RISC V PROCESSOR

Duțu Alin Călin

UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
1818

CONTENTS

1 INTRODUCTION

1.1 Context

This document outlines the design and implementation of a pipeline for a RISC-V processor. The project focuses on building an efficient instruction pipeline to optimize the execution of RISC-V assembly instructions. The pipeline is designed to improve the overall processing speed by enabling multiple instructions to be processed simultaneously at different stages.

The RISC-V architecture is gaining widespread adoption due to its open-source nature, modularity, and flexibility. One of the key features of modern processor design is the use of a pipeline to improve performance by overlapping the execution of multiple instructions.

1.2 Objectives

This project aims to implement a pipeline for a RISC-V processor, discovering the benefits and capabilities of setting up the assembly line for CPU operations. The following objectives are taken into consideration:

- Discover the stages of the assembly line
- Understand the roles of each assembly stage for a RISC-V processor
- Design and implement a multi-stage instruction pipeline for a RISC-V processor

2 PROPOSED SOLUTION

The project aims to implement a pipeline for a RISC V processor that can execute a significant number of instructions. This pipeline is designed to improve the overall processing speed by processing multiple instructions simultaneously at different stages. The nature and complexity of the project have led to the design of the following architecture:

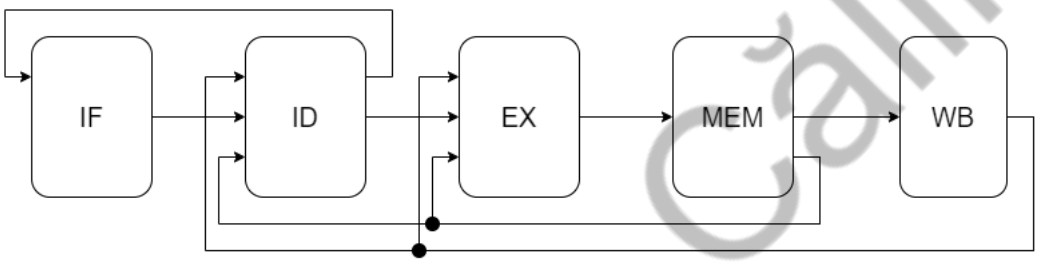

Figure 1: Architecture

There are 5 stages identified in the pipeline's architecture:

- Instruction Fetch stage (IF) - serves the crucial role of fetching the next instruction to be executed from the memory
- Instruction Decode stage (ID) - it is responsible for interpreting the fetched instruction and preparing the necessary data for execution
- Execution stage (EX) - performs the actual computation or address calculation for the executing instruction
- Memory Access stage (MEM) - handles operations related to reading from or writing to memory
- Write-Back stage (WB) - writes the results of executed instructions back to the register file

The best-case scenario for the pipeline's functionality is considered when every pipeline stage has an instruction to execute without any conflicts that can generate hazards. However, up to this moment, no processor can successfully avoid all hazards as multiple scenarios lead to such conflicts. Therefore, it is mandatory to implement a module that handles hazards.

3 IMPLEMENTATION DETAILS

3.1 Instruction Fetch stage

The Instruction Fetch stage is the first stage of the pipeline. Its main purpose is to bring instructions into the processor so that subsequent stages can process and execute them. This is the representation[1] of the IF stage:

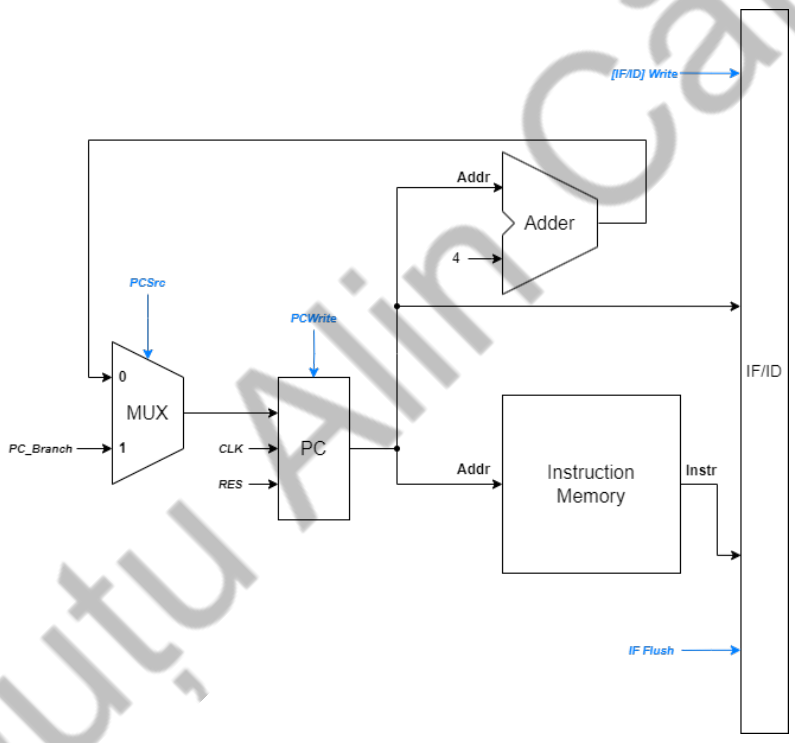

Figure 2: Instruction Fetch Architecture

The first instruction component is a 2:1 MUX that decides based on the $PCSrc$ selection entry what instruction should be executed next. In position 0 of the MUX, the address of the current instruction to execute is found, whilst in position 1, an older instruction address might be found. The MUX will choose the older instruction when a hazard that requires changes in the execution order is detected. Otherwise, the address of the current instruction is going to be processed.

Following the MUX component, there is a PC unit which is simply a D-register that receives the provided instruction, a clock, and a reset input and returns either the instruction's address

when the clock is positive or 32 bytes of 0s when reset is triggered or when the clock is negative.

The result of the PC unit is going to be processed by the Adder which provides the address for the next instruction, and by the Instruction Memory Unit which returns the instruction from the provided address.

In the end, the block that splits the IF stage from the ID stage transfers the instruction to be processed, the address of the instruction, and the IF Flush flag, if the $[IF_ID]Write$ signal is positive.

3.2 Instruction Decode stage

The Instruction Decode stage is the second stage of the pipeline and is responsible for interpreting the fetched instruction and preparing the necessary data for execution. The following diagram is the representation[1] of the ID stage:

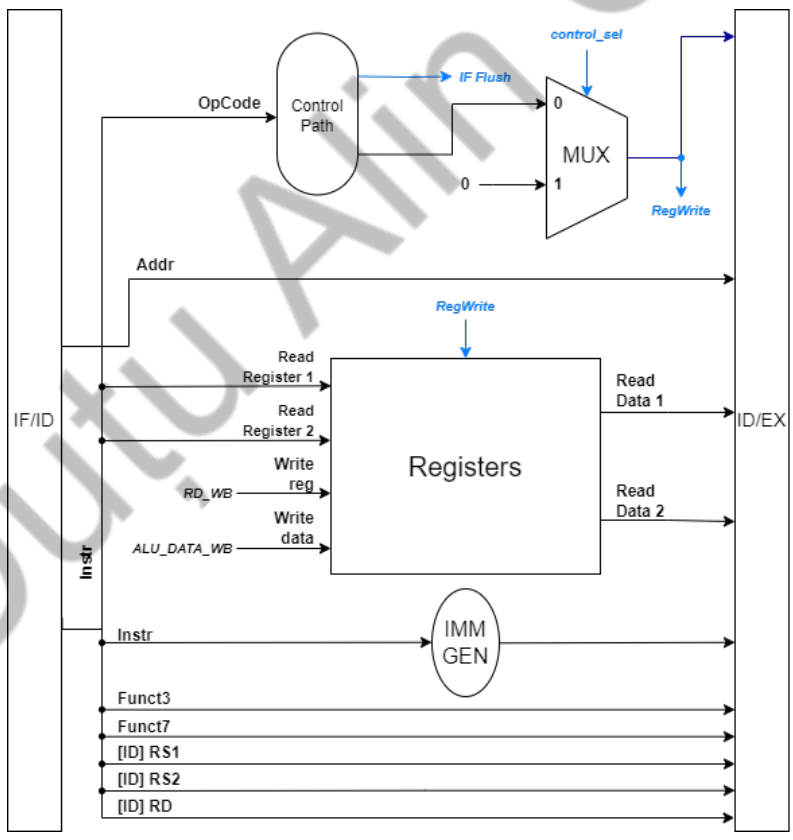

Figure 3: Instruction Decode Architecture

There are 2 main components in this stage. The first module is the Registers module and it contains 32 registers of 32 bits used for reading and writing based on the input. The module

itself represents a multi-port memory with two input ports for reading instruction operands and one output port for writing the instruction result. The reading and writing actions are executed only when the clock is at a positive edge.

When the $RegWrite$ signal is set to true, it activates writing the $WRITE\_DATA$ data in one of the module's registers identified by the $WRITE\_REG$ address. The register reading is done asynchronously as soon as the reading addresses from the module's outputs are changed.

The other module is the IMM GEN module, which generates immediate values based on the instruction type. However, only some instructions are going to be processed to simplify the project's difficulty since this is a discovering project. Here is the classification of the compatible instructions based on their type:

- I type: lw, $addi$, $andi$, ori, $xori$, $slti$, $sltiu$, $srli$, $srai$, $slli$
- S type: sw
- B type: beq, bne, blt, $bgem$, $bltu$, $bgeu$

Next, the figure below shows how are the immediate values generated based on the instruction type:

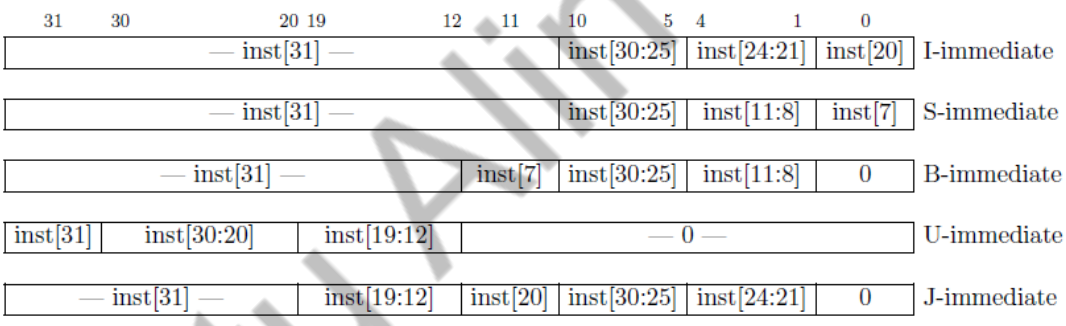

Figure 4: Immediate values for each instruction type[2]

Combining the figure and the classification defines the module's logic, which generates the immediate value based on the provided instruction.

In the end, the block that splits the ID stage from the EX stage transfers the address of the instruction, the reading addresses, the generated immediate value, and parts of the actual instruction that represent the $funct3$, $funct7$, $RS1$, $RS2$, and RD.

3.3 Execution stage

The Execute stage is the third stage in the pipeline and it is responsible for performing the actual execution or address calculation for the instruction. The next figure is the architecture[3] of the EX stage:

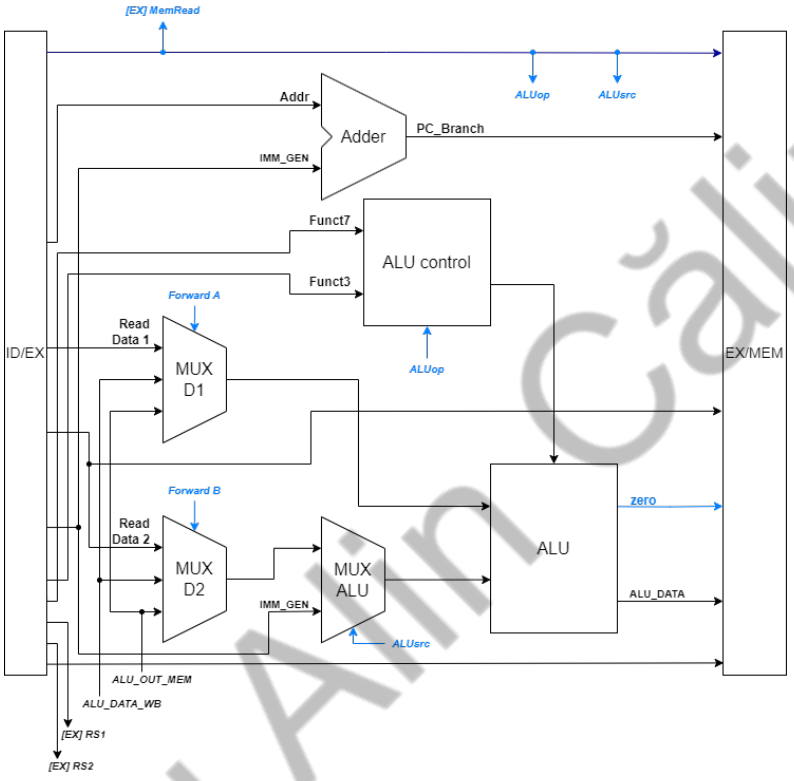

Figure 5: Execution stage Architecture

There are multiple modules that have different roles in the Execution stage. The Data MUXes set the operands that take part in the instruction execution process. There are multiple sources from which the instruction operands are taken: data from registers, ALU Data from the WB stage, ALU output from the MEM stage, or the Immediate value. The operands are chosen based on the $ForwardA$, $ForwardB$, and $ALUsrc$ signals.

In addition, the ALU requires to know the operation to be executed. So, the ALU control takes this responsibility by providing the operation based on the $funct3$, $funct7$, and $ALUop$ signals that determine the instruction and associate an $ALUinput$ code. In the tables down below there are the logic table[2] for generating the $ALUinput$ code along with the table with the associations between the instruction and the $ALUinput$ code:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Figure 6: Logic table for ALUinput code generation

Instruction	ALUinput code
ld, sd	0010
add	0010
sub	0110
and	0000
or	0001
xor	0011
srl, srli	0101
sll, slli	0100
sra, srai	1001
sltu	0111
slt	1000
beq, bne	0110
blt, bge	1000
bltu, bgeu	0111

Figure 7: Instruction - ALU code association

With the operands chosen and the ALU operation code set, the ALU module does the specified operation and puts the result in the ALU_DATA output along with a $zero$ signal that notifies the processor if the output is zero or not.

Furthermore, the Execution stage has an adder used to do the sum of the address computed in the IF stage with the immediate value computed in the ID stage to calculate the address of the next instruction to be executed which is found in the PC_Branch signal.

In the end, the block that splits the EX stage from the MEM stage transfers the PC_Branch signal, the ALU_DATA output, the $zero$ signal, and the instruction's RD.

3.4 Memory Access stage

The Memory Access stage is the fourth stage in the pipeline and handles operations related to reading from or writing to memory. More details can be seen in the following diagram[3]:

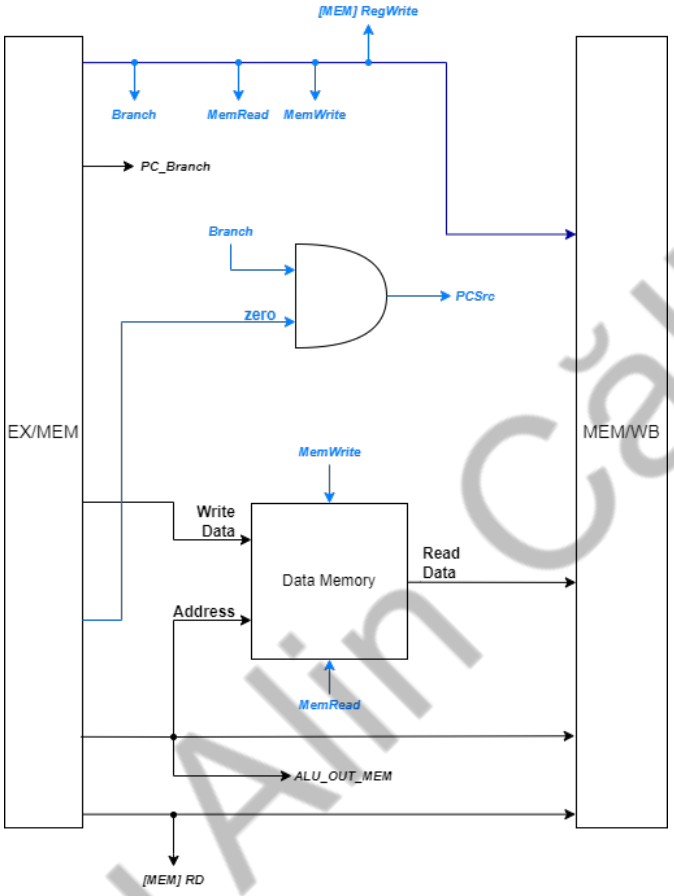

Figure 8: Memory stage Architecture

The main module of this stage is the Data Memory module which either writes data at the specified address or reads data from the specified address based on the $MemWrite$ and $MemRead$ signals. When a write is executed, the written data is saved in the Data memory module every time the address is changed, however, when a read is executed, it is done when the clock is at the positive edge.

Additionally, there is a module that makes logic AND between the $Branch$ and $zero$ signals, the result being set in the output $PCSrc$ signal.

In the end, the outputs of the MEM stage are PC_Branch and ALU_OUT_MEM and the block that splits the MEM stage from the WB stage transfers the Read data output, the result from the EX stage's ALU, and the instruction's RD.

3.5 Write-Back stage

The Write-Back stage is the final stage of the pipeline, and its primary purpose is to write the results of the executed instructions back to the register file. This stage ensures that the result of an instruction execution or a memory load is correctly stored in the destination register. This specific detail can be noticed in the next architectural schematic[3]:

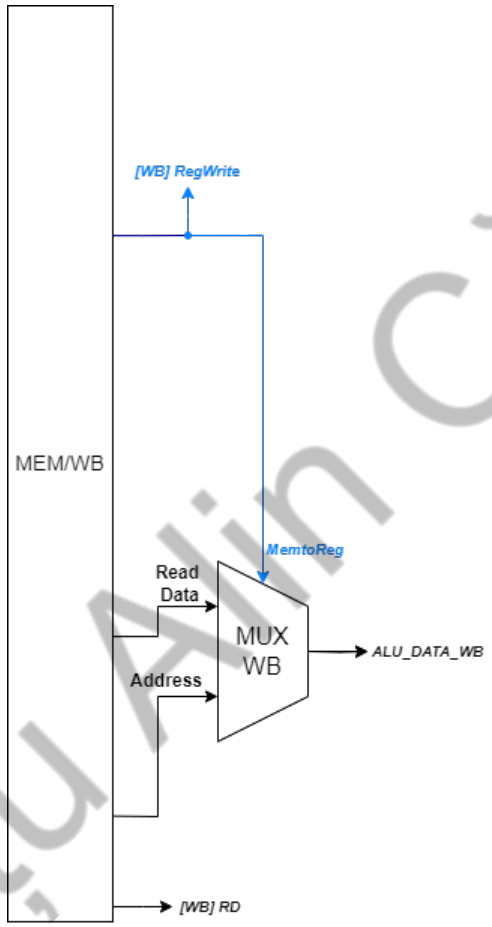

Figure 9: Write Back Architecture

This stage has a MUX module that receives read data and an instruction address as input and based on the $MemtoReg$ signal returns one of the inputs as the ALU_DATA_WB output. Moreover, this stage has the ALU_DATA_WB and the RD_WB as outputs.

