



**Faculty of Engineering & Technology**

**Electrical & Computer Engineering Department**

**ENCS4370, Computer Architecture**

**Project No. 2**

**Design and Verification of a Simple Pipelined RISC Processor in  
Verilog**

---

**Prepared by:**

Raseel Jafar

**ID:** 1220724

**Section :** 3

Aya Abusnaina

**ID:** 1221414

**Section :** 1

Ahmad Sous

**ID:** 1221371

**Section :** 1

**Instructor:**

Dr. Aziz Qaroush

**Date:**

10<sup>th</sup> June , 2025

## Abstract

This project involves designing and verifying a 32-bit pipelined RISC processor in Verilog. The processor features a 5-stage pipeline (fetch, decode, execute, memory access, write-back), 16 general-purpose registers, and separate instruction/data memories. The ISA includes arithmetic, logical, load/store, and branch instructions, with a fixed 32-bit instruction format. Key tasks include designing the datapath and control path, implementing the processor in Verilog, and verifying functionality through simulation using test programs. The final deliverables are a well-documented RTL design, testbench, and a comprehensive report covering design choices, verification results, and teamwork

Table 1: Check Table

Team Member Name	Team Member ID	Contributions							
Raseel Jafar	1220724	33.3%							
Aya Abusnaina	1221414	33.3%							
Ahmad Sous	1221371	33.3%							
*explained with details on Teamwork Section									
<b>Processor Implementation (Tick One)</b>									
Single Cycle							-		
Multi Cycle							-		
Pipelined							✓		
<b>In case of pipeline implementation (Tick the correct answer)</b>									
		<b>Implemented in RTL Code?</b>		<b>Verified and Correctly Worked?</b>					
Data hazards detection		✓		✓					
Control hazards detection		✓		✓					
Structural hazards detection		✓		✓					
Forwarding		✓		✓					
Stalling		✓		✓					
<b>Tick the correct answer</b>									
Instruction	Did you implement this instruction in the RTL?		Did you write the verification code for this instruction?		Did the instruction Work perfectly when it has been verified?				
	Yes	No	Yes	No	Yes	No			
OR Rd, Rs, Rt	✓	-	✓	-	✓	-			
ADD Rd, Rs, Rt	✓	-	✓	-	✓	-			
SUB Rd, Rs, Rt	✓	-	✓	-	✓	-			
CMP Rd, Rs, Rt	✓	-	✓	-	✓	-			
ORI Rd, Rs, Imm	✓	-	✓	-	✓	-			
ADDI Rd, Rs, Imm	✓	-	✓	-	✓	-			
LW Rd, Imm(Rs)	✓	-	✓	-	✓	-			

LDW Rd, Imm(Rs)	✓	-	✓	-	✓	-
SDW Rd, Imm(Rs)	✓	-	✓	-	✓	-
BZ Rs, Label	✓	-	✓	-	✓	-
BGZ Rs, Label	✓	-	✓	-	✓	-
BLZ Rs, Label	✓	-	✓	-	✓	-
JR Rs	✓	-	✓	-	✓	-
J Label	✓	-	✓	-	✓	-
CALL Label	✓	-	✓	-	✓	-
<b>Tick one of the following</b>						
My processor can execute only test programs consisting of one instruction only						-
My processor can execute complete programs (A simulation screenshot must be provided as evidence)						✓

## Table of Contents

<b>Abstract .....</b>	<b>I</b>
<b>Table of figures.....</b>	<b>VI</b>
<b>Table of tables.....</b>	<b>VII</b>
<b>1. Theory .....</b>	<b>1</b>
<b>1.1 Instruction Set Architecture (ISA) :.....</b>	<b>1</b>
<b>Instruction Format:.....</b>	<b>1</b>
<b>1.2 Instruction Categories : .....</b>	<b>2</b>
<b>i) Arithmetic Instructions : .....</b>	<b>2</b>
<b>ii) Logical Instructions : .....</b>	<b>2</b>
<b>iii) Comparison Instruction: .....</b>	<b>2</b>
<b>iv) Memory Access Instructions: .....</b>	<b>3</b>
<b>v) Control Flow Instructions: .....</b>	<b>3</b>
<b>2. Procedure.....</b>	<b>4</b>
<b>2.1 Instructions Format and Corresponding meaning in RTN .....</b>	<b>4</b>
<b>2.2 RTL Design .....</b>	<b>6</b>
<b>2.3 Single Cycle Datapath:.....</b>	<b>10</b>
<b>2.3.1 Instruction Fetch (IF) .....</b>	<b>10</b>
<b>2.2.2 Instruction Decode (ID).....</b>	<b>11</b>
<b>2.2.3 Execution (EX).....</b>	<b>11</b>
<b>2.2.4 Memory Access (MEM).....</b>	<b>12</b>
<b>2.2.5 Write Back (WB) .....</b>	<b>12</b>
<b>2.2.6 Control Unit .....</b>	<b>12</b>
<b>2.4 Pipelining.....</b>	<b>13</b>
<b>2.4.1 Datapath.....</b>	<b>13</b>
<b>2.4.2 Instruction Fetch (IF) .....</b>	<b>14</b>
<b>2.4.3 Instruction Decode (ID) .....</b>	<b>16</b>
<b>2.5 Control Signals .....</b>	<b>23</b>
<b>2.5.1 ALU &amp; Main control unit.....</b>	<b>23</b>
<b>2.5.1 PC Control Unit.....</b>	<b>26</b>
<b>2.5.3 Hazard Control Unit.....</b>	<b>29</b>
<b>3. Testing and Verification.....</b>	<b>31</b>

