Design of a RISC-V LITE Processor

Integrated System Architecture
Laboratory experience 3 - **Group 11**

Francesco	Federico	Mario
Ricci	Pozzana	Zurlo
252959	254758	253946

Contents

1 Background

A central processing unit (CPU), also called central processor, is the electronic circuitry within a computer that executes instructions composing a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by each instruction. The group of instructions that a processor can execute is called Instruction Set. Historically two main approaches were investigated:

1. CISC (Complex Instruction Set Computer): a single instruction can perform numerous low-level operations at once (like a load from memory, logic or arithmetic operations, memory store, etc). Each instruction can assume a different length, requiring a complicated decode stage. A CISC processor usually supports multiple addressing modes as well, resulting in a more complex architecture with respect to the RISC counterpart. Noticeable mentions are the Zilog Z80 and the Intel core CPUs family.
2. RISC (Reduced Instruction Set Computer): a single instruction is usually associated with a low-level operation. It is based on a load-store architecture (no support for complex addressing modes): as a result, the Instruction Set is composed by fewer instructions with respect to CISC counterpart. Each instruction has the same length, leading to a simpler decoding stage. Most commands are completed in one machine cycle, allowing the usage of pipelining to speed up RISC machines. Noticeable mentions are ARM based CPUs.

In the early stages of microelectronics development, the count of transistors integrated on a single silicon chip was very small. the CISC Instruction Set Architecture was adopted, because it was not effective to allocate a large part of available transistors to realize a large enough register file to support a load-store architecture. Today technology advancement forced the switch towards the RISC architecture (that leads to better performance, lower power consumption and smaller unit cost). While RISC is ubiquitous in many application domains (IoT, Embedded and Mobile Systems to name a few), it still struggles to enter other markets like the laptop/PC segment, due to lack of code portability between CISC and RISC processors.

2 RISC-V Brief Introduction

CPU design requires design expertise in several specialties: electronic digital logic, compilers, and operating systems. To cover the costs of such a team, commercial vendors of computer designs, such as ARM Holdings and MIPS Technologies, charge royalties for the use of their designs, patents and copyrights.
RISC-V was started to solve these problems. The goal was to make a practical ISA that was open-sourced, based on established reduced instruction set computer (RISC) principles, usable academically and in any hardware or software design without royalties.
Unlike other academic designs which are optimized only for simplicity of exposition, the designers state that the RISC-V instruction set is for practical computers. It has features to increase computer speed, yet reduce cost and power use, including load–store architecture, bit patterns to simplify the multiplexers across the CPU, a design that is architecturally neutral, and placing most-significant bits at a fixed location to speed sign extension.
The instruction set is designed for a wide range of uses. It is variable-width and extensible so that more encoding bits can always be added. It supports three word-widths, 32, 64, and 128 bits, and a variety of subsets.
The project began in 2010 at the University of California, Berkeley, but many contributors are volunteers not affiliated with the university.

3 Instruction Set

The Instruction Set of the Risc-V "Lite" is summarized in table 1.

It is a basic subset of the RV32I Instruction Set: it supports load and store (obviously), a jump and a branch type, AUIPC and LUI instructions (useful in case of branches/jumps), and some arithmetical and logical operations.

However, the processor design includes a Control Unit (and an Alu controller) able to support almost all instructions belonging to the RV32I Instruction Set (only CALL and RETURN functionalities are not supported,since a windowed register file must be employed).

funct7	rs2	rs1	funct3	rd	opcode	R-TYPE
imm[11:0]		rs1	funct3	rd	opcode	I-TYPE
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-TYPE
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	SB-TYPE
imm[31:12]				rd	opcode	U-TYPE
imm[20|10:1|11|19:12]				rd	opcode	J-TYPE

Risc-V Lite Instruction Set

							Description	Pseudo code
imm[31:12]				rd	0010111	AUIPC	Load Upper Immediate	rd ← pc + offset
imm[31:12]				rd	0110111	LUI	Add Upper Immediate to PC	rd ← imm
imm[20|10:1|11|19:12]				rd	1101111	JAL	Jump and Link	rd ← pc + length(inst) pc ← pc + offset
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ	Branch Equal	if rs1 = rs2 then pc ← pc + offset
imm[11:0]		rs1	010	rd	0000011	LW	Load Word	rd ← s32[rs1 + offset]
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	Store Word	u32[rs1 + offset] ← rs2
imm[11:0]		rs1	000	rd	0010011	ADDI	Add Immediate	rd ← rs1 + sx(imm)
imm[11:0]		rs1	111	rd	0010011	ANDI	And Immediate	rd ← ux(rs1) ∧ ux(imm)
0100000	shamt	rs1	101	rd	0010011	SRAI	Shift Right Arithmetic Immediate	rd ← sx(rs1) >>ux(imm)
0000000	rs2	rs1	000	rd	0110011	ADD	Add	rd ← sx(rs1) + sx(rs2)
0000000	rs2	rs1	010	rd	0110011	SLT	Set Less Than	rd ← sx(rs1) <sx(rs2)
0000000	rs2	rs1	100	rd	0110011	XOR	Xor	rd ← ux(rs1) ⊕ ux(rs2)

