a proper way all the internal data and control signals.

The processor architecture is able to manage several types of hazards, which are introduced by the pipeline organization.
Structural hazards are faced by means of a flexible register file, while data hazards are managed with a forwarding mechanism.
Control hazards, that represent the remaining critical conditions, are managed through the insertion of NOP instructions. Since the instruction set does not contain a NOP, it is implemented as a special immediate addition between null values.

addi x0, x0, 0

1.2 Instruction fetch

This stage aims to fetch the proper instruction from the instruction memory and correctly update the value of the PC register. The block scheme is reported in the following picture.

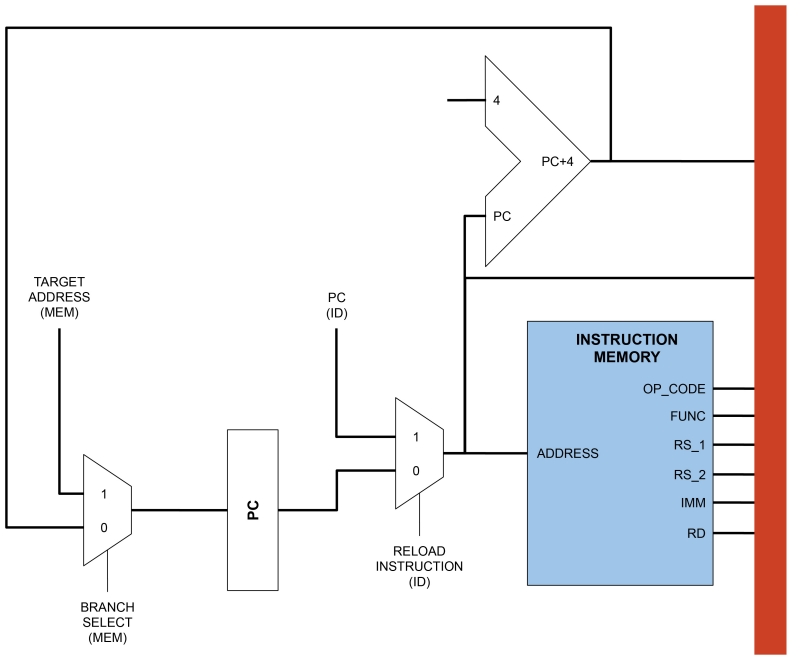

Figure 1.2: Instruction fetch stage block scheme

- **PC**: it is a 32 bits wide register that contains the indexing of the current instruction to be fetched.
- **MUX_JUMP**: it allows to select the next content for the program counter, which can be the target address of a jump instruction, evaluated in the MEM stage, or the previous value incremented by 4, as explained in the introduction.
- **MUX_RELOAD_INSTRUCTION**: it is exploited during the management of hazards conditions. Its purpose is to re-access the instruction memory with the same address, in order to guarantee the correct flow of instructions.
- **ADD4**: it calculates the next instruction address, adding 4 to the actual content of the PC.

1.3 Instruction decode

This stage is supposed to accomplish several tasks: decoding the actual instruction, managing the access to the register file, providing the correct immediate value.
Moreover, its purpose is also to check for possible control hazards and, if needed, overcome them.
The scheme is reported below.