<b>3.1 Test 1 .....</b>	<b>31</b>
<b>3.1.1 Analysis.....</b>	<b>32</b>
<b>3.1.2 Instruction Execution Profile Report.....</b>	<b>32</b>
<b>3.1.3 Contact of registers.....</b>	<b>33</b>
<b>3.1.4 Waveform .....</b>	<b>33</b>
<b>3.2 Test 2 .....</b>	<b>35</b>
<b>3.2.1 Analysis .....</b>	<b>36</b>
<b>3.2.3 Contact of registers.....</b>	<b>38</b>
<b>3.2.4 Waveform .....</b>	<b>39</b>
<b>3.3 Test 3 .....</b>	<b>40</b>
<b>3.3.1 Analysis .....</b>	<b>41</b>
<b>3.3.2 Instruction Execution Profile Report.....</b>	<b>42</b>
<b>3.3.3 Contact of registers.....</b>	<b>42</b>
<b>3.3.4 Waveform .....</b>	<b>43</b>
<b>4. Conclusion .....</b>	<b>44</b>
<b>5. Teamwork.....</b>	<b>45</b>

## Table of figures

figure 1: single cycle datapath. -----	10
Figure 2: pipelined processor datapath -----	13
figure 3: pipelined processor (IF). -----	14
Figure 4: pipelined processor (ID) -----	16
figure 5: pipelined processor (EX). -----	20
figure 6: pipelined processor (MEM). -----	21
figure 7: pipelined processor (WB). -----	22
Figure 8 : Test 1 -----	31
Figure 9: Instruction Execution Analysis -----	32
Figure 10 : Register values after program execution -----	33
Figure 11 : Test 1 waveform 1 -----	33
Figure 12 : Test 1 waveform 2 -----	33
Figure 13 : Test 1 waveform 3 -----	34
Figure 14 : Test 1 waveform 4 -----	34
Figure 15 : Test 2 -----	35
Figure 16: Instruction Execution Analysis -----	38
Figure 17: Register values after program execution -----	38
Figure 18: Test 2 waveform 1 -----	39
Figure 19: Test 2 waveform 2 -----	39
Figure 20: Test 2 waveform 3 -----	39
Figure 21: Test 2 waveform 4 -----	39
Figure 22: Test 3 -----	40
Figure 23: Instruction Execution Analysis -----	42
Figure 24: Register values after program execution -----	42
Figure 25: Test 2 waveform 1 -----	43
Figure 26: Test 2 waveform 2 -----	43
Figure 27: Test 2 waveform 3 -----	43
figure 28: Team Contribution Distribution. -----	45

## Table of tables

Table 1: Check Table	II
Table 2: Instructions Format with their RTN	4
Table 3: RTL Design Table for all instructions	6
Table 4: ALU & Main control unit signals.	23
Table 5: Boolean Expressions for Hazard Control Unit Signals.	29

## 1. Theory

The processor designed in this project is a 5-stage pipelined RISC processor with a simple, consistent 32-bit instruction format. It follows the RISC design philosophy, emphasizing a small, efficient instruction set with uniform instruction length and separate instruction and data memory. The processor includes 16 general-purpose registers (R0 to R15), where R15 is reserved as the Program Counter (PC), and R14 is typically designated to hold the return address.

### 1.1 Instruction Set Architecture (ISA) :

The Instruction Set Architecture (ISA) defines the functionality that a processor can perform. It serves as the interface between software and hardware, outlining how programs interact with the processor. In this project, the processor is designed with a custom **32-bit** RISC-style ISA that emphasizes simplicity, uniformity, and efficiency.

All instructions in this ISA are **32 bits** wide and conform to a single instruction format. The format includes a **6-bit opcode** field that determines the operation, along with **4-bit fields for destination and source registers**, and a **14-bit immediate** field that can represent constants or address offsets.

This ISA supports a carefully selected subset of operations that are sufficient to implement general-purpose programs. These operations are grouped into categories explained in the section followed.

#### Instruction Format:

Opcode <sup>6</sup>	Rd <sup>4</sup>	Rs <sup>4</sup>	Rt <sup>4</sup>	Imm <sup>14</sup>
---------------------	-----------------	-----------------	-----------------	-------------------

- **Opcode (6 bits):** Specifies the operation to be performed.
- **Rd (4 bits):** Destination register.
- **Rs (4 bits):** First source register.
- **Rt (4 bits):** Second source register.
- **Imm (14 bits):** Immediate value. This value is:
  - ➔ **Sign-extended** for arithmetic and control flow instructions.
  - ➔ **Zero-extended** for logical instructions.

For branch and jump instructions (e.g., BZ, BGZ, BLZ, J, and CLL), the immediate field is interpreted as an offset to be added to the current Program Counter (PC).

The processor supports a subset of instructions categorized into **arithmetic**, **logical**, **memory access**, **comparison**, and **control flow** operations. These instructions form the foundation of typical programs, and the processor is designed to execute them efficiently using pipelining, hazard detection, and forwarding mechanisms.

## **1.2 Instruction Categories :**

### **i) Arithmetic Instructions :**

Arithmetic instructions perform basic mathematical operations between register values or between a register and an immediate constant.