Risc-V Lite Instruction Set Custom Extension

						Description	Pseudo Code
imm[11:0] (ignored)		rs1	funct3	rd	1111111	ABS	Absolute Value	rd ← abs(rs1)

Table 1: Risc-V Lite Instruction Set

4 Design and architecture overview

A processor is mainly composed by two elements: a datapath and a control unit. The datapath contains all the functional units that may be used to execute correctly an instruction; the control unit is in charge of propagating correctly the instruction along the datapath, controlling the flow (multiplexers) and the functional units behaviour (enable signals, operation selection in ALU, etc).

Some instructions must propagate along more datapath components than others: the worst case is represented by the load instruction. In fact, it must go through Instruction memory → Register File → ALU → Data Memory → Register File. To improve the performance, a 5 stage pipeline organization is used:

1. Fetch (IF): a new instruction is read from Instruction Memory (IRAM)
2. Decode (ID): the instruction is decoded and register file is accessed to

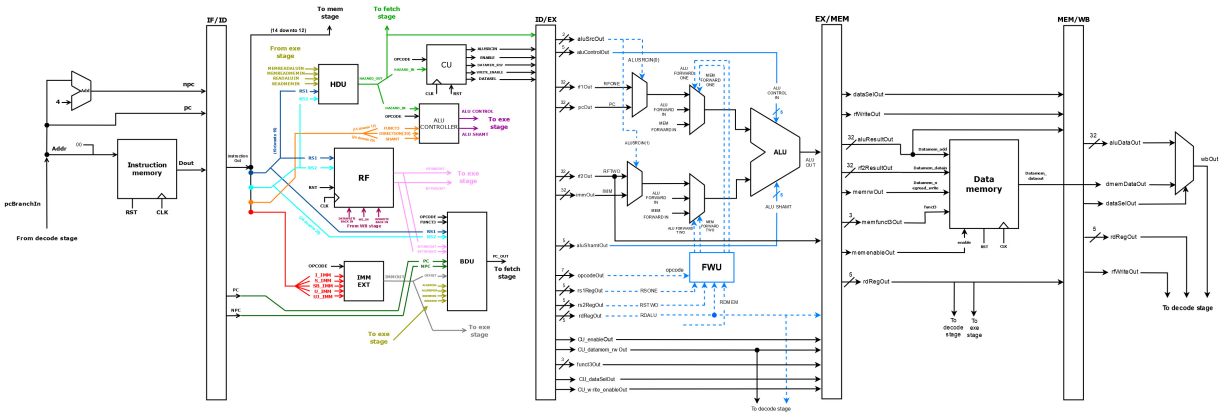

Fig. 1: Risk-V Lite Processor (Datapath and Memories)

obtain operands

3. Execute (EX): the instruction operands are combined (operation performed depends on the given instruction)
4. Memory (MEM): an instruction may need to access a data memory (DRAM) to store the result or read an operand
5. Write back (WB): the operation result is written back into register file

The pipelined processor takes the same control signals as the single-cycle processor and therefore uses the same control unit. The control unit examines the opcode and funct fields of the instruction in the decode stage to produce the control signals. These control signals must be pipelined along with the data so that they remain synchronized with the instruction. A view of the pipelined processor is proposed in figure 1

4.1 Fetch stage

The fetch stage basically contains the Instruction Memory (IRAM). The memory is accessed by means of a program counter (PC), that is incremented by 4 at each clock cycle (called Next Program Counter). Usually a multiplexer is present, that selects either the NPC or the target address of a Jump/Branch instruction. However, in our processor design, the multiplex is embedded inside the BDU, located at the decode stage. The fetch stage is summarized in figure 2. The Program counter register is not shown in the picture, because it is embedded in the register bank between decode and execute stages.