3.6 Hazard modules

3.6.1 Forwarding Unit

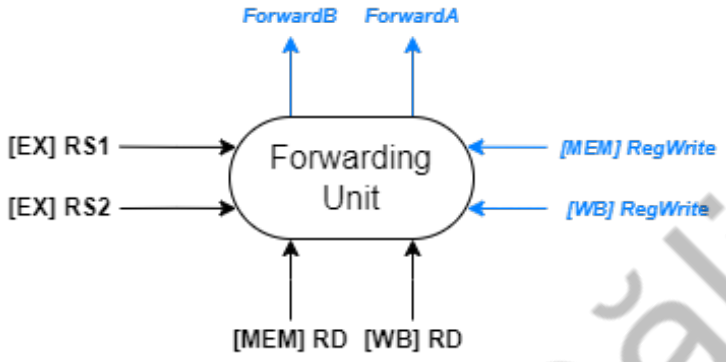

Figure 10: Forwarding Unit Architecture

The Forwarding Unit is a specialized module[3] for resolving Read after Write hazards and it's the first unit that takes this task. This unit bypasses the values calculated in ALU which were not written in the registry module and are still in the pipeline.

It does this by using the $RS1$ and $RS2$ outputs from the ID stage and the RD register from the MEM and WB stages respectively and generates the $ForwardA$ and $ForwardB$ signals for the EX stage MUXes. This module detects RAW hazards from the EX and MEM stages.

3.6.2 Hazard Detection Unit

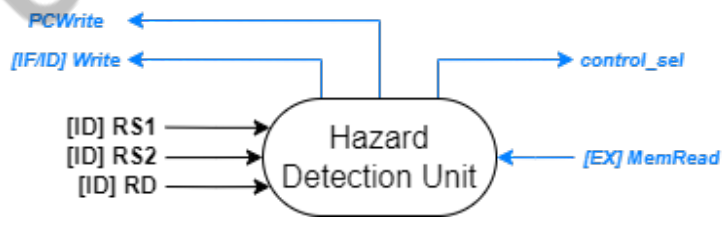

Figure 11: Hazard Detection Unit architecture

The Hazard Detection Unit is another specialized module[3] with the main role of stalling the pipeline when a Read after Write hazard is detected until it is resolved. The inputs used are the $MemRead$ signal and the RD output from the EX stage along with the $RS1$ and $RS2$ outputs from the ID stage.

The module checks if the $MemRead$ signal is not set to 0 to see if there is a read, and then it checks if the RD coincides with either $RS1$ or $RS2$ meaning that the reading address is the same as the register that stores the result from a loading instruction, which is the definition of a Read after Write hazard.

If a hazard has been detected, the $PCWrite$ and $[IF/ID]$ $Write$ are set to 0 and the $control\_sel$ is set to 1, which means in practice, adding stalls in the pipeline. This is going to take action if the Forwarding Unit cannot resolve the RAW hazard. If no hazard has been detected, the $PCWrite$ and $IF/IDWrite$ are set to 1 and the $control\_sel$ is set to 0.

3.6.3 Control pipeline

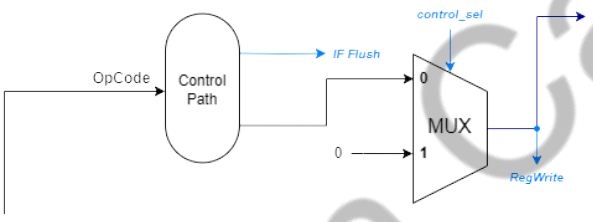

Figure 12: Control Path pipeline architecture

The Control pipeline[3] executes in the ID stage and contains 2 elements: The Control Path Module and the Control MUX.