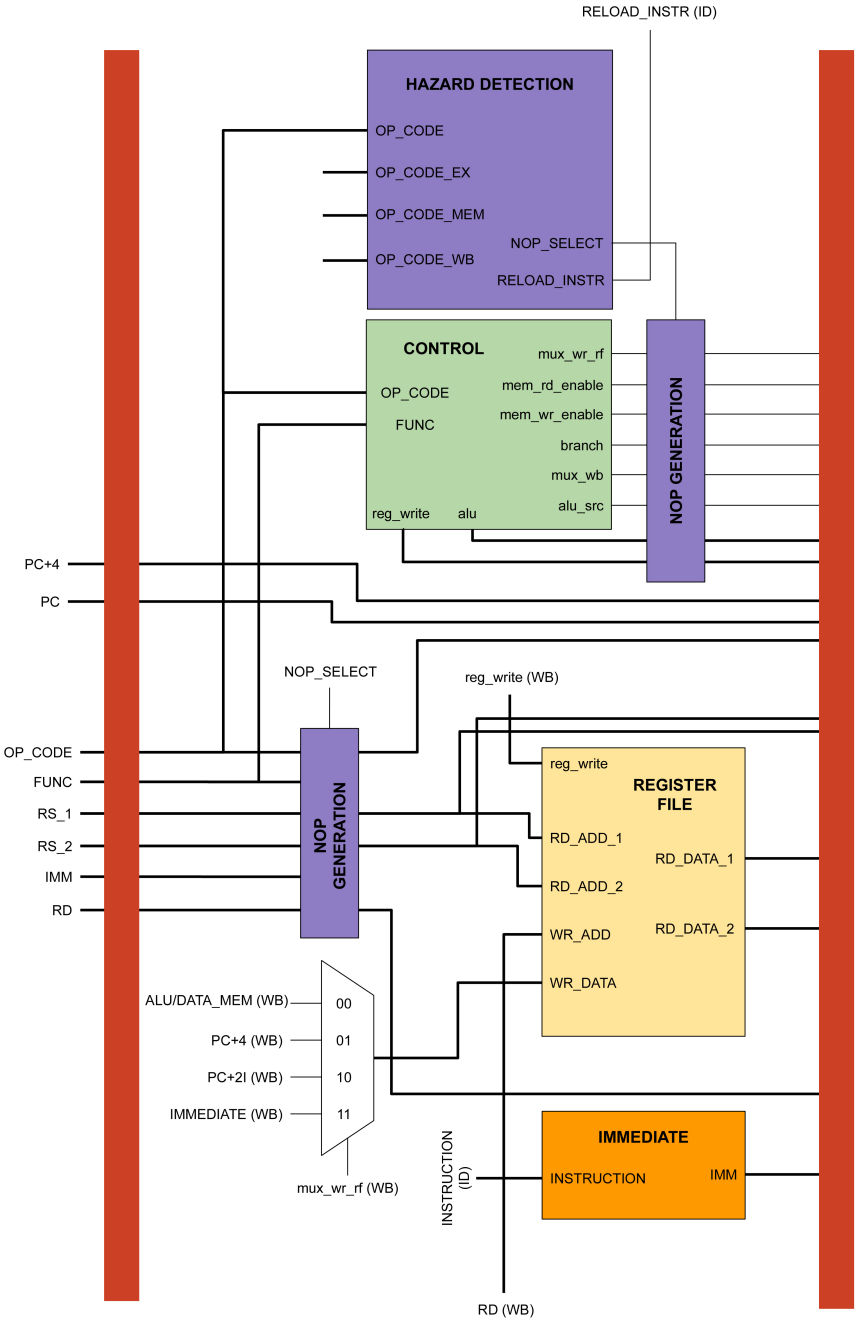

Figure 1.3: Instruction decode stage block scheme

- **CONTROL_UNIT**: starting from the OPCODE and the FUNC fields of the instruction, it generates the proper control signals for the operations to be executed in the next steps.
- **HAZARD_DETECTION**: this unit checks the presence of hazards, in particular two important conditions are detected. The first one is related to the detection of jump instructions, that are possible sources of control hazards: when a BEQ or JAL are recognized, a proper number of NOP is introduced until the jump condition and target address are well known. The second important condition is related to a data dependency between a LW and a generic consecutive instruction; since this scenario can't be managed directly by the forwarding unit, a NOP is inserted.
- **NOP_GENERATION**: it is enabled by a control signal, generated by the Hazard detection unit, if a NOP is required. This unit acts as a filter on both the instruction fields and the control signals, changing them into a NOP if the latter is requested; otherwise the original fields are propagated.
- **REGISTER_FILE**: it is a synchronous multiport memory which allows to simultaneously perform, on the falling edge of the clock signal, two read and a write operations. This property is needed to face structural hazards. Indeed, in a read-after-write condition, data is first written and then bypassed to the output port. All the read addresses are directly taken from the ID stage. On the other hand, fields related to the write operation, which are write address and write data signals, have to be delayed since they belong to the WB stage.
- **MUX_WR_RF**: the internal register file can receive data from different sources: ALU result/memory data, immediate, jump target address and next sequential instruction address.
- **IMMEDIATE_GENERATION**: this unit is intended to manage the generation of the immediate value, starting from the proper field in the instruction. The latter is correctly ordered, according to the instruction format to be executed, and extended on 32 bits.