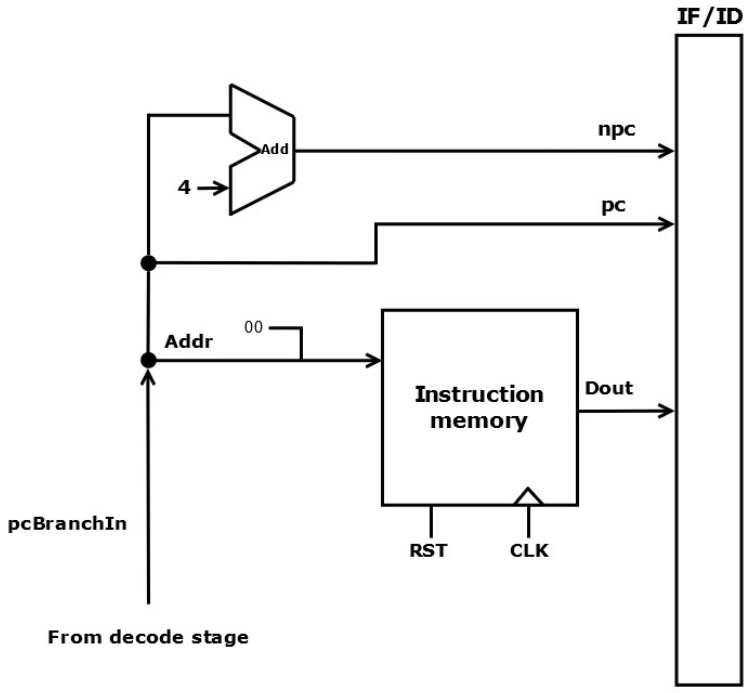

Fig. 2: Fetch Stage

4.2 Decode stage

The decode stage is the most important stage, and certainly the most complex. It has three main functionalities, each one realized by means of different components:

1. Produce operands: the execution unit (containing the ALU) receives always two operands.
   (a) Register File: provides one or more operands to the ALU inside the execution unit. It receives two 5-bit read addresses embedded into the instruction and one 5-bit write address (that is delayed to be synchronized with the result of the instruction computed in the write back stage).
   (b) Immediate Extender: depending on the kind of instruction, it retrieves and reorders the immediate bits embedded in the instruction. It also perform a proper extension to 32 bits.
2. Decoding and Control: the main functionality of the decode stage is to

identify the current instruction and to produce the corresponding control signals to configure next stages correctly.

(a) Control Unit: it looks at the opcode and funct3 fields to identify the current instruction. It produces control signals
- to propagate the correct operands to the ALU,
- to manage DRAM read/write operations,
- select the correct output for the write back multiplexer,
- to choose if the instruction result should be stored inside the register file.

(b) Alu controller: depending on the instruction, a 5-bit control signal is produced to tell the ALU (inside the execution stage) with operation to perform.

(c) Hazard detection Unit (HDU): if a data conflict is present, one or two stalls must be introduced into the pipeline. For a detailed description, please refer to section **5.3**.

3. Branch handling: a branch is taken if the corresponding condition is true. The condition test i can be performed by the ALU inside the execution stage or can be anticipated to decode stage by allocating a specific component: the Branch Decision Unit. Please refer to section 5.4 for more detailed description of the BDU component.

The decode stage is summarized in figure 3.

4.3 Execute stage

Due to the features of the decode stage, the execution stage is highly simplified. In fact, it hosts only two components, namely:

1. ALU, that can perform a variety of operations. It is controlled by the Alu controller in the decode stage and receive two operands from the decode stage.

2. Forwarding Unit (FWU), that aims at reducing the amount of data conflicts to speed up execution. Please refer to section 5.2 for more detailed information about forwarding.

The execution stage is summarized in figure 4.

