Princess Sumaya University for Technology

King Abdullah II Faculty of Engineering

Computer Engineering Department

Princess Sumaya University for Technology جامعة الأميرة سميّة للتكنولوجيا

5-Stage Pipelined RISC-V Processor

Computer Architecture Final Project

Authors:

Mahmoud Abu-Qtiesh	20210383	Computer Engineering
Mohammad AbdelAziz	20200168	Computer Engineering

Abstract

This report delineates the design, simulation, and assessment of a 5-stage pipelined RISC-V processor, epitomizing advancements in processor architecture. The processor architecture, divided into distinct stages - IF, ID, EXE, MEM, and WB - capitalizes on pipelining principles to enhance execution throughput. Each stage's intricate design, including control units, ALU operations, memory handling, hazard detection, and exception handling, is meticulously detailed. Notably, the implementation of forwarding logic, hazard detection, and branch control mechanisms underscores the project's complexity. Performance evaluation, based on simulated tests and benchmarks, showcases a notable 5x speedup compared to a non-pipelined single-cycle processor. However, challenges posed by hazards and branch misprediction impact the attained speedup. The successful validation of the processor's functionality across diverse benchmarks highlights its efficacy, albeit emphasizing the need for refining hazard handling for optimal performance. This project underscores the advancements made in pipelined architectures while acknowledging the ongoing demand for nuanced optimizations in managing complex instruction streams.

Table of Contents

1 Introduction 2
2 Processor Design 3
2.1 IF Stage 3
2.2 ID Stage 4
2.3 EXE Stage 5
2.4 MEM Stage 6
2.5 WB Stage 7
2.6 Forwarding Unit 8
2.7 Hazard Detection Unit 8
2.8 Exception Handling Unit 9
2.9 Branch Controller 9
2.10 Top Level Module 9
3 Design Analysis 10
4 Performance Analysis and Comparison 10
5 Tests and Runs 11
6 Conclusion 12

1 INTRODUCTION

The relentless pursuit of processor efficiency and performance has led to the development of intricate architectures, among which the 5-stage pipelined RISC-V processor represents a pinnacle of streamlined design and optimal execution throughput. This report serves as a comprehensive exposition detailing the meticulous design, simulation, and verification processes involved in crafting a 5-stage pipelined RISC-V processor, meticulously engineered with five distinct modules corresponding to each pipeline stage: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory (MEM), and Write-back (WB).

At the core of this processor architecture lies 5 distinct modules, each handling specific stages of instruction execution. The processor operates on the principles of pipelining, dividing the instruction execution into five sequential stages. Between each consecutive stage, dedicated registers act as inter-stage buffers, facilitating a smooth transfer of information and outlining the discrete boundaries of the pipeline stages.

The Instruction Fetch (IF) stage initiates the pipeline by fetching instructions from memory based on the program counter (PC). These instructions are then passed onto the Instruction Decode (ID) stage, where they are decoded into control signals and operands in addition branch determination occurs here but it is only used in the following stage. The decoded instructions progress to the Execute (EXE) stage, where arithmetic or logic operations are performed, and addresses are calculated for load and store operations and branch results are sent to the IF.

Following the EXE stage, the instructions enter the Memory (MEM) stage, where data memory operations such as loads and stores take place. Finally, in the Write-back (WB) stage, the results of operations are written back to the registers.

The strategic placement of registers between each pipeline stage serves as latches, effectively separating and synchronizing the flow of instructions. These registers mitigate data hazards and maintain coherence between the different stages, ensuring proper sequencing and preventing inter-stage data corruption.

In addition, we wrote an assembler using C++, a tool made specifically for this project, it automates the translation of RISC-V assembly code to hex code, this tool minimizes the need for manual translation, thereby accelerating the overall development cycle and ensuring accuracy in the encoded instructions fed into the processor.

2 Processor Design

2.1 IF Stage

The IF stage consists of 2 main components the Program Counter register (PC), the 32K x 1 Instruction ROM, and additional circuits used to determine the PC value.

The inputs into the IF stage are addresses from potential branches and jumps in addition to 3 signals from the control unit in the ID stage, these aforementioned control signals are used by the PC Control unit to determine the select lines for the PC MUX which also has PC+4 as input calculated by the adder.

Once the PC is calculated it is read by the instruction ROM at the negative edge and then written onto the IF/ID.Reg alongside the PC.