1.4 Execution

This stage is dedicated to the execution of logic and arithmetic operations, both on data and addresses. Moreover, data hazards are detected and solved by the proposed forwarding mechanism.

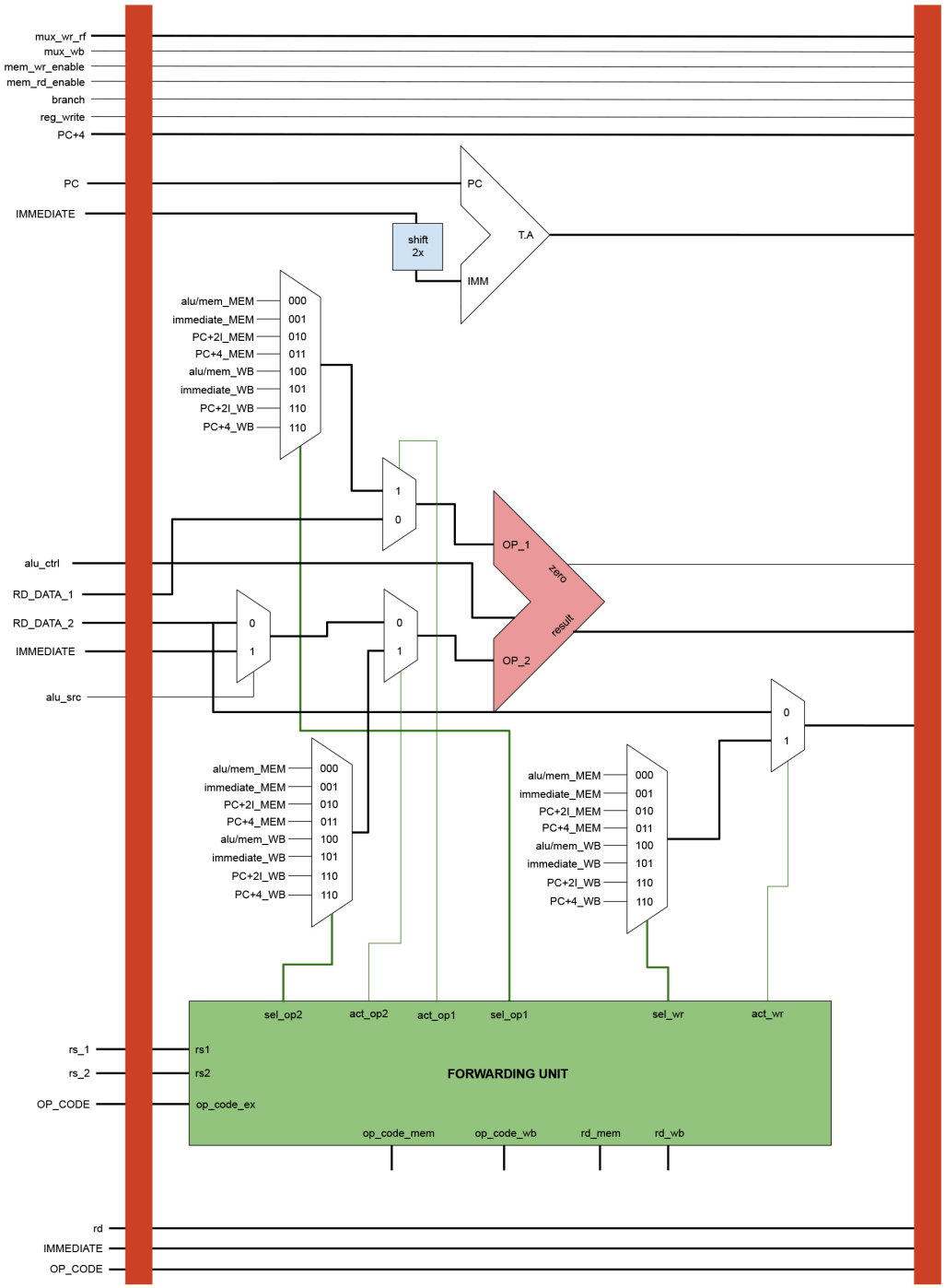

Figure 1.4: Execution stage block scheme

- **ADD_OFFSET**: this adder generates the target address, through a signed addition. The sum is performed between the PC value of the executed instruction and the offset generated by the Immediate Unit.
- **ALU_SRC_MUX**: since the second operand of the ALU can be selected between the second output of the register file and the generated immediate value, this multiplexer manages the selection.
- **MUX_FORWARD_ACTIVATE_OP1/OP2/WR**: the operands of the ALU and the data to be written in memory can be selected between the output of the register file and the bypassed values from the Forwarding Unit.
- **FORWARD_MUX_OP1/OP2/WR**: these multiplexers allow to select all the values that can be provided by the Forwarding Unit, both from MEM and WB stages. The choices are the ALU result/memory data, immediate, jump target address and next sequential instruction address.
- **ALU_UNIT**: it applies a specific operation between the two input operands according to the ALU control signal. The managed operations are a consequence of the specifications given by the instruction set. Furthermore it properly manages the BEQ instruction with a zero flag: this output status signal is asserted when the two input operands are equal.
- **FORWARDING_UNIT**: the bypassing operation has to be ensured both from the MEM and WB stages, if data dependencies between consecutive instructions are detected. The condition to be checked is the equality between source registers of the current instruction and destination registers of the previous ones. This control needs to keep track of the OPCODEs involved, in order to understand whether both source register fields have to be checked or only one of them. It is important to highlight that the result of each instruction can be involved in a bypassing operation.

1.5 Data memory access

This step manages the jumps and branches condition requests and provides as outputs the signals involved in the memory access.