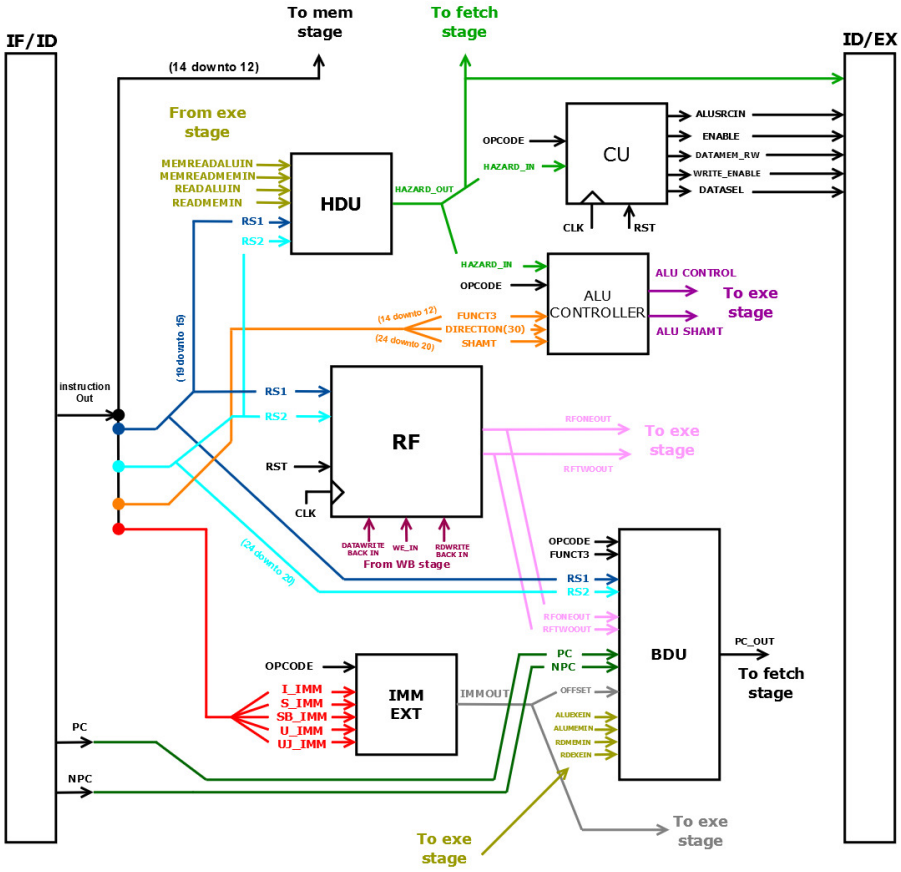

Fig. 3: Decode Stage

4.4 Memory stage

The memory stage, as the name suggests, contains the Data Memory (DRAM). It receives as address the outcome of the ALU: it may be accessed to retrieve or save data. The ALU outcome is also propagated to the write back stage. The memory stage is summarized in figure 5.

4.5 Write Back stage

The write back stage is basically a multiplexer, that propagates either the ALU outcome or the Memory location addressed by the ALU outcome. The Write Back stage is summarized in figure 6.

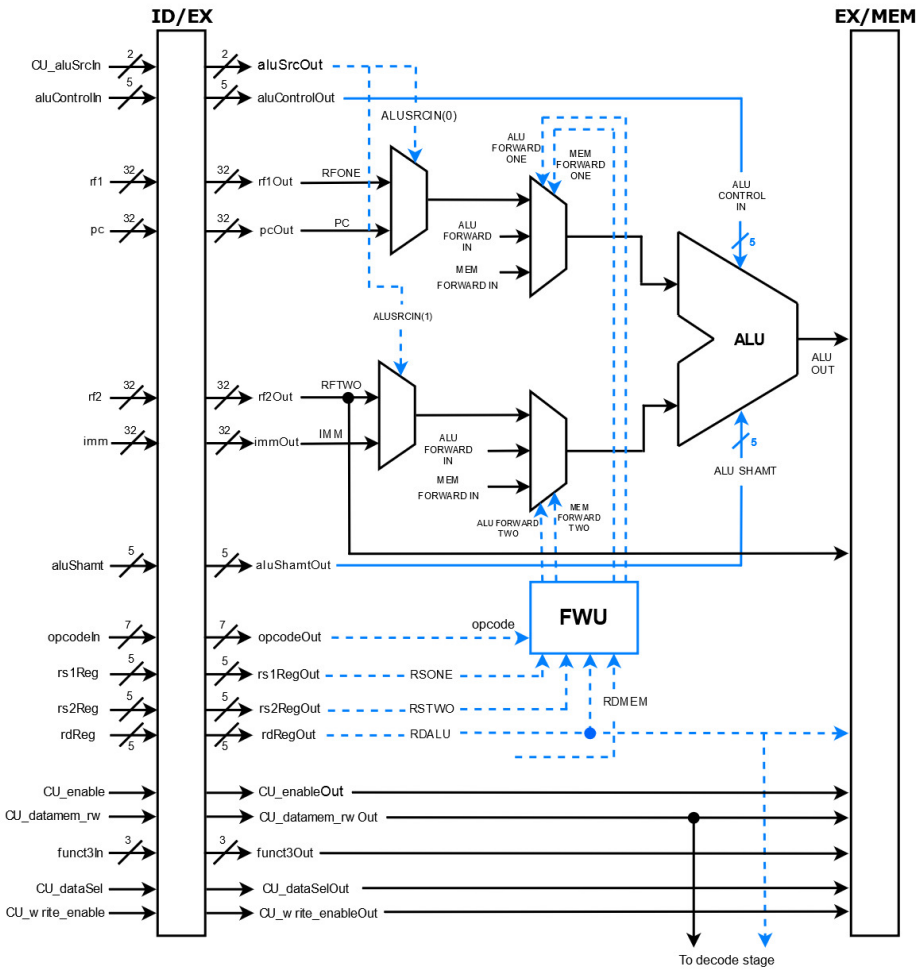

Fig. 4: Execute Stage

5 Advanced features

5.1 Data Memory

The CPU must interacts with two kinds of memories: instruction memory and data memory. While they may implemented using one single component (Von Neumann architecture), to avoid structural hazards in a pipeline processor, the two memories are kept separated (Harvard architecture).