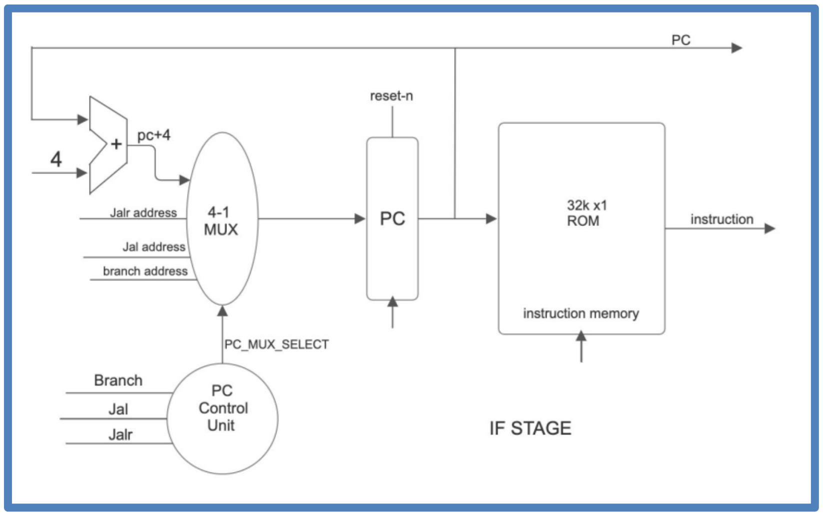


Figure 1: IF Stage Block Diagram

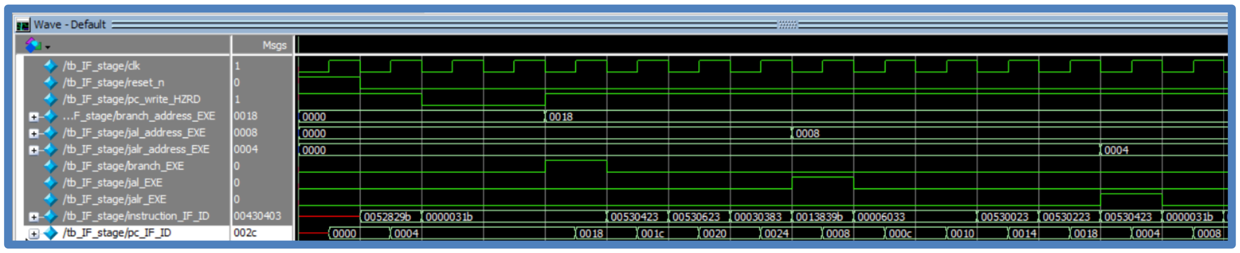

Wave - Default
Msgs
/tb_IF_stage/clk 1
/tb_IF_stage/reset_n 0
/tb_IF_stage/pc_write_HZRD 1
...F_stage/branch_address_EXE 0018
/tb_IF_stage/jal_address_EXE 0008
/tb_IF_stage/jalr_address_EXE 0004
/tb_IF_stage/branch_EXE 0
/tb_IF_stage/jal_EXE 0
/tb_IF_stage/jalr_EXE 0
/tb_IF_stage/instruction_IF_ID 00430403
/tb_IF_stage/pc_IF_ID 002c
0000 0018
0000 0008
0000 0004
0052829b 0000031b 00530423 00530623 00030383 0013839b 00006033 00530023 00530223 00530423 0000031b
0000 0004 0018 001c 0020 0024 0008 000c 0010 0014 0018 0004 0008

Figure 2: tb_IF_stage Waveform

2.2 ID STAGE

The ID Stage consists of 7 modules, most notably the register file which consists of 32 registers each of size 32 bits, in addition to an immediate generator which calculates the immediate for I-type instruction, S-instruction and so on. In addition there is a control unit which using the OpCode and funct-3 as inputs determines the control signals for the remainder of the instruction’s execution in the processor.
Lastly, there are 4 modules used for branching, the comparator and 2 adders for the address in addition to a shift_left_by_1 unit used for branch instructions as they have different addressing modes compared to the rest of the instructions.

For the register file, the rs1 and rs2 are inputs from the IF/ID.Reg used to find the operands of the instruction however, rd, write data and RegWrite are inputs from the WB stage.

Once R[rs1] and R[rs2] are read from the register file they are used by the comparator alongside the control signals BEQ and BNE and then the comparator determines if the branch condition is met. These results alongside the branch-target and jalr target are sent to the ID/EXE.Reg but from there they won't be sent to the EXE stage but rather the IF stage.

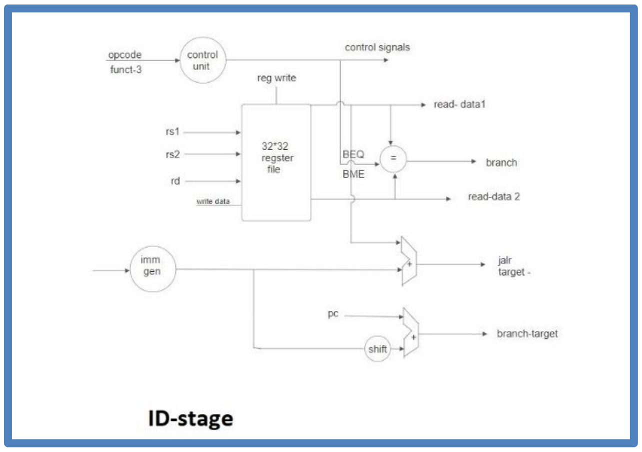
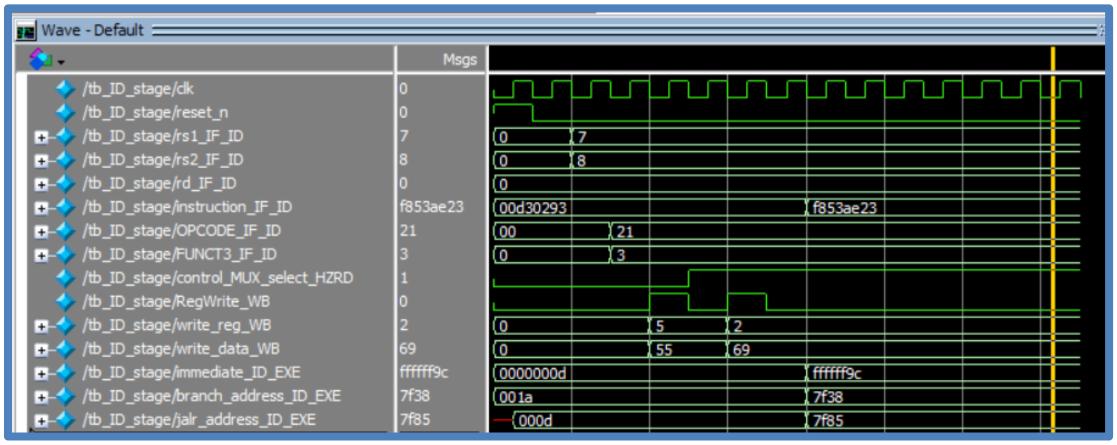


Figure 3: ID Stage Block Diagram



	Msgs
/tb_ID_stage/clk	0
/tb_ID_stage/reset_n	0
/tb_ID_stage/rs1_IF_ID	7
/tb_ID_stage/rs2_IF_ID	8
/tb_ID_stage/rd_IF_ID	0
/tb_ID_stage/instruction_IF_ID	f853ae23
/tb_ID_stage/OPCODE_IF_ID	21
/tb_ID_stage/FUNCT3_IF_ID	3
/tb_ID_stage/control_MUX_select_HZRD	1
/tb_ID_stage/RegWrite_WB	0
/tb_ID_stage/write_reg_WB	2
/tb_ID_stage/write_data_WB	69
/tb_ID_stage/immediate_ID_EXE	ffffff9c
/tb_ID_stage/branch_address_ID_EXE	7f38
/tb_ID_stage/jalr_address_ID_EXE	7f85

Figure 4: tb_ID_stage Waveform

2.3 EXE STAGE

Other than the MUX's the EXE stage consists of only an ALU which has 3 inputs, the 32-it operands R[rs1] and either R[rs2] or the immediate (this is determined by the ALUSrc from the control unit) and the 4-bit ALUOp determined by the control unit.