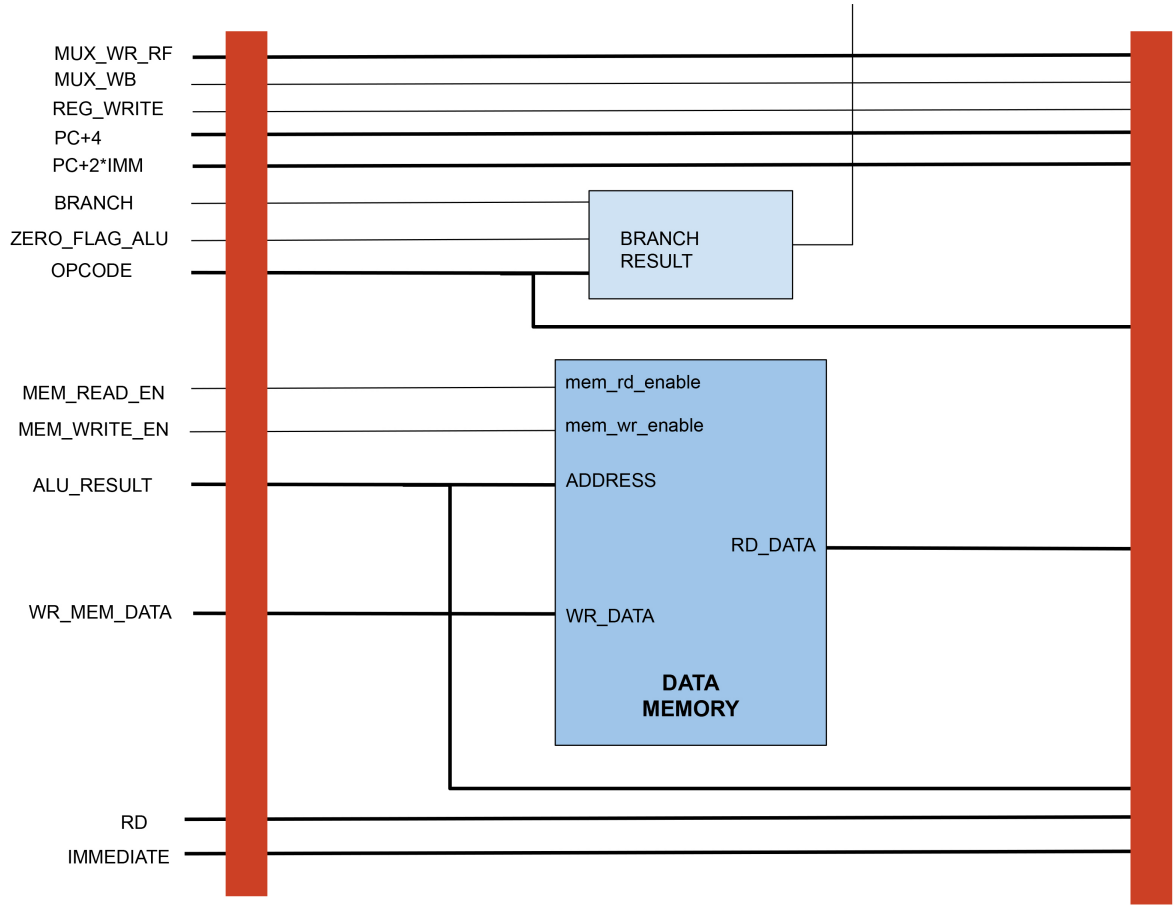

Figure 1.5: Data memory stage block scheme

- **BRANCH_RESULT**: whenever a BEQ instruction has to be executed, the branch control signal is sensitive to the zero flag provided by the ALU in the previous stage. On the other hand, in the case of a JAL instruction, the jump is forced to be taken.

1.6 Write back

This stage is designed to perform a write operation of the results into