Technically, memories are not part of the CPU. However, in modern processors, complex memory hierarchies are exploited (to obtain a good trade-off between speed and cost): usually cache memories are integrated on the same die as the processor datapath.

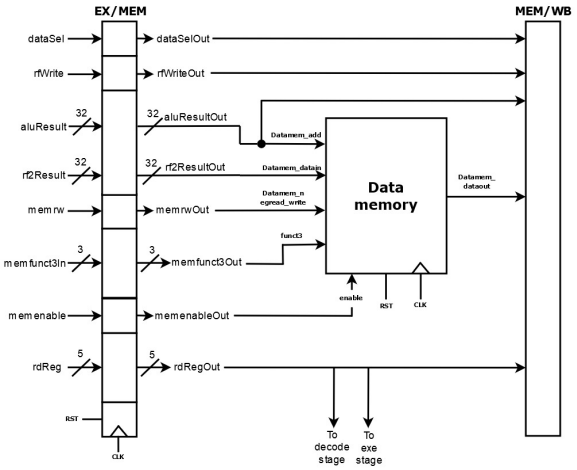

Fig. 5: Memory stage

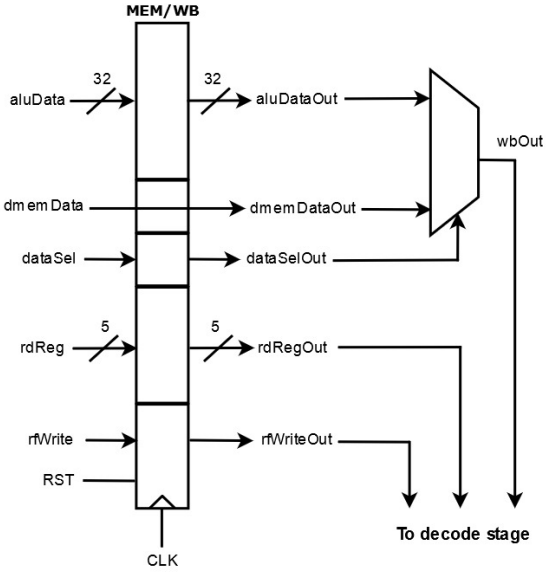

Fig. 6: Write Back Stage

The two memories have one input port and one output port, and a parallelism of 8 bits. The data is stored following a Little Endian ordering, to match the Risc-V architecture. However, the data memory is a bit more flexible, because was designed to support all different kinds of loads and stores available in the RV32I Instruction set:

- **LB (load byte):** the output is composed by the byte pointed by the address (placed in the lowest significant position) and 24 bits sign extension;
- **LH (load half):** the output is composed by two consecutive bytes, placed in the lowest significant position (address points at the lowest significant one) and 16 bits sign extension;
- **LW (load word):** the output is composed by four consecutive bytes (address points to the lowest significant byte);
- **LBU (load byte unsigned):** the output is composed by the byte pointed by the address (placed in the lowest significant position) and no sign extension (24 zeros)
- **LHU (load half unsigned):** the output is composed by two consecutive bytes (address points at the lowest significant one) and no sign extension

(16 zeros);

- **SB (store byte):** the addressed memory location is updated;
- **SH (store half):** four memory locations are updated (address points at lowest significant byte);
- **SW (store word):** four memory locations are updated (address points at lowest significant byte).

To support such operations, the data memory was designed to receive multiple control signals:

1. 3-bit control signal (encoded in the load and stores instructions of the Risc-V itself), to distinguish among the different loads or stores;
2. 1-bit enable signal, to grant permission of read or write into the data memory. The signal is generated by the control unit by looking at the instruction opcode;
3. 1-bit read/write control signal, to correctly access the memory. The signal is generated by the control unit by looking at the instruction opcode;

All signals are computed at the decode stage, and are opportunely delayed 2 clock cycles to keep consistency across the pipelined architecture.

The support for multiple load/store types was embedded in the memory itself, but it can be possible to extract the functionality into a memory controller. In any case, this functionality requires the described control signals surrounding the memory stage.

All instructions were tested and are working; the figure 7 showcase the functionality. The data memory testbench is included in the project folder.

5.2 Forwarding

To reduce the overhead caused by data hazards (introduction of pipeline stalls or "bubbles"),

A data hazard can lead to a pipeline stall when the current operation has to wait for the results of an earlier operation which has not yet finished. to limit the performance deficits related to data hazard, a forwarding unit was included into the execution stage.