We must also mention the 2 3-1 MUXs located before the ALU. The purpose of these MUXs is to allow for forwarding logic, the inputs to these MUXs are the operands from the current cycle, ALU Result from the previous instruction and the Write Data of the instruction before that. In tandem with the ForwardA and ForwardB signals from the forwarding unit, this ensures there are now Read-After-Write (RAW) hazards.

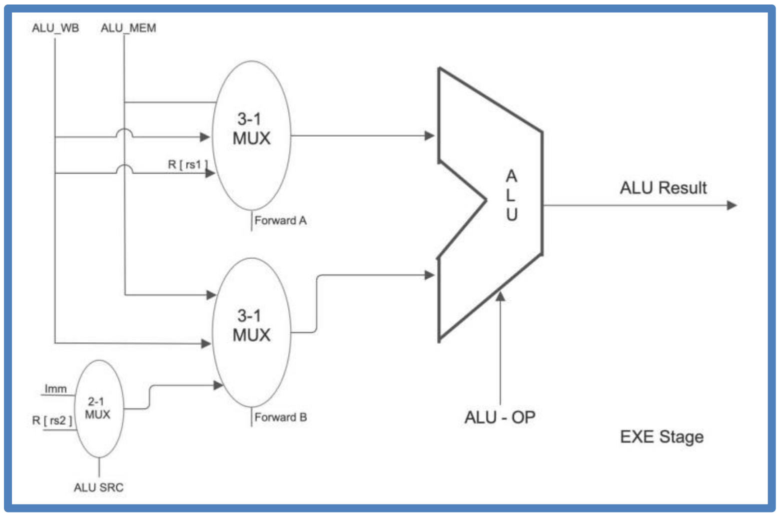
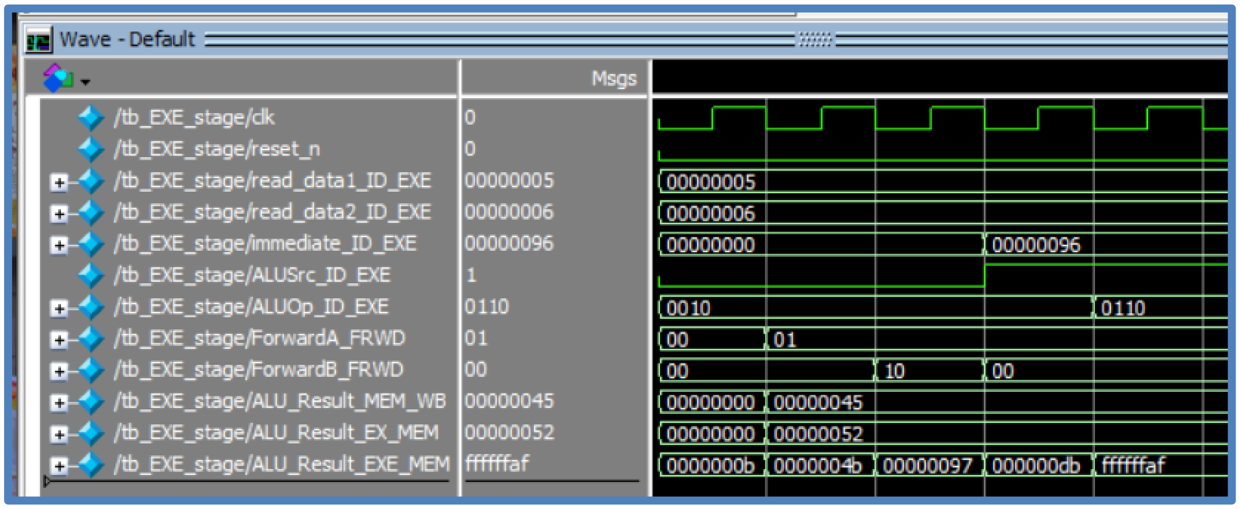


Figure 5: EXE Stage Block Diagram



```
/tb_EXE_stage/clk                 0
/tb_EXE_stage/reset_n             0
/tb_EXE_stage/read_data1_ID_EXE   00000005   00000005
/tb_EXE_stage/read_data2_ID_EXE   00000006   00000006
/tb_EXE_stage/immediate_ID_EXE    00000096   00000000                        00000096
/tb_EXE_stage/ALUSrc_ID_EXE       1
/tb_EXE_stage/ALUOp_ID_EXE        0110       0010                                       0110
/tb_EXE_stage/ForwardA_FRWD       01         00        01
/tb_EXE_stage/ForwardB_FRWD       00         00                   10         00
/tb_EXE_stage/ALU_Result_MEM_WB   00000045   00000000  00000045
/tb_EXE_stage/ALU_Result_EX_MEM   00000052   00000000  00000052
/tb_EXE_stage/ALU_Result_EXE_MEM  ffffffaf   0000000b  0000004b  00000097  000000db  ffffffaf
```

Figure 6: tb_EXE_stage Waveform

2.4 MEM Stage

The MEM stage is where Load/Store instructions take place using the address calculated from the EXE stage, either write data is written onto M[address] or read data is set to M[address], this is decided by MemWrite and MemRead.

It must be noted that MemWrite and MemRead are not 1-bit control signals as we have 2 load instructions and 2 store instructions each with a different size. So MemWrite and MemRead are 4-bits (e.g. in the case of SB MemWrite = 4’b0001 and in case of SW MemWrite = 4’b1111).