the register file. For this reason, all data and control signals involved in the write back phase have been properly delayed through the pipeline registers.

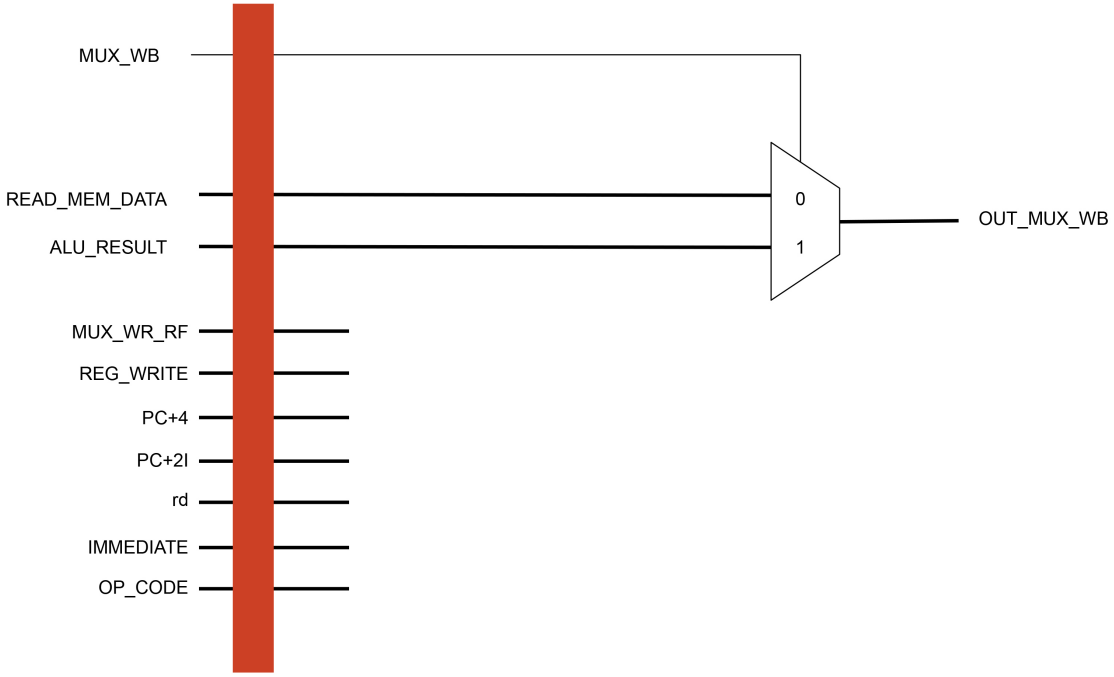

Figure 1.6: Write back stage block scheme

- **MUX_WB**: this multiplexer allows to select the data that have to be written in the register file. The selection can involve either the ALU result or the data read from the memory.