- **ADD:** Adds two register values and stores the result in a destination register.
- **ADDI:** Adds a register value and an immediate constant and stores the result in a destination register.
- **SUB:** Subtracts the second register from the first and stores the result.

These instructions are essential for performing computations such as counters, sums, and index calculations.

### **ii) Logical Instructions :**

Logical instructions perform bitwise operations used in low-level data manipulation.

- **OR:** Performs a bitwise OR between two register values.
- **ORI:** Performs a bitwise OR between a register value and an immediate value.

Logical operations are often used in mask operations, flag setting, or conditional bit manipulation.

### **iii) Comparison Instruction:**

This instruction is used to compare two register values.

- **CMP:** Compares two registers and stores a result indicating whether the first is less than, equal to, or greater than the second.

Comparison results are used for conditional branching and decision-making in programs.

#### iv) Memory Access Instructions:

These instructions are responsible for reading from or writing to memory.

- **LW (Load Word):** Loads a word from memory into a register.
- **SW (Store Word):** Stores a register value into memory.
- **LDW (Load Double Word):** Loads two consecutive words from memory into two consecutive registers. Requires the destination register to be even-numbered.
- **SDW (Store Double Word):** Stores the contents of two consecutive registers into memory in two consecutive locations. Also requires an even-numbered register.

These are critical for accessing data stored outside of the processor, such as arrays or global variables.

#### v) Control Flow Instructions:

These instructions modify the normal sequential flow of execution, allowing for loops, conditionals, and function calls.

- **BZ (Branch if Zero):** Branches to a new address if a register is zero.
- **BGZ (Branch if Greater than Zero):** Branches if a register contains a positive value.
- **BLZ (Branch if Less than Zero):** Branches if a register contains a negative value.
- **JR (Jump Register):** Jumps to an address stored in a register.
- **J (Unconditional Jump):** Jumps to a new instruction address unconditionally.
- **CLL (Call):** Calls a subroutine by jumping to an address and saving the return address in R14.

Control flow instructions enable conditional execution, looping, and modular code execution through function calls.

## 2. Procedure

### 2.1 Instructions Format and Corresponding meaning in RTN

Table 2: Instructions Format with their RTN

Instruction	Meaning	Opcode Value
OR Rd, Rs, Rt	$\text{Reg(Rd)} = \text{Reg(Rs)} \mid \text{Reg(Rt)}$	0
ADD Rd, Rs, Rt	$\text{Reg(Rd)} = \text{Reg(Rs)} + \text{Reg(Rt)}$	1
SUB Rd, Rs, Rt	$\text{Reg(Rd)} = \text{Reg(Rs)} - \text{Reg(Rt)}$	2
CMP Rd, Rs, Rt	$\text{Reg(Rd)} = 0$ if $(\text{Reg(Rs)} == \text{Reg(Rt)})$ $\text{Reg(Rd)} = -1$ if $(\text{Reg(Rs)} < \text{Reg(Rt)})$ $\text{Reg(Rd)} = 1$ if $(\text{Reg(Rs)} > \text{Reg(Rt)})$	3
ORI Rd, Rs, Imm	$\text{Reg(Rd)} = \text{Reg(Rs)} \mid \text{Imm}$	4
ADDI Rd, Rs, Imm	$\text{Reg(Rd)} = \text{Reg(Rs)} + \text{Imm}$	5
LW Rd, Imm(Rs)	$\text{Reg(Rd)} = \text{Mem}(\text{Reg(Rs)} + \text{Imm})$	6
SW Rd, Imm(Rs)	$\text{Mem}(\text{Reg(Rs)} + \text{Imm}) = \text{Reg(Rd)}$	7
LDW Rd, Imm(Rs)	In the first clock cycle: <ul style="list-style-type: none"><li><math>\text{Reg(Rd)} = \text{Mem}(\text{Reg(Rs)} + \text{Imm})</math> In</li></ul> the second clock cycle: <ul style="list-style-type: none"><li><math>\text{Reg(Rd} + 1) = \text{Mem}(\text{Reg(Rs)} + \text{Imm} + 1)</math></li><li>LDW (load double word) loads two consecutive words from memory to two consecutive destination registers in two consecutive clock cycles.</li><li>The destination register number in the instruction must be an even number.</li><li>If the processor detects an LDW instruction with an odd destination register number, it will throw an exception</li><li>When the processor detects an LDW instruction in the decode unit, it will not bring a new instruction in the next clock cycle, because in the next cycle, the processor will</li></ul>	8

	handle the second word	
SDW Rd, Imm(Rs)	<p>In the first clock cycle:</p> <ul style="list-style-type: none"> <li>• <math>\text{Reg(Rd)} \rightarrow \text{Mem(Reg(Rs) + Imm)}</math></li> </ul> <p>In the second clock cycle:</p> <ul style="list-style-type: none"> <li>• <math>\text{Reg(Rd + 1)} \rightarrow \text{Mem(Reg(Rs) + Imm + 1)}</math></li> <li>• SDW (store double word) stores the content of two consecutive source registers into two consecutive words in memory in two consecutive clock cycles.</li> <li>• The source register number in the instruction must be an even number.</li> <li>• If the processor detects an SDW instruction with an odd source register number, it will throw an exception</li> <li>• When the processor detects an SDW instruction in the decode unit, it will not bring a new instruction in the next cycle, because in the next cycle, the processor will handle the second word</li> </ul>	
BZ Rs, Label	<p>if (<math>\text{Reg(Rs)} == 0</math>)</p> <p>Next PC = branch target else Next PC = PC + 1</p>	10
BGZ Rs, Label	<p>if (<math>\text{Reg(Rs)} &gt; 0</math>)</p> <p>Next PC = branch target</p> <p>else Next PC = PC + 1</p>	11
BLZ Rs, Label	<p>if (<math>\text{Reg(Rs)} &lt; 0</math>)</p> <p>Next PC = branch target else Next PC = PC + 1</p>	12
JR Rs	Jump to the target address stored in the register Rs	13