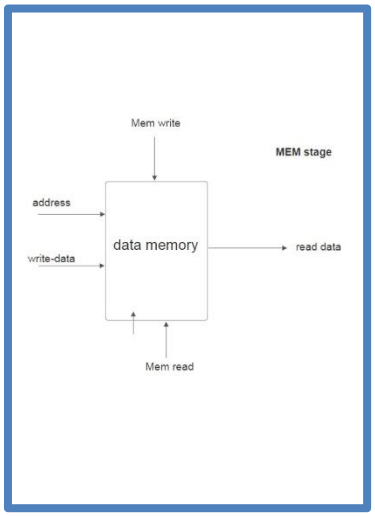
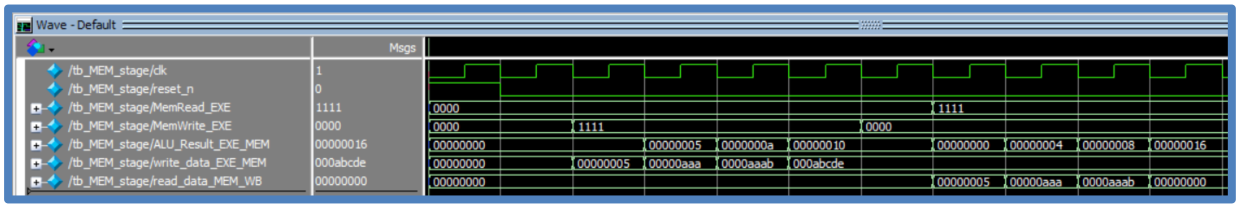
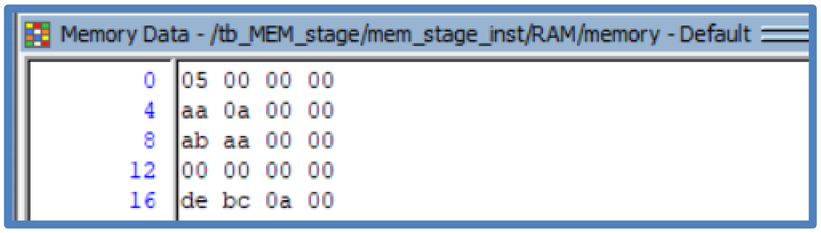


Figure 7: MEM Stage Block Diagram



```
/tb_MEM_stage/clk                1
/tb_MEM_stage/reset_n            0
/tb_MEM_stage/MemRead_EXE        1111      0000                                                1111
/tb_MEM_stage/MemWrite_EXE       0000      0000      1111                            0000
/tb_MEM_stage/ALU_Result_EXE_MEM 00000016  00000000            00000005  0000000a  00000010      00000000  00000004  00000008  00000016
/tb_MEM_stage/write_data_EXE_MEM 000abcde  00000000  00000005  00000aaa  0000aaab  000abcde
/tb_MEM_stage/read_data_MEM_WB   00000000  00000000                                              00000005  00000aaa  0000aaab  00000000
```

Figure 8: tb_MEM_stage Waveform



Memory Data - /tb_MEM_stage/mem_stage_inst/RAM/memory - Default

```
 0  05 00 00 00
 4  aa 0a 00 00
 8  ab aa 00 00
12  00 00 00 00
16  de bc 0a 00
```

Figure 9: tb_MEM_stage RAM contents

2.5 WB Stage

This is the final stage of an instruction's lifetime in the processor, here we have only 1 module a 3-1 MUX with PC, read data and ALU result as inputs in addition the MemToReg as the select line which is determined by the control unit, once write data is determined it is sent to the ID stage alongside the WriteReg and RegWrite (which is 1 in the case of Load, Add, Sub, …).

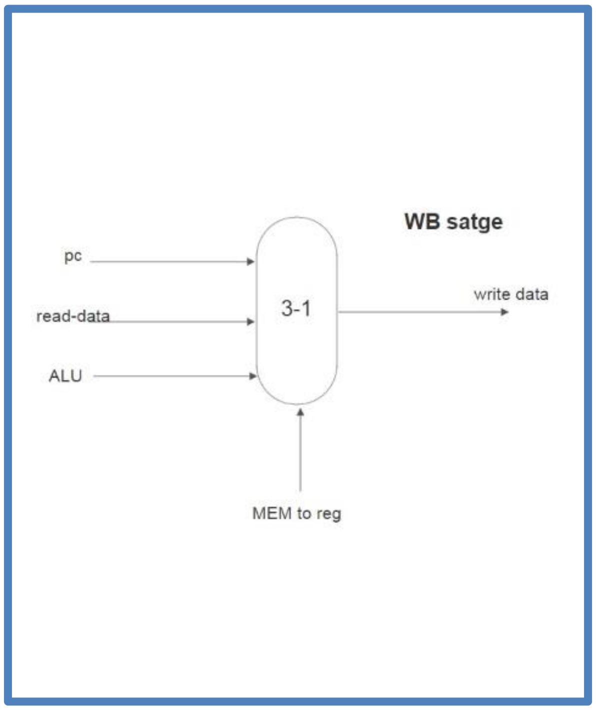
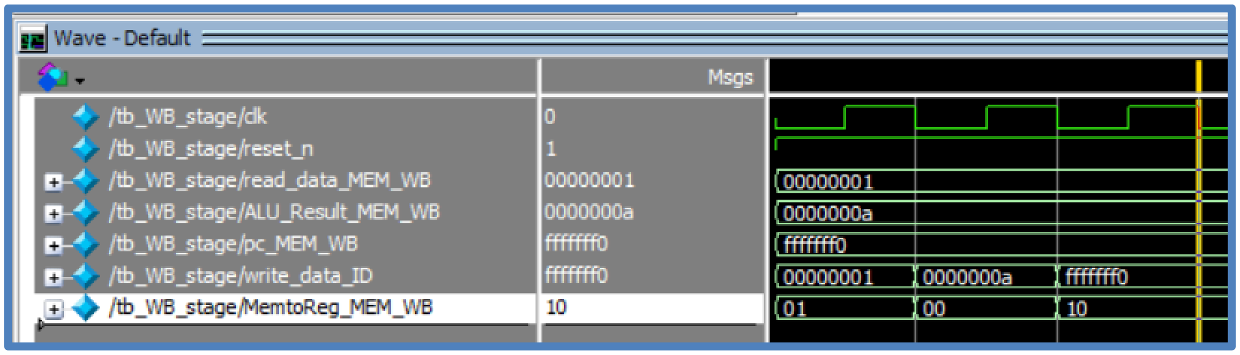


Figure 10: WB Stage Block Diagram



```
/tb_WB_stage/clk                0
/tb_WB_stage/reset_n            1
/tb_WB_stage/read_data_MEM_WB   00000001   00000001
/tb_WB_stage/ALU_Result_MEM_WB  0000000a   0000000a
/tb_WB_stage/pc_MEM_WB          fffffff0   fffffff0
/tb_WB_stage/write_data_ID      fffffff0   00000001   0000000a   fffffff0
/tb_WB_stage/MemtoReg_MEM_WB    10         01         00         10
```

Figure 11: tb_WB_stage Waveform

2.6 FORWARDING UNIT

Using the forwarding paths in the diagram below we perform some logic on the inputs and determine the need to forward, the forwarding unit send 2 outputs to the EXE stage ForwardA and ForwardB.