Testbench

In order to test the correct behavior of the RISC-V lite processor, an assembly algorithm, reported in the file *min-rv.s*, has been employed. The aim of the algorithm is to evaluate the minimum absolute value of a given set of signed numbers.

The given assembly instructions must be converted into binary code in order to be correctly interpreted by the designed architecture: a given tool, called RARS, can be exploited to accomplish this conversion. The obtained machine instructions are then dumped into the text file *instructions.txt*.

The execution of the program assumes that data are already stored into the proper data memory locations. Therefore, the needed data vector has been firstly identified from the assembly code and then reported on a text file, named *data.txt*. Data are supposed to be represented in 2'complement notation on 32 bits.

After considering all the needed initial data, a mixed VHDL-Verilog hierarchical testbench has been developed. The modules included in the testbench are represented in the following block scheme.

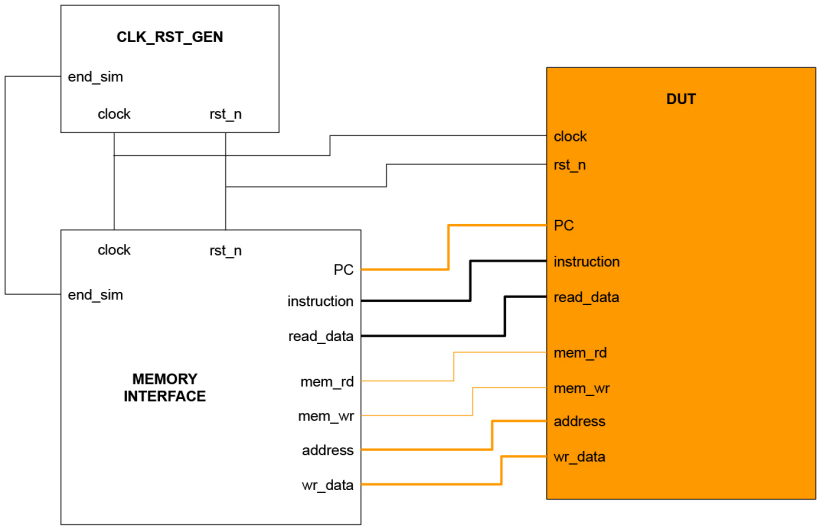

Figure 2.1: Testbench modules block scheme

- **Clk_rst_gen**: its purpose is to generate the initial reset condition and a clock signal with a given period. In order to correctly stop the simulation, the clock signal is frozen when an end simulation signal is asserted.

- **Memory_interface**: this entity is the core of the testbench. Its main purpose is to interact with the DUT, emulating the instruction and data memories. When the algorithm is succeeded and completed, an end simulation signal is asserted and the content of the data memory is reported on the *results.txt* file.
  During the reset condition, the content of the files *instructions.txt* and *data.txt* are read and reported in order into the first locations of the proper memories.
  Even if the number of instructions to be executed is 22, the number of allocated words in the memory is 32: the remaining locations are filled with NOP.
  Regarding the data memory, the first seven locations are filled with the array elements, while the last one is reserved to store the final result.
  Due to the provided machine instructions, the processor accesses the data memory in precise locations: in order to verify the correctness of the behavior, the MSBs of the requested addresses are checked before giving access to the memory.
  For sake of simplicity, a dummy string signal has been declared with the purpose of better understanding which instruction is currently fetched during the simulation.

- **DUT**: it is the RISC-V lite architecture, which represents the device under test.

Finally, the developed testbench has been launched on ModelSim and it provided the following results.

row	value
0	10
1	-47
2	22
3	-3
4	15
5	27
6	-4
7	3

The content of the file *results.txt* is reported in the table above: a correct behaviour is noticeable since the last row contains the minimum absolute value, considering the previous rows as input set.

Absolute value accelerator

In the provided algorithm, a sequence of operations needs to be applied on each element of the vector before comparing it with the temporary minimum. These instructions are aimed to calculate the absolute value of each array element. The mentioned sequence is reported below.

```
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)
```

As a consequence, each time a value is analyzed, these four instructions have to be executed.