J Label	Unconditionally jump to the target address specified by the label	14
CLL Label	Jump to the function at the specified label and store the return address in register R14	15

## 2.2 RTL Design

Table 3: RTL Design Table for all instructions

Instruction	Stage	RTL
OR Rd, Rs, Rt	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] AluOperand2 = Reg[inst[17:14]] DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	AluResult = AluOperand1    AluOperand2
	MEM(Memory Access)	Not required
	WB (Write Back)	RegFile[DestinationAddress] = AluResult
ADD Rd, Rs, Rt	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] AluOperand2 = Reg[inst[17:14]] DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	AluResult = AluOperand1 + AluOperand2
	MEM(Memory Access)	Not required
	WB (Write Back)	RegFile[DestinationAddress] = AluResult
SUB Rd, Rs, Rt	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] AluOperand2 = Reg[inst[17:14]] DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	AluResult = AluOperand1 - AluOperand2
	MEM(Memory Access)	Not required
	WB (Write Back)	RegFile[DestinationAddress] = AluResult

CMP Rd, Rs, Rt	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] AluOperand2 = Reg[inst[17:14]] DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	if (AluOperand1 == AluOperand2) AluResult = 0 else if (AluOperand1 < AluOperand2) AluResult = -1 else AluResult = 1
	MEM(Memory Access)	Not required
	WB (Write Back)	RegFile[DestinationAddress] = AluResult
ORI Rd, Rs, Imm	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] Imm=zero_extend(inst[13:0]) DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	AluResult = AluOperand    Imm
	MEM(Memory Access)	Not required
	WB (Write Back)	RegFile[DestinationAddress] = AluResult
ADDI Rd, Rs, Imm	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] Imm=sign_extend(inst[13:0]) DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	AluResult = AluOperand + Imm
	MEM(Memory Access)	Not required
	WB (Write Back)	RegFile[DestinationAddress] = AluResult
LW Rd, Imm(Rs)	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] Imm=sign_extend(inst[13:0]) DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	AluResult = AluOperand + Imm
	MEM(Memory Access)	MemData = DMem[AluResult]
	WB (Write Back)	RegFile[DestinationAddress] = MemData
SW Rd, Imm(Rs)	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	AluOperand1 = Reg[inst[21:18]] Imm=sign_extend(inst[13:0]) DestinationAddress = inst[25:22] Opcode = inst[31:26]
	EX (Execute)	AluResult = AluOperand + Imm
	MEM(Memory Access)	DMem[AluResult]=RegFile[DestinationAddress]
	WB (Write Back)	Not required

LDW Rd, Imm(Rs)	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4	
	ID (Instruction Decode)	Base = Reg[inst[21:18]] Imm=sign_extend(inst[13:0]) DestinationAddress = inst[25:22] Opcode = inst[31:26] // check if Dest % 2 != 0 → throw exception	
	EX (Execute)	Cycle#1	EffAddr1 = Base + Imm
		Cycle#2	EffAddr2 = Base + Imm + 1
	MEM(Memory Access)	Cycle#1	MemData1 = DMem[EffAddr1]
		Cycle#2	MemData2 = DMem[EffAddr2]
	WB (Write Back)	Cycle#1	RegFile[Dest] = MemData1
		Cycle#2	RegFile[Dest+1] = MemData2
SDW Rd, Imm(Rs)	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4	
	ID (Instruction Decode)	Base = Reg[inst[21:18]] Imm=sign_extend(inst[13:0]) Source = inst[25:22] Opcode = inst[31:26] // check if Source % 2 != 0 → throw exception	
	EX (Execute)	Cycle#1	EffectiveAddr = Base + Imm
		Cycle#2	EffectiveAddr2 = Base + Imm + 1
	MEM(Memory Access)	Cycle#1	RegFile[Source] → DMem[EffectiveAddr]
		Cycle#2	RegFile[Source + 1] → DMem[EffectiveAddr2]
	WB (Write Back)	Not required	
BZ Rs, Label	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4	
	ID (Instruction Decode)	Rs = Reg[inst[21:18]] If (zero) PC = PC +sign_ext(Imm) else PC =PC+4	
	EX (Execute)	Not required	
	MEM(Memory Access)	Not required	
	WB (Write Back)	Not required	
BGZ Rs, Label	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4	
	ID (Instruction Decode)	Rs = Reg[inst[21:18]] If (positive) PC = PC +sign_ext(Imm) else PC =PC+4	
	EX (Execute)	Not required	
	MEM(Memory Access)	Not required	
	WB (Write Back)	Not required	
	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4	