The Control Path module takes the $OpCode$ from the ID stage and based on the current instruction type it generates the $ALUSrc$, $MemtoReg$, $RegWrite$, $MemRead$, $MemWrite$, $Branch$, and $ALUop$ signals.

Meanwhile, based on the $control\_sel$ signal provided by the Hazard Detection Unit, the Control MUX either sets the signals to 0 to set the stalls in the pipeline or lets the signals pass as they are. The results provided by the MUX will be distributed to the following stages.

3.7 Testing

For testing the proposed solution, a test input[3] is provided with the following instructions for the new processor to execute:

```
00008133  //add x2,x1,x0
00108093  //addi x1,x1,1
0020F1B3  //and x3,x1,x2
```

```
0010E213  //ori x4,x1,1
0042A223  //sw x4,4(x5)
00802603  //lw x12,8(x0)
04090E63  //beq x18,x0,5c
```

The simulation output, after a thorough analysis, matches the expected ones hence the implemented pipeline achieves its objective.

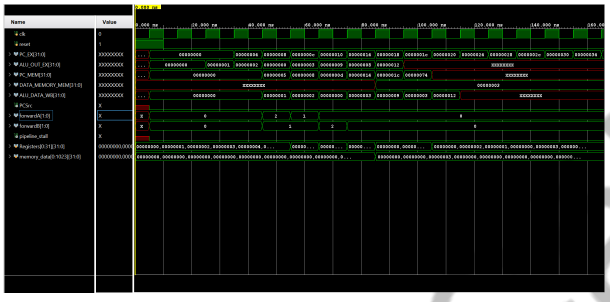

Figure 13: The complete simulation of the RISC V processor

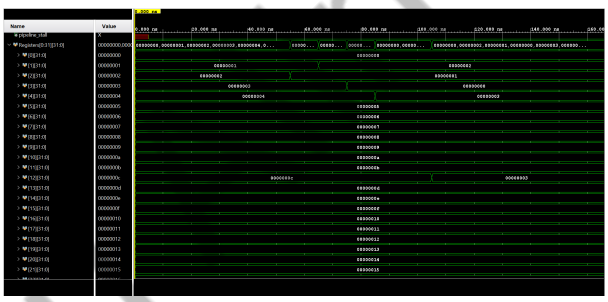

Figure 14: RISC V processor registers

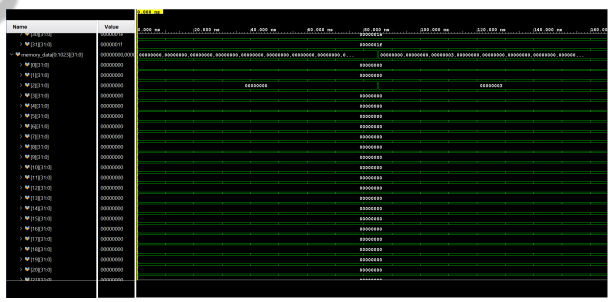

Figure 15: RISC V processor Memory Data

Furthermore, multiple manual tests have been conducted to further challenge the conclusion of the first set of tests, thus confirming the correctness of the provided solution.

4 CONCLUSIONS

In conclusion, this program aims to provide an overview of creating a processor and its development process to understand the capabilities of the RISC V architecture.

The pipeline solution offers a drastic reduction of the execution time compared to a sequential solution, but it comes along with compromises, such as Data hazards with the most commonly met being the Read after Write hazards that can be resolved using different methods such as forwarding done by the Forwarding Unit, and stalling the pipeline which is done by the Hazard Detection Unit.

Bibliography

[1] CN 2 team @ Polytechnic University of Bucharest. Risc v - i. `https://archive.curs.upb.ro/2021/mod/resource/view.php?id=80393`. Latest access: 15.09.2024.

[2] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017. Latest access: 15.09.2024.

[3] CN 2 team @ Polytechnic University of Bucharest. Lucrarea 3. `https://archive.curs.upb.ro/2021/pluginfile.php/432517/mod_folder/content/0/Lucrarea%203.pdf?forcedownload=1`. Latest access: 15.09.2024.