The latency required to complete the algorithm can be reduced by exploiting the introduction of an hardware accelerator, involved in the absolute value computation.

3.1 Design

First of all, a new entity, named Absolute_Value_Unit, has been declared. Since it is meant to work with the other processor components, it is defined with a parallelism of 32 bits.
In order to perform this operation, an additional instruction has been introduced in the set.

Immediate	RS1	Func	RD	OPCODE
000000000000	01000	001	01010	0010011

As visible in the reported instruction, the OPCODE is designed to be the same of the arithmetic I-type instructions, since the accelerator works only with one operand. In order to distinguish this instruction from the other ones, the FUNC field has been customized.

As visible from the figure below, the developed entity has been inserted in the execution stage, so that it can be used in parallel with the ALU, exploiting its first input operand. An additional multiplexer has been instanciated to select either the ALU result or the accelerator one. The selection is driven by a further signal generated by the Control Unit.

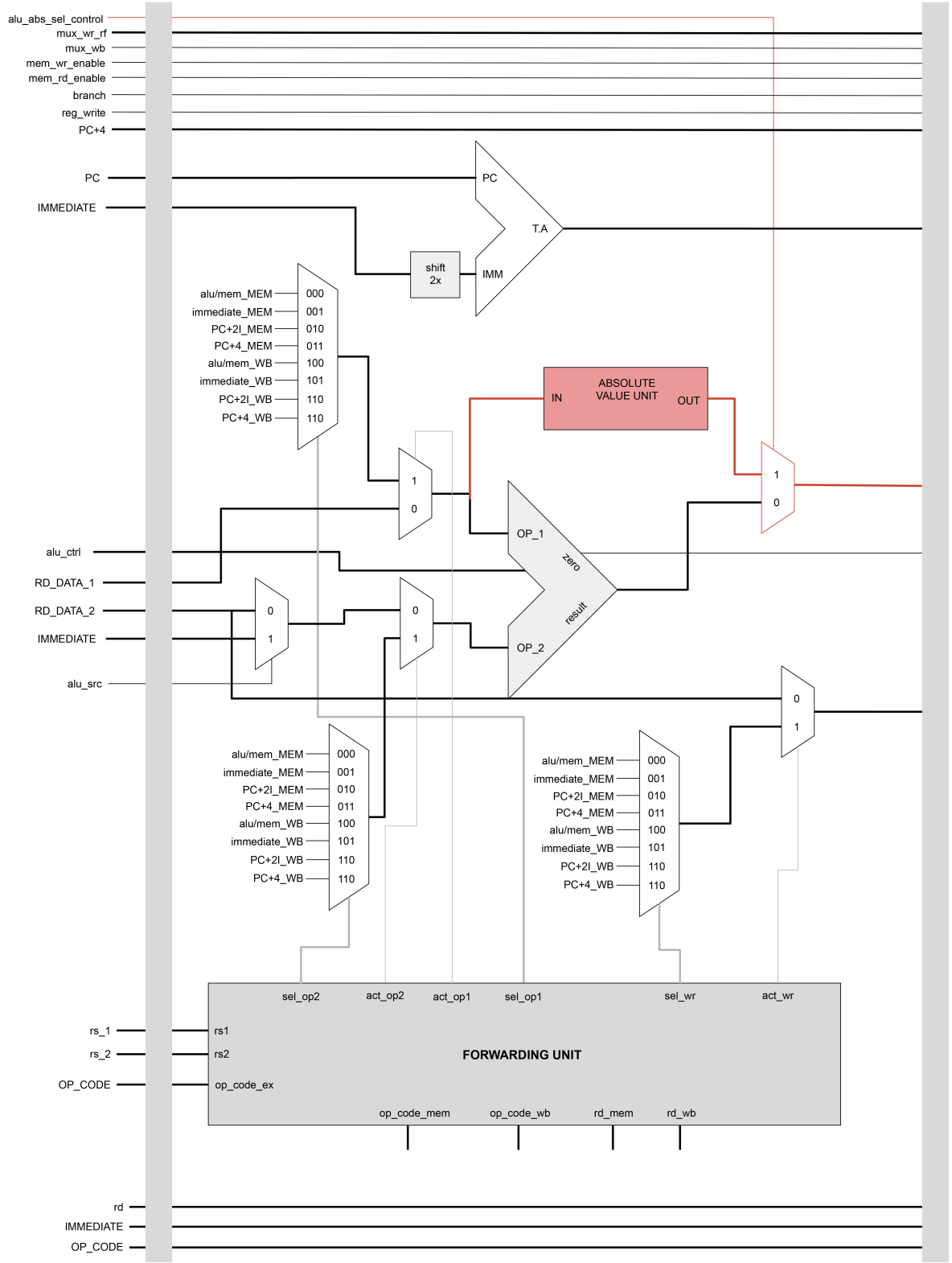

Figure 3.1: Absolute value accelerator in the execution stage

It is important to highlight that, since both the hardware accelerator and the relative instruction have been designed to be introduced in a transparent way, the Forwarding Unit is able to correctly apply the bypass mechanism, without introducing additional complexity in the operations management.

3.2 Simulation

In order to simulate the same algorithm on the modified RISC-V processor, the assembly code has been changed by replacing the sequence meant to evaluate the modulus with an ABS instruction.

abs x10, x8, 0

A conversion from assembly to binary is needed once again, to take into account the new relative target addresses for jump instructions.

The testbench previously developed has been adapted and the simulation has been launched. Also in this case, the processor is able to operate in a correct way.
In order to estimate the performance improvement, the end simulation time instants of the two architectures have been compared.

	RISC-V	**accelerated RISC-V**
end simulation [ns]	1440	1230
clock cycles	144	123

As expected, the accelerated RISC-V can execute the algorithm faster, with a performance improvement around 15%.

From the ModelSim waveform viewer, it is possible to notice that the overall execution time has been dramatically decreased by the forced introduction of NOP operations to manage jumps. The implemented solution is a basic approach to avoid control hazards during the execution; a further optimization could involve a speculation mechanism.