BLZ Rs, Label	ID (Instruction Decode)	Rs = Reg[inst[21:18]] If (negative) PC = PC +sign_ext(Imm) else PC =PC+4
	EX (Execute)	Not required
	MEM(Memory Access)	Not required
	WB (Write Back)	Not required
JR Rs	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	TargetAddr = Reg[inst[21:18]]
	EX (Execute)	Not required
	MEM(Memory Access)	Not required
	WB (Write Back)	Not required
J Label	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	TargetAddr = PC +sign_ext(Imm)
	EX (Execute)	Not required
	MEM(Memory Access)	Not required
	WB (Write Back)	Not required
CLL Label	IF (Instruction Fetch)	inst = instMem[PC] PC = PC + 4
	ID (Instruction Decode)	TargetAddr = PC +sign_ext(Imm)
	EX (Execute)	Not required
	MEM(Memory Access)	Not required
	WB (Write Back)	R14 = PC + 4

Each instruction either manipulates data in registers, interacts with memory, or alters the program's control flow. The simplicity of the instruction format and the use of a consistent encoding scheme make it particularly suitable for pipelined implementation, as decoding is straightforward and execution stages are well-defined.

## 2.3 Single Cycle Datapath:

Initially, a **single-cycle processor** was implemented to verify correct execution of instructions and validate the control signals. This served as the foundation for understanding datapath requirements and verifying instruction behavior.

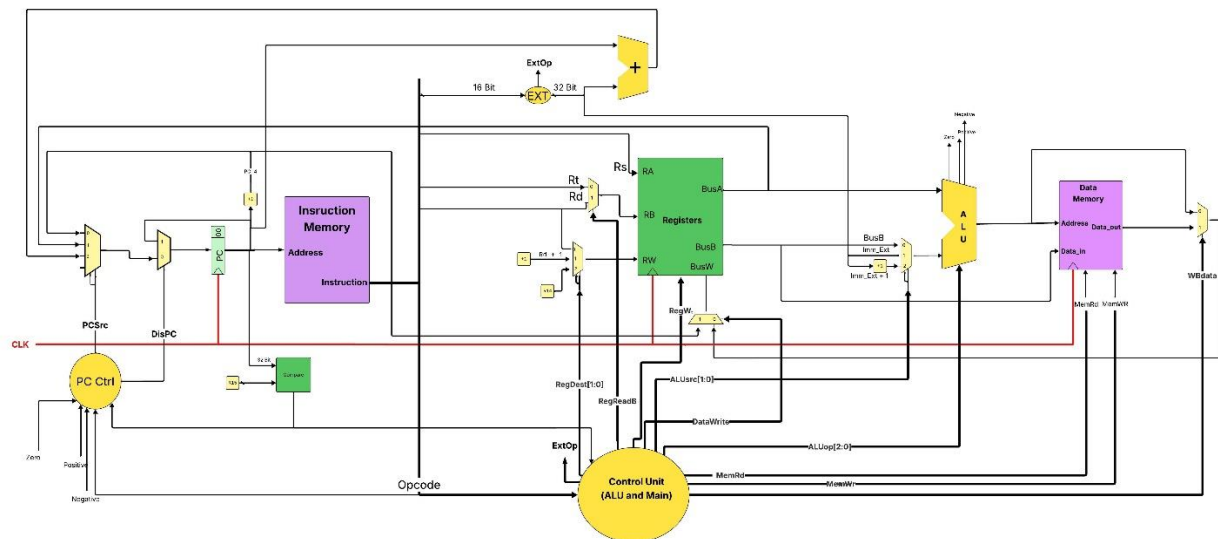


figure 1: single cycle datapath.

### 2.3.1 Instruction Fetch (IF)

- **Program Counter (PC):**  
The PC holds the address of the next instruction. Every clock cycle, it gives this address to the instruction memory. It can also jump to another address if needed.
- **Instruction Memory:**  
This block stores all the program instructions. It gives the instruction found at the address from the PC.
- **Adder (PC + 1):**  
This adds 1 to the PC value so the next instruction can be fetched in the next cycle.
- **PC Ctrl (PC Control):**  
This block chooses the next PC value based on jump or branch conditions like zero, positive, or negative.

- **First Mux (PCSrc):**  
This selects between different sources for the next PC, like normal PC+1, branch, or jump.
- **Second Mux (DisPC):**  
This decides if the PC should take the normal path or a special value (like in JUMP or Branch not taken conditions).

### 2.2.2 Instruction Decode (ID)

- **Registers (Register File):**  
This block contains the registers. It reads two source registers (RA and RB) and can write back a value to one register.
- **Instruction Fields:**  
The instruction is split into parts: opcode, source register, destination register, and immediate value.
- **EXT (Extender):**  
It takes a 14-bit immediate value and extends it to 32-bit for ALU operations.
- **Mux for write reg:**  
This selects the RegDest. It can be a register d or register d + 1 or register 14(link reg) .
- **Mux for Reg B:**  
This selects the second Reg RB for Bus B, either a Rt or the Rd .

### 2.2.3 Execution (EX)

- **ALU (Arithmetic Logic Unit):**  
The ALU does math and logic operations, like add, subtract, or compare. It uses the inputs from the previous step.
- **Mux for ALU input B:**  
This selects the second ALU input, either a register value or the extended immediate or immediate + 1 .

### 2.2.4 Memory Access (MEM)

- **Data Memory:**  
This is where data is read from or written to. The ALU gives the address. If it's a load, data is read. If it's a store, data is written.
- **MUX (WBdata Mux):**  
This selects what data to send back to the register file – either from the ALU or from data memory.

### 2.2.5 Write Back (WB)

- **MUX (DataWrite Mux):**  
This selects what data to send back to the register file – either from the WBdata Mux or from R15.
- **Register File (again):**  
The result from the ALU or memory is written into the destination register. This finishes the instruction cycle.