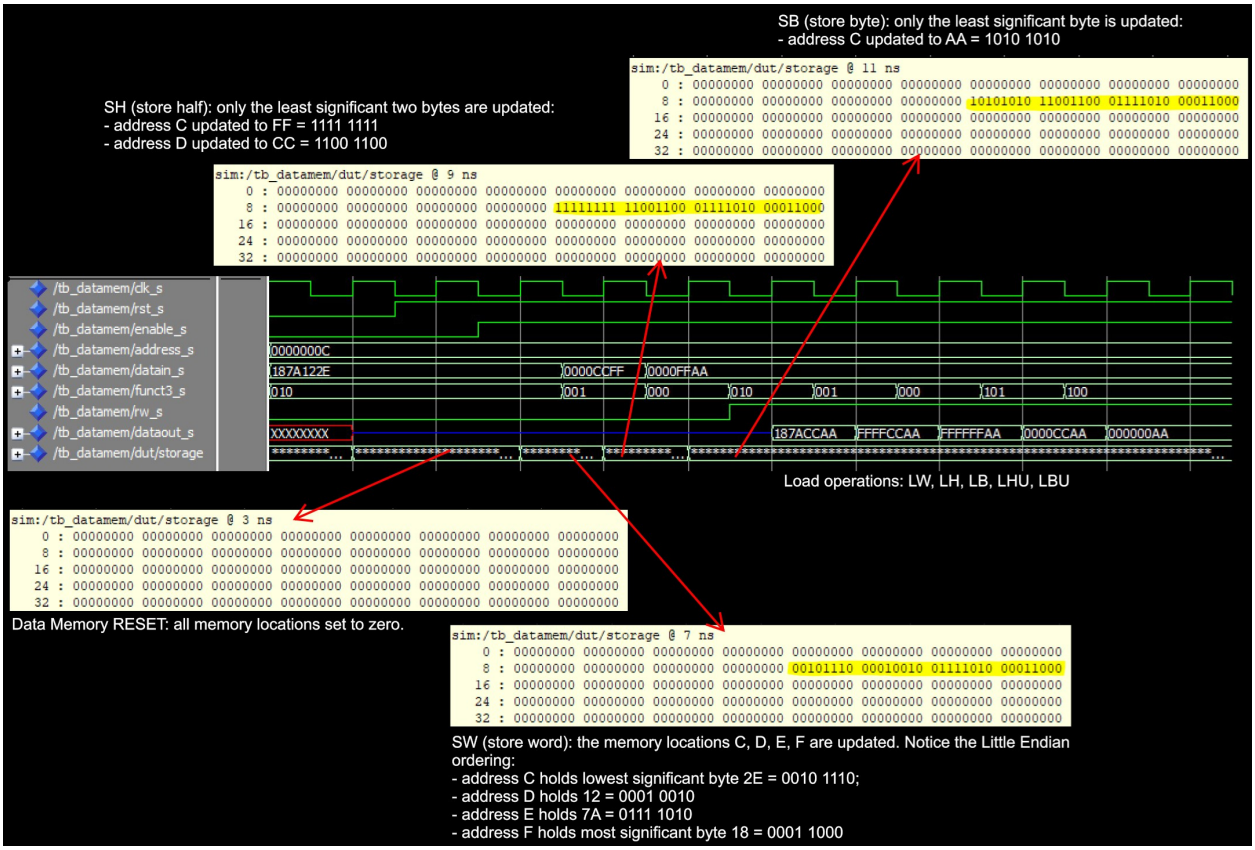

Fig. 7: Data memory test for all different kinds of loads and stores.

The forwarding mechanism allows the usage of a result as soon as it is computed, without waiting to write it back into the register file. This is achieved by introducing some extra connections inside the datapath, and by preforming a comparison between current source registers and the destination registers of previous two instructions. If the comparison has a true outcome, the forwarding mechanism is activated and the forwarding multiplexer propagates the most recent result. The comparison prioritize the most recent data from execution stage over the older result coming from the memory stage, in case the destination register of the two previous instructions in the same.

5.3 Hazard Detection Unit

The forwarding mechanism is a great tool to avoid the introduction of stalls in the pipeline. However, it can't forward back in time! Let's explain this with two similar examples:

```
# Example 1
ld x1, 0x(x2)
sub x4, x1, x5 # data conflict
...
...
```

```
# Example 2
ld x1, 0x(x2)
add x6, x7, x8 # no data conflict
sub x4, x1, x5 # data conflict
...
```

The load instruction computes the value to be stored in *x1* only at the end of memory stage: when the *sub* is at the execution stage, the result is not computed yet!