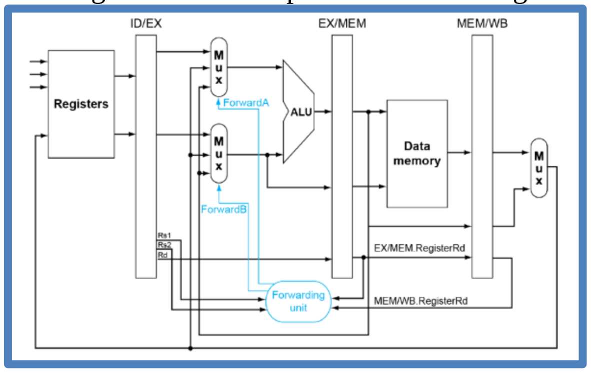
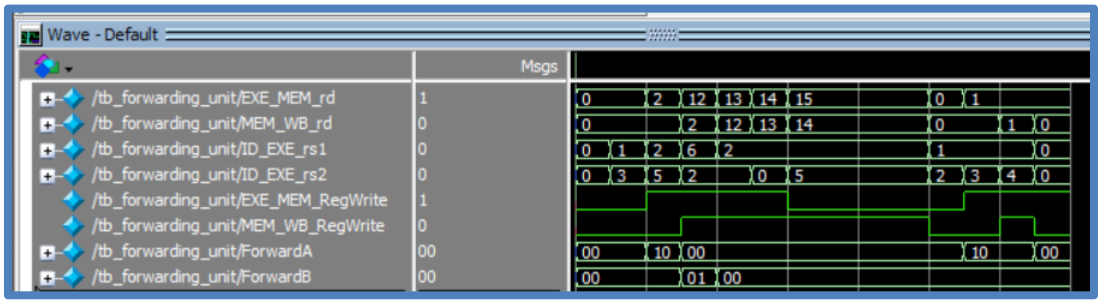


Figure 12: Forwarding Paths



	Msgs
/tb_forwarding_unit/EXE_MEM_rd	1
/tb_forwarding_unit/MEM_WB_rd	0
/tb_forwarding_unit/ID_EXE_rs1	0
/tb_forwarding_unit/ID_EXE_rs2	0
/tb_forwarding_unit/EXE_MEM_RegWrite	1
/tb_forwarding_unit/MEM_WB_RegWrite	0
/tb_forwarding_unit/ForwardA	00
/tb_forwarding_unit/ForwardB	00

Figure 13: tb_forwarding_unit Waveform

2.7 HAZARD DETECTION UNIT

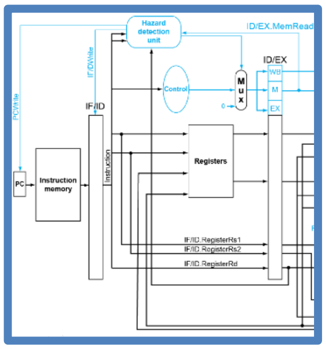


Figure 14: Hazard Detection Datapath

The hazard detection unit (HDU) is used to detect any Load-Use hazards and to stall the pipeline (insert bubble) is detected.
This is done using simple if-statements, if a load-use hazard is detected the hazard detection will flush the instruction currently in the ID stage by setting all control signals to 0. In addition it will stall the instruction in the IF stage by setting PCWrite to 0 and IF/ID.RegWrite to 0, meaning the instruction that was in the ID stage will re execute but this time there will be a 1-cycle difference between it and the previous load instruction.

2.8 Exception Handling Unit

The exception handling unit (EHU) consists of 2 register the SEPC and the SCAUSE which determine the exception causing instruction and the SCAUSE determines the exception cause.

The EHU handles the following cases:

Case	Example	Code
Incorrect Load	MemRead == 4’b0110;	0
Incorrect Store	MemWrite == 4’b1001;	1
Incorrect ALU operation	ALUOp == 4’b1100;	2
Out-of-Range Branch Fetch	branch_address == 15’h200c;	3
Incorrect Op Code	OPCODE == 7’h34;	4

For each case the SCAUSE holds the code and the reason (e.g. it will store the OpCode if the code is 4, or it will store the branch_address if the code is 3).

2.9 Branch Controller

This simple modules is used to flush the last 2 instructions in the case of a branch misprediction. The prediction model dictated by the design requirements was always assume branch not taken, so whenever we branch to another instruction the last 2 fetched cycles must be flushed.

This is done by adding an input called flush to the IF/ID.Reg and the ID/EXE.Reg which is sent from the branch controller, if the branch is to be takes the flush input will be set to 1 and the instructions stored in the IF/ID.Reg and ID/EXE.Reg will be erased and the correct instruction will be fetched.

2.10 Top Level Module

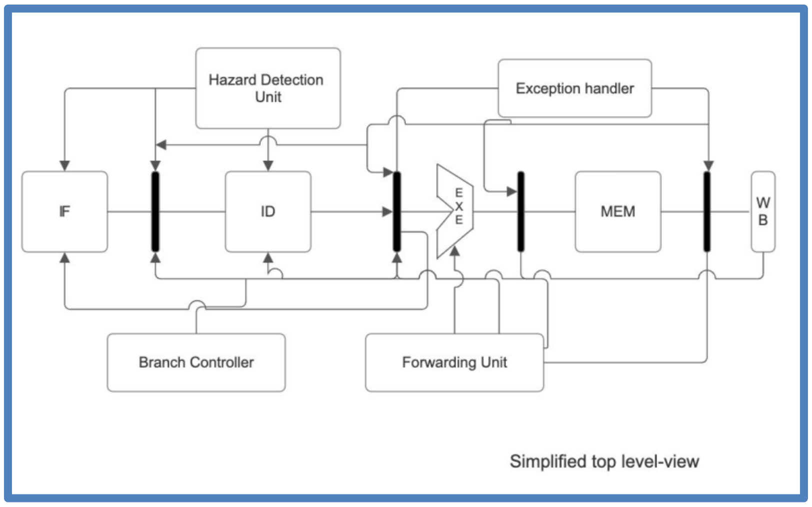


Figure 15: Top Level Block Diagram

3 Design Analysis

During this part of the report we will discuss issues we faced during the design process and the important decisions we made to address them.