### 2.2.6 Control Unit

- **Main AND ALU Control Unit:**  
This block looks at the opcode of the instruction and creates all the control signals. These signals control the ALU, multiplexers, register file, memory, and PC.

## 2.4 Pipelining

### 2.4.1 Datapath

The pipelined datapath is designed to improve instruction throughput by dividing execution into five stages: IF, ID, EX, MEM, and WB. Pipeline registers between stages allow multiple instructions to be processed simultaneously. To handle data hazards, the design includes forwarding and hazard detection units, which resolve or stall dependencies as needed. Basic control hazard handling is also implemented for branch instructions. This structure enables efficient, parallel instruction execution while maintaining correct program behavior.

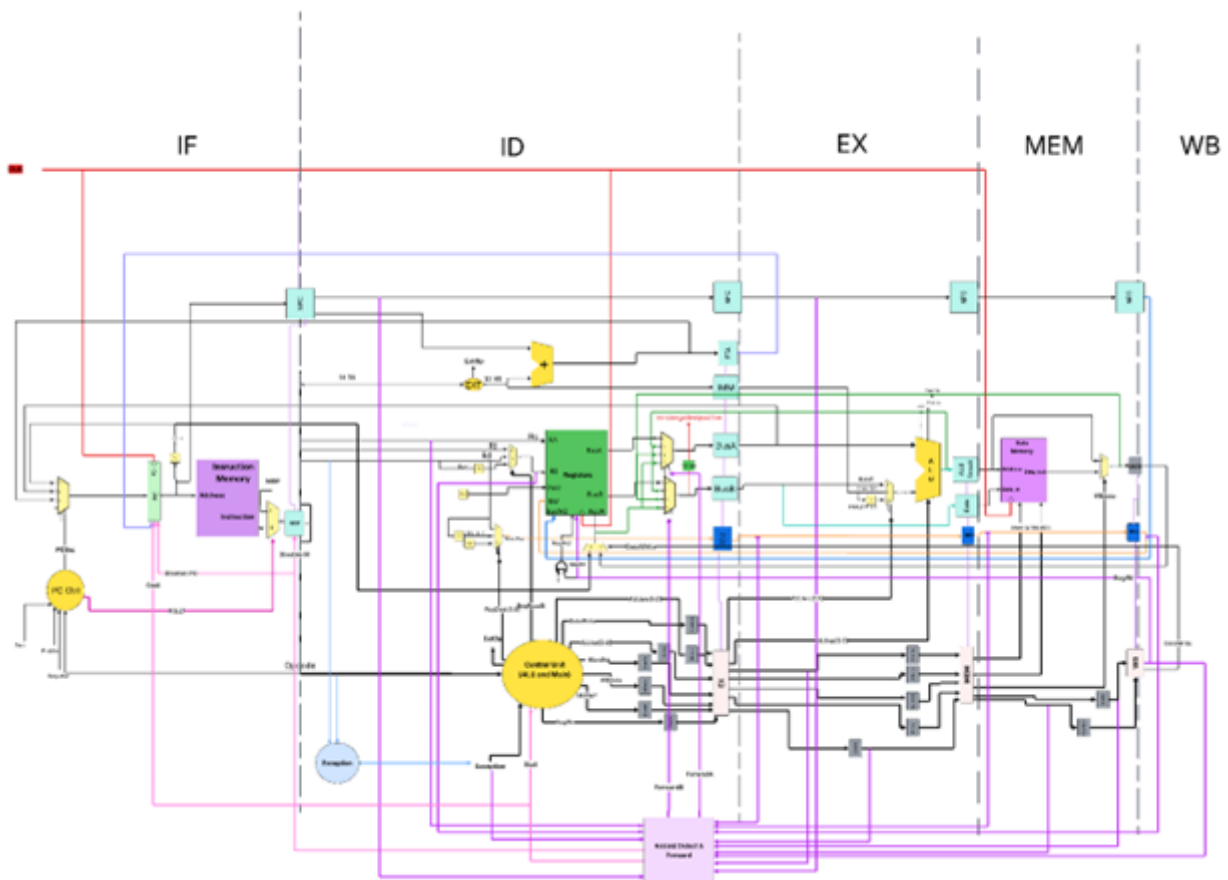


Figure 2: pipelined processor datapath

### 2.4.2 Instruction Fetch (IF)

The Fetch stage is the first stage in a pipelined CPU. Its role is to Fetch the instruction from instruction memory using the Program Counter (PC). Pass the instruction and the PC value to the next pipeline stage (Decode) via a pipeline register.