The HDU performs comparisons between the destination register of load instructions with source registers of the two subsequent instructions. If the outcome of the comparison is true, a stall must be introduced inside the pipeline. Up to two stalls can be introduced (Example 1: the comparison outcome is true for two subsequent clock cycles); after that the register file content is updated and the program execution resumes.

5.4 Branch Decision Unit

The idea of a Branch Decision Unit (or BDU for short) is the reduction of the overhead introduced by branch operations. In principle, to know if a branch is taken or not, we must wait until the branch instruction is inside the execution stage. In fact, the ALU inside the execution stage is in charge of test conditions ($=, !=, <=, >=, etc.$).

The BDU anticipates the branch condition test at the decode stage: therefore, it can be seen as a redundant smaller ALU, anticipated at the decode stage. There are two main benefits and one main drawback:

- ✓ The branch test condition is computed earlier, thus speeding up the program execution by one clock cycle every time the branch is taken;
- ✓ if a branch is taken, all instructions that entered the pipeline afterwards must be eliminated. To erase an instruction from the pipeline, the control signals must be modified on the fly. Without the BDU, the control signal override is more complex, because the instruction following the branch

is leaving the decode stage and it is entering the execution stage. If the BDU is implemented, the instruction must be erased is leaving the fetch stage and it is entering the decode stage (where the control unit is located).

× It requires extra hardware resources (a small ALU!), increases complexity and possibly the overall critical path.

5.5 Register File Bypass

To further reduce the overhead of RAW hazards, the register file was designed to support a bypass functionality. The bypass consist in providing as output the input data of the register file, instead of the content of the register file location. The bypass is performed only if one of the source registers of the current instruction is equal to the destination register of an instruction decoded 3 clock cycles earlier. The bypass allows to avoid the introduction of a stall, and can be seen as an extension of the forwarding functionality implemented inside the execution stage.

The bypass functionality was implemented in a full behavioural way, but a possible structural implementation is suggested in figure 8. If a RAW hazard is present (one of the source registers of current instruction is the same as the destination register of 3-clock-cycles previous instruction) the multiplexers propagate the second input (data_in of the RF), bypassing the register file entirely.

5.6 Control Unit & ALU Controller

The complexity of the datapath and the variety of operations to support (that require a different interaction with different functional units), implies a large amount of control signals to be correctly defined and propagated across the pipelined stages, by means of a Control Unit (CU for short).

Our design includes a Control Unit able to support almost the whole RV32I instruction set (only CALL and RETURN functionalities are not supported, since a 'Windowed' register file is necessary). For the purpose of this laboratory experience, the extra instructions are commented, reducing the complexity of the processor, even if it is possible to unlock full functionalities.

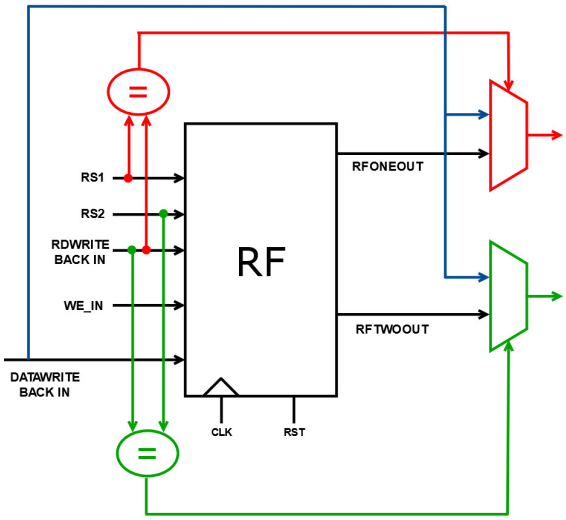

Fig. 8: Register File Bypass suggested HW implementation

5.7 Custom ABS instruction

To implement a custom instruction, a new opcode (”1111111”) was selected. The abs instruction was classified as an I-TYPE, even if the immediate field is actually ignored. The *funct3* is usually exploited to define functionally similar operations under the same opcode; this field was also ignored, since no variations of absolute value operation were designed.

To add the custom instruction to the existing datapath, the following updates were made:

1. The control unit was updated to support the extra opcode: the control signals are the same of every I-TYPE instruction.
2. The Alu controller supports the extra opcode: a new 5-bit control signal was defined.
3. The Alu component (implemented behaviourally) receive 5-it control signal from Alu controller; if it receives ”10100”, it performs the absolute value operation.
4. The immediate extender was also updated, to give as output an all zeros signal. This is not necessary to compute the absolute value (it not even used by the Alu), but was handy during testing.