As the design requirements stated branch determination must occur in the EXE stage however as the EXE stage is made up of only combinational logic this caused an issue with synchronization, so we decided to determine whether or not a branch will occur in the ID stage and copy the result to the ID/EXE.Reg and then extract it from there to the branch controller which sends signals to the IF stage to update the PC in case we need to branch, using the synchronous registers we ensure that everything is done in order and in case of a mis-prediction we only have to flush 2 cycles.

Staying on the topic of flushing. We faced an issue with how to flush the previous 2 instructions in the case of a branch misprediction, we solved this by adding an input to the stage separating registers called (flush). This input is hardwired to GND for the EXE/MEM.Reg and the MEM/WB.Reg as once an instruction enters the EXE stage we are sure it is the correct instruction, however for the ID/IF.Reg and ID/EXE.Reg the flush input is received from the branch controller which in the case of a mis-prediction sets flush to 1 and the instructions are stored in the IF/ID.Reg and ID/EXE.Reg are erased, the instructions fetched before them continue their execution and the correct instruction is then fetched leading to a 2-cycle delay in the case of a misprediction.

4 Performance Analysis and Comparison

The improvement of our design compared to a non-pipelined single-cycle processor can be calculated using the following equation:

$SpeedUp = \lim_{n\to\infty} \frac{n\cdot t_n}{(5+n-1)\cdot t_p} = \frac{t_n}{t_p} = \frac{5\cdot t_p}{t_p} = 5$, and as we can see the speed-up ration between the pipelined and non-pipelined architecture is 5, meaning our pipelined design is 5x faster than the standard RISC-V single-cycle processor.

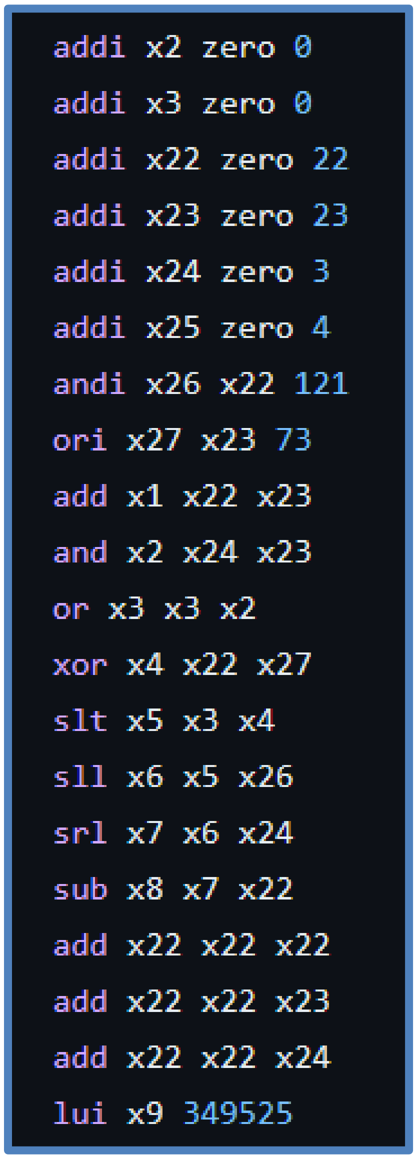
Nonetheless, our design is much more complex due to the vast number of hazards involved in parallel execution of instructions, such as Load-Use hazards and Control-Hazards due to branch misprediction, these issues required advanced circuits to handles them, in addition they might reduce the calculated speedup of 5, such as in the case of test3.asm from the GitHub repository, the code consists of 1 while loop and 2047 iteration, on a single cycle processor the amount of cycles will be (8189) however on our design it took (12,289) cycles due to branch misprediction. The reduced speed up can be calculated using the following equation:

$SpeedUp' = \frac{8189\cdot 50\,ps}{12289\cdot 10ps} = 3.33$, this shows that our design is still faster than the non-pipelined architecture however, not as fast as the ideal case of 5.

5 Tests and Runs

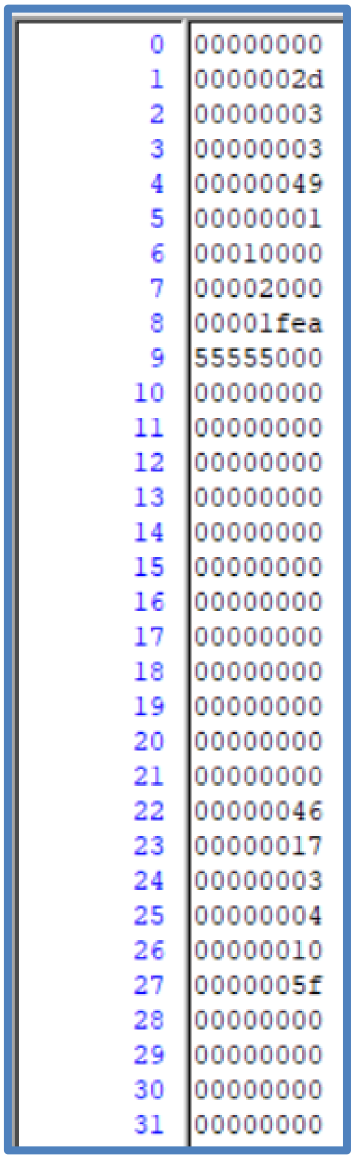
We will now test the top-level module on 4 different benchmarks [1] , which encompass the required ISA and handle many cases, we will also compare the outputs of our test to the RARS simulator, and our outputs should be the same:

5.1 Arithmetic Operations



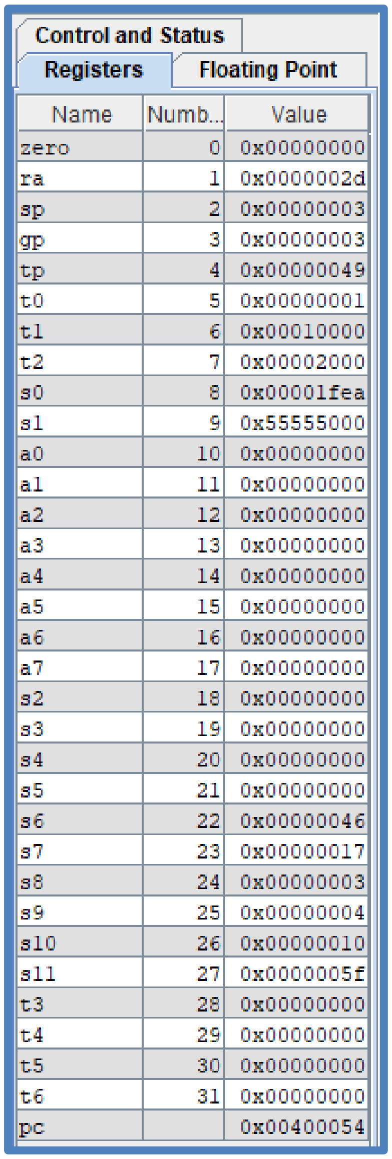
```
addi x2 zero 0
addi x3 zero 0
addi x22 zero 22
addi x23 zero 23
addi x24 zero 3
addi x25 zero 4
andi x26 x22 121
ori x27 x23 73
add x1 x22 x23
and x2 x24 x23
or x3 x3 x2
xor x4 x22 x27
slt x5 x3 x4
sll x6 x5 x26
srl x7 x6 x24
sub x8 x7 x22
add x22 x22 x22
add x22 x22 x23
add x22 x22 x24
lui x9 349525
```

Figure 16: test1.asm



```
 0  00000000
 1  0000002d
 2  00000003
 3  00000003
 4  00000049
 5  00000001
 6  00010000
 7  00002000
 8  00001fea
 9  55555000
10  00000000
11  00000000
12  00000000
13  00000000
14  00000000
15  00000000
16  00000000
17  00000000
18  00000000
19  00000000
20  00000000
21  00000000
22  00000046
23  00000017
24  00000003
25  00000004
26  00000010
27  0000005f
28  00000000
29  00000000
30  00000000
31  00000000
```

Figure 17: test1 Actual Results

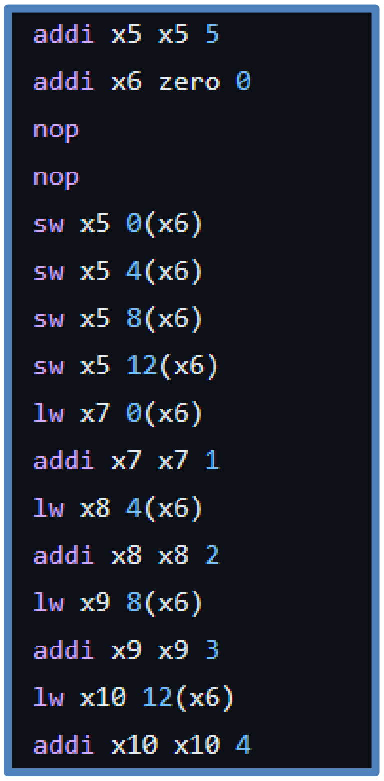


Name	Numb...	Value
zero	0	0x00000000
ra	1	0x0000002d
sp	2	0x00000003
gp	3	0x00000003
tp	4	0x00000049
t0	5	0x00000001
t1	6	0x00010000
t2	7	0x00002000
s0	8	0x00001fea
s1	9	0x55555000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000046
s7	23	0x00000017
s8	24	0x00000003
s9	25	0x00000004
s10	26	0x00000010
s11	27	0x0000005f
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400054

Figure 18: test1 Expected Resutls

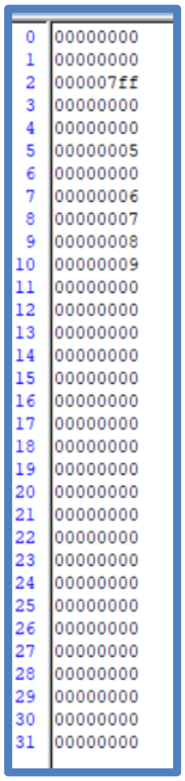
Our design passed the arithmetic test which tests all modes of operation of the ALU in addition to the forwarding capability of our design as lines 17-19 contain a double-data-hazard.

5.2 Load and Store



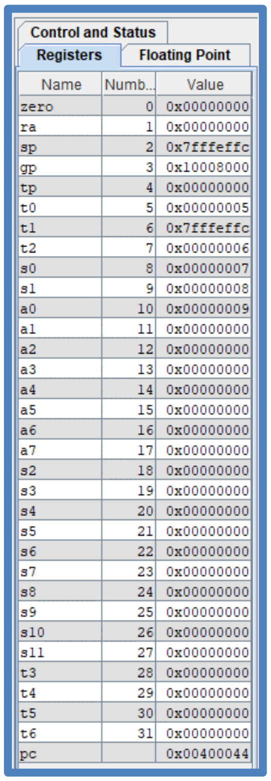
```
addi x5 x5 5
addi x6 zero 0
nop
nop
sw x5 0(x6)
sw x5 4(x6)
sw x5 8(x6)
sw x5 12(x6)
lw x7 0(x6)
addi x7 x7 1
lw x8 4(x6)
addi x8 x8 2
lw x9 8(x6)
addi x9 x9 3
lw x10 12(x6)
addi x10 x10 4
```

Figure 19: test2.asm



```
0   00000000
1   00000000
2   000007ff
3   00000000
4   00000000
5   00000005
6   00000000
7   00000006
8   00000007
9   00000008
10  00000009
11  00000000
12  00000000
13  00000000
14  00000000
15  00000000
16  00000000
17  00000000
18  00000000
19  00000000
20  00000000
21  00000000
22  00000000
23  00000000
24  00000000
25  00000000
26  00000000
27  00000000
28  00000000
29  00000000
30  00000000
31  00000000
```

Figure 20: test2.asm Actual Results



Name	Numb...	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000005
t1	6	0x7fffeffc
t2	7	0x00000006
s0	8	0x00000007
s1	9	0x00000008
a0	10	0x00000009
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400044

Figure 21: test2.asm Expected Results

We had to modify the asm file to work on the RARS simulator as the portion of memory we used in our ModelSim simulation is reserved in RARS which is will explain why the value of x6 is different in Figure (20) and Figure (21).