#### **Components:**

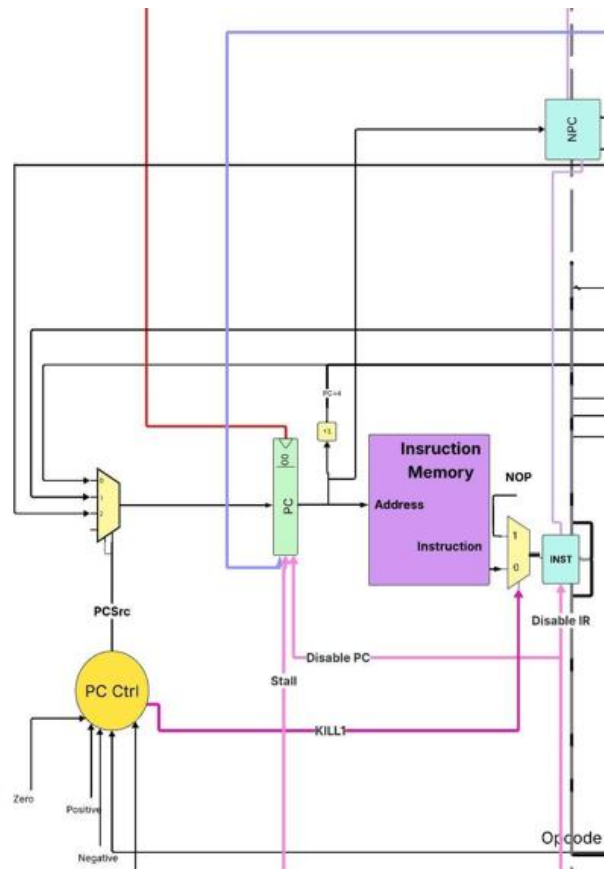


figure 3: pipelined processor (IF).

The **PC Control** unit determines the next source of the program counter (PC) update based on the current instruction and condition flags. It receives the opcode from the Instruction Register and status flags such as zero, positive, and negative from the ALU. Using this information, it outputs a 2-bit control signal **PCSrc** that selects among different PC update options: normal sequential increment ( $PC + 4$ ), branch target address, jump register address, or others depending on the instruction type. Additionally, the PC Control generates a kill signal to flush or invalidate instructions in the pipeline when a branch or jump is

taken. This logic ensures that the processor correctly changes the program flow on branches, jumps, or exceptions.

The **Program Counter (PC)** holds the address of the instruction to be fetched next. On each clock cycle, it updates according to the control signal PCSrc received from the PC Control unit. Depending on PCSrc, the PC either increments by 4 (to point to the next sequential instruction), or it loads a branch target, jump target, or other specified address. The PC also supports stalling and flushing mechanisms: it can hold its value during pipeline stalls to avoid fetching new instructions, or update accordingly during exceptions. This module plays a central role in guiding instruction fetch by controlling which instruction address is sent to the instruction memory.

The **Instruction Memory** stores the program's instructions as 32-bit words in a word-addressable array. It takes the current PC value as its address input. Since instructions are word-aligned (4 bytes each), the least significant two bits of the PC are ignored, and bits [31:2] of the PC address are used to index the memory array. This means the instruction memory accesses instructions based on word addresses rather than byte addresses. The output is the 32-bit instruction located at the specified address, which is then passed to the Instruction Register for decoding and execution.

**4-to-1 Multiplexer:** Chooses the next PC value using the PCSrc signal, selecting between PC+4 (next instruction), branch target address, jump target address, or holding the PC for stalls.

**2-to-1 Multiplexer:** Uses the kill signal to decide whether to load the fetched instruction into the Instruction Register or replace it with a NOP, effectively flushing the pipeline when necessary.

**Instruction Register (IF/ID Pipeline Register):** Temporarily stores the fetched instruction and PC value, passing them to the next pipeline stage while supporting pipeline stalls and flushes.

### 2.4.3 Instruction Decode (ID)

The Decode Stage plays a critical role in translating the fetched instruction into control signals and operands needed for the next pipeline stages. It breaks down the instruction, identifies operand registers, extends immediate values, and prepares all necessary control signals. In addition, it handles data hazards through forwarding logic and hazard detection. Below is a detailed explanation of the major components involved in the Decode stage:

### Components:

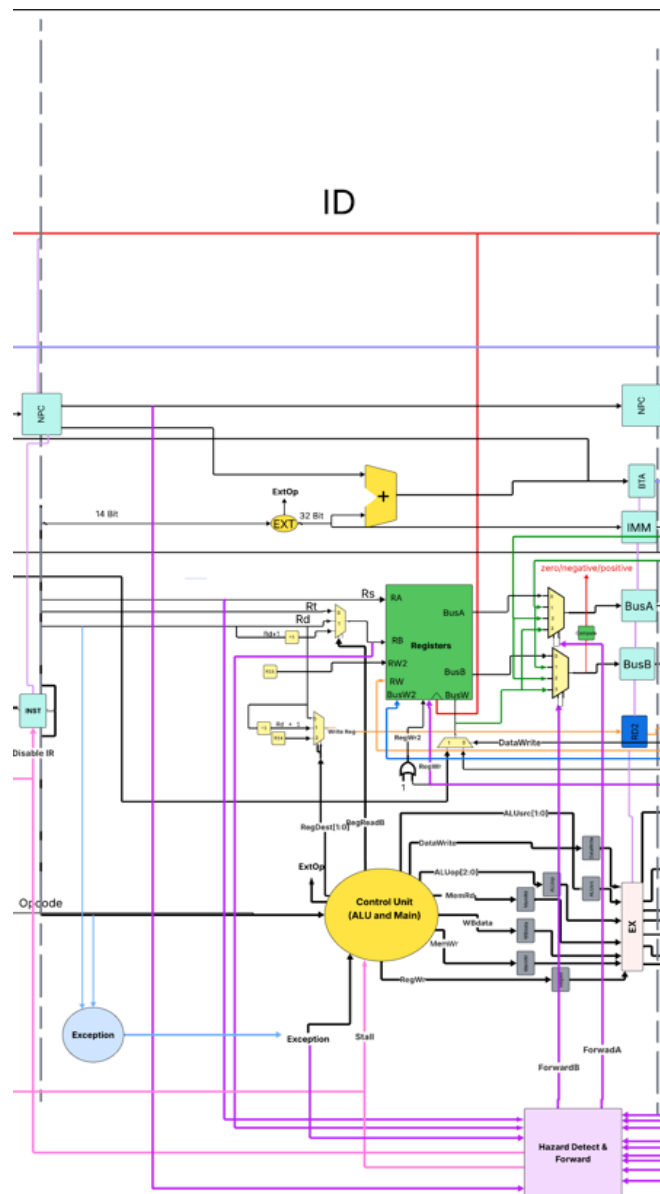


Figure 4: pipelined processor (ID)