Synthesis and Place & Route

Both the designed processors have been synthesized with Synopsys Design Compiler. An iterative approach has been used to find the maximum working frequencies, starting from the application of 0 ns clock as a constraint and observing the slack.

After having synthesized the architectures using the maximum clock frequencies, the corresponding Verilog netlists and .sdf delays reports have been generated and the correct behaviour has been verified again through a ModelSim simulation.

Moreover, also a report of the total area has been generated and the following results have been obtained.

	RISC-V	**accelerated RISC-V**
$T_{clk,\ min}$ [ns]	3.95	3.96
$F_{clk,\ max}$ [MHz]	253.16	252.53
Area [um^2]	18008	18277

It can be noticed that the two minimum clock periods are quite similar, a slight increment is noticed in the accelerated architecture. The reason of this variation is related to the introduction of a multiplexer placed after the ALU unit, on the critical path.

From the area occupation point of view, as expected, the accelerated RISC-V covers a larger area due to the introduction of the absolute value accelerator.
As a consequence, it is expected a larger power consumption that could be reduced by properly gating the accelerator when not used.

The execution time of a generic algorithm can be expressed as function of the following quantities

$$T_{exe} = T_{clk} \cdot CPI \cdot \#\ \text{instructions}$$

As noticeable, the hardware accelerator impacts on the number of instructions parameter. The whole algorithm execution is sped up since less instructions are needed to compute the absolute value.

The next step is the application of the place & route procedure on both architectures using Innovus software. In order to accomplish that, several steps have been followed such as:

- Floorplanning
- Power planning
- Placement
- Filler placement
- Routing
- Verification

Moreover, several optimizations have been applied in between the steps listed above. Since no errors are reported after the verification processes, the two netlists and the corresponding .sdf files have been exported and the correct behaviors have been verified again on ModelSim.

Furthermore, area information has been collected on Innovus.

	RISC-V	**accelerated RISC-V**
Area [um^2]	16338	16697

As expected, the accelerated architecture occupies a larger amount of area, as commented before. Moreover, it can be noticed how Innovus has optimized the disposition and the routing of the used cells, with the consequent area reduction if compared to data obtained on design compiler.