6 Simulation

The simulation of the processor was performed using the ModelSim tool from Mentor Graphics. To speed up the design, and to simplify the troubleshooting, each stage of the processor was described as an independent entity. A testbench for each stage was produced, mainly to remove as much design errors a possible. After that, the datapath entity (that connects together all 5 stages) was tested. To simplify the simulation, the waveform configuration was saved (file extension .do), in order to save the carefully handpicked signals and load them for every future simulation. In fact, the "distributed approach" of one entity per stage (and one entity for each set of registers between adjacent stages) produces a lot of redundancy in terms of waveform signals. Therefore, signal repetitions were removed as much as possible. Moreover, we added the register file and DRAM data structures to monitor the correct outcome of each instruction.

The testbench used is included in the project folder.

7 Synthesis

After the simulation, the design has been synthesized using Design Compiler from Synopsis. The same additional constraints used in previous lab experiences (suggested in the design flow document), were applied during the synthesis process. A script was made to speed up the synthesis process; every component has been analyzed and only the datapath has been synthesized (i.e. memories were not included).

Two different datapaths, namely LITE and ABS, have been synthesized:

- LITE: processor that supports the Risc-V Lite Instruction Set;
- ABS: processor that supports the custom extension, reported in table 1.

Both synthesized netlists have been simulated afterwards, to verify their correct behavior.

7.1 Synthesis process

Starting from the VHDL high level hierarchical description, the synthesis process produced two Verilog netlists (alongside .ddc, .sdc and .sdf files) ready

to be used for the place&route process. Area, timing and power reports have been obtained for both synthesized datapaths. The results obtained can be observed in table 2.

Design	LITE	ABS
Timing $[ns]$	3	3
Area $[\mu m^2]$	15168.650180	15247.652166
Power $[mW]$	3.7039	3.7228

Table 2: Synthesis result summary

7.2 Synthesis results

As we can see from table 2 the design with the custom instruction have a larger area (0.5% increase) and a bigger power consumption (0.5% increase). The constraint on the clock period have been set to 3 ns for both designs and have been met by both designs as well.

7.3 Post synthesis simulation

Both netlists have been simulated in order to guarantee the correct behavior; since after the synthesis process the internal signals couldn't be observed anymore (every behavioral block and names have been replaced by gates belonging to the linked library), looking at the DRAM content was the only method possible to verify the correct functionality.
For both designs in the right memory cell (the one right after the vector of values initialized at the reset signal) the value 3 (000...0011 in binary) was stored, as expected.

8 Place and route

The place and route was performed using the Cadence Innovus tool on the synthesized design. Following the same steps as in place&route process of previous laboratory experiences, the two implemented datapaths have been routed.

8.1 Place&Route process

A brief explanation of each step of the place and route process is reported in the following:

1. **Design import**: the names of the Verilog netlists came out from synthesis process have been modified to match the Top Level Design entity in order to fit in the design.globals file. The design.globals file was used to import designs.

2. **Floor-planning**: it consist in the draw of the base of the chip on which it will be placed cells. It is a critical step, because a bad floorplan can create issues (congestion, timing, noise) and can increase area, power and reliability. The area and the height of each cells were defined, specifying the Core aspect ratio (1.0), utilization (0.6) and the Core to die boundary ($5\ \mu m$).

3. **Power planning and routing**: any digital circuit requires supply voltage and ground. By means of the *power planning* tab of Innovus, two metal rings were placed on the chip. By means of the *connect global nets* tab, all the necessary wires to distribute the power and ground to the standard cells were installed.

4. **Cell placing**: the standard cells were placed; Innovus provides an abstract representation of each cell with corresponding sizing, orientation and position of its pins. A Post Clock-Tree-Synthesis optimization was performed to become compliant with the timing constraints.

5. **Signal routing**: at this moment all the cells composing the design were laid out, as well as all the power supply and ground connections. All the connections between cells are still missing, and are routed by running the *NanoRoute* command. The connections are routed on different metal layers, to avoid congestion; timing crucial connections are routed first.

6. **Timing and design analysis**: the place and route is now complete, and the design must be tested to ensure that timing and design constraints are met.

8.2 Place&Route results

In table 3, the main design parameters obtained after the P&R process are summarized.

Design	RISCV_DATAPATH_LITE	RISCV_DATAPATH_ABS
Gates	19292	19474
Cells	7622	7754
Area $[\mu m^2]$	15397.4	15541.5

Table 3: Place and route result summary

In the datapath supporting the ABS instruction it is possible to notice a 0.94% increase in the number of gates and thus in the overall area.

8.3 Post P&R simulation

Two scripts were developed in order to simulate the routed netlists for the LITE and ABS implementation. As in the post synthesis simulation, the correct functionality of the routed datapaths was checked by reading the DRAM content.