## Control Unit

The control unit receives the extracted opcode and generates control signals that direct how the datapath behaves in later stages. It determines whether the instruction will write to a register (RegWr), read from or write to memory (MemRd, MemWr), or use immediate values (ExtOp, ALUSrc). It also defines which data should be written back to the register file (WBdata) and selects the destination register (RegDst). Importantly, the control unit considers pipeline stalls and exceptions, ensuring the proper behavior of the pipeline under hazard conditions.

## Immediate Extender

For instructions involving immediate values (such as addi, lw, or sw), the 14-bit immediate must be extended to 32 bits before it can be used in arithmetic or address computations. The Immediate Extender performs either **sign-extension** or **zero-extension** based on the ExtOp signal from the control unit. This ensures that operations behave correctly regardless of whether the immediate represents a signed or unsigned value.

## Register File

Register File stores all 32 general-purpose registers and provides the values of the source registers Rs and Rt during the Decode stage. These values are read using two separate read busses, BusA and BusB, and forwarded to later pipeline stages. The Register File supports writing through two write busses BusW and BusW2. While BusW is used for general register writes (storing ALU or memory results), BusW2 is dedicated to writing to register R15, which is reserved for holding the updated Program Counter (PC).

## RegDst and RegReadB Muxes

Two multiplexers determine which registers are used as the destination and source for certain instructions. The **RegReadB Mux** selects the true second source register (real\_Rt) based on control signals. This is important for instructions that vary in how they interpret operand fields. The **RegDst Mux** determines which register will be the destination for the write-back stage: Rd, Rd + 1, or a fixed register like the return address register R14 (used in call instructions). These decisions are based on the RegDst signal from the control unit.

## Branch Target Address (BTA) Calculation

To support branch instructions, the Decode stage calculates the Branch Target Address (BTA). This is done by adding the sign-extended immediate value to the current program counter (PC). The BTA is used later to update the PC if the branch condition evaluates to true.

## **Comparator**

The comparator evaluates the branch condition by comparing the values of source registers Rs and Rt. It checks whether the values are equal, positive, or negative, producing zero, positive, or negative flags. These flags help determine the outcome of conditional branches and guide the PCSrc selection in the Fetch stage.

## **Forwarding Unit**

The Forwarding Unit is responsible for resolving data hazards in the pipeline by forwarding results from later stages back to earlier ones when necessary. It compares the destination registers of previous instructions with the source registers of the instruction currently in the Execute stage. If it detects that a required value has not yet been written back to the Register File but is available in a later stage, it sets control signals (ForwardA and ForwardB) to redirect the appropriate data. This helps to avoid incorrect computations and unnecessary stalls. Additionally, the unit handles special conditions such as load-use hazards by asserting a stall signal, and it raises an exception when an invalid operation is detected, such as attempting to load into an odd-numbered register (e.g., LDW to an odd register).

### **MUX2to1 ForwardA**

The MUX2to1 ForwardA is a multiplexer that selects the appropriate value for the first ALU operand in the Execute stage, based on the ForwardA control signal generated by the Forwarding Unit. When ForwardA is 00, the value is taken directly from the Register File (BusA). If ForwardA is 01, it selects the ALU result from the EX stage (EX/MEM pipeline). If the signal is 10, it uses data from the MEM/WB stage, which represents the value written back by a previous instruction. Lastly, a signal of 11 means the value is taken from the BUSW line, allowing the pipeline to forward special results such as those used in control operations or instructions like CLL. This dynamic selection ensures that the ALU receives the most up-to-date operand value.

### **MUX2to1 ForwardB**

Similarly, the MUX2to1 ForwardB is a multiplexer that determines the source for the second ALU operand. It uses the ForwardB control signal to select among four input options. A control value of 00 means the operand comes from the Register File (BusB). A value of 01 selects the EX stage ALU output. If the signal is 10, the operand is taken from the MEM/WB stage. Finally, if the control is 11, the selected value is BUSW, supporting cases where data needs to be forwarded for instructions that perform memory writes or other special operations. Like ForwardA, this forwarding path avoids delays and ensures correct data is used for computation.

### **ID/EX Pipeline Register**

At the end of the Decode stage, all control signals, operand values, and instruction-specific information are passed into the ID/EX pipeline register. This register holds the extended immediate, BTA, source operands (BusA and BusB), ALU control signals, memory access signals, and the register destination. This ensures that all necessary data is available for the Execute stage in the next cycle.

### 2.4.4 Execution (EX)

The Execution (EX) stage is responsible for performing arithmetic and logic operations based on decoded instructions. It receives control signals and operand values from the Decode stage and produces the result of the operation to be forwarded to the Memory stage. The EX stage includes key components such as the ALU, various multiplexers for operand