Other than that our design passed this test which featured accesses to the memory, in addition to Load-Use hazard which occur after every load.

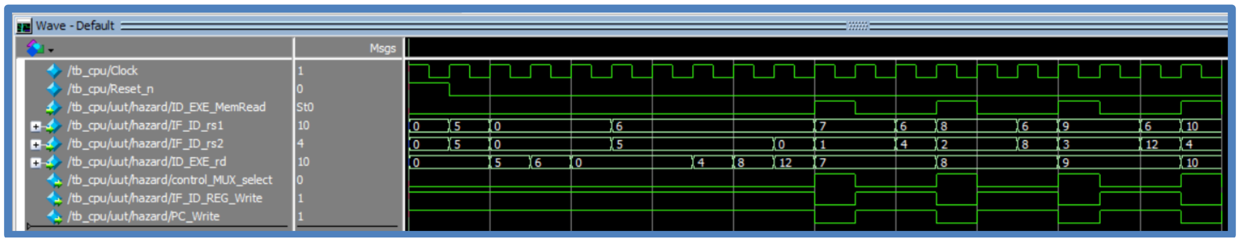
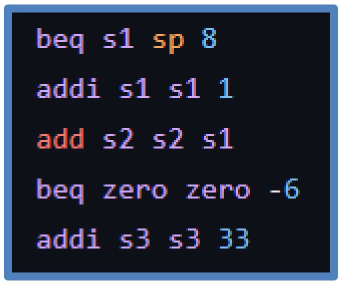


Figure 22: test2.asm Hazard Detection Waveform

The waveform above shows how the HDU deals with Load-Use hazards by stalling the pipeline for 1-cycle.

5.3 Branching and Looping

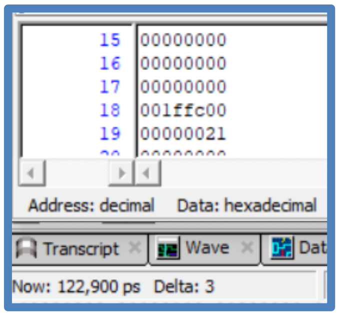
In the following test we will test our design capabilities to branch to other instructions in the instruction ROM, we will also test the flushing capabilities of the processor.



```
beq s1 sp 8
addi s1 s1 1
add s2 s2 s1
beq zero zero -6
addi s3 s3 33
```

Figure 23: test3.asm

The asm file above is used to calculate the summation $\sum_{i=0}^{sp} i$, which in our case sp = 2047, so the value of s2 should equal $\sum_{i=0}^{2047} i = \frac{(2047)(2048)}{2} =$ 0x001ffc00.

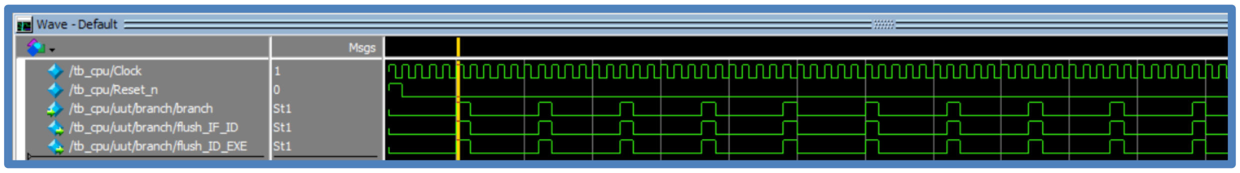


```
15 00000000
16 00000000
17 00000000
18 001ffc00
19 00000021
```

```
Now: 122,900 ps  Delta: 3
```

Figure 24: test3.asm Actual Results

We can see that at t = 122,900 ps (which is exactly the same out of time we calculated earlier when discussing performance, but with the addition of 1 extra cycle to allow the CPU to reset), the value of 0x21 was written onto register s3 and the value of register s2 is 0x001ffc00.



```
/tb_cpu/Clock                   1
/tb_cpu/Reset_n                 0
/tb_cpu/uut/branch/branch       St1
/tb_cpu/uut/branch/flush_IF_ID  St1
/tb_cpu/uut/branch/flush_ID_EXE St1
```

Figure 25: test3.asm Branch Controller Waveform

The waveform above shows that whenever the branch signal is high the branch controller flushes the IF/ID.Reg and the ID/EXE.Reg and the difference between every branch is 6 cycles, 4 cycles for the instructions in the loop body and another 2 for the flushes.

[1] There is another Test on the GitHub repository however due to page limitation we were not able to show its results here.

6 CONCLUSION

The development and analysis of the 5-stage pipelined RISC-V processor showcased a significant leap in processor architecture. Implementing distinct stages for instruction execution improved throughput, evident in the 5x speedup compared to a non-pipelined single-cycle processor. However, the complexity surged due to handling hazards like Load-Use and branch misprediction, impacting the ideal speedup. The design demonstrated successful functionality across various benchmarks, validating arithmetic operations, memory access, and branching capabilities. Challenges in test scenarios were addressed, highlighting the effectiveness of hazard detection and control mechanisms. Although the design excels in performance, the intricate handling of hazards denotes areas for future refinement to maximize efficiency and minimize the impact on speedup. Overall, this project underscores the effectiveness of pipelined architectures but emphasizes the ongoing need for nuanced optimization in handling complex instruction streams.

7 REFERENCES

[1] D. A. Patterson and J. L. Hennessy, Computer Organization and Design, 4th edition.

[2] S. Kadam and S. Mali, "Design of Instruction Fetch Unit and ALU for Pipelined RISC Processor," *International Journal Of Engineering And Computer Science*, p. 4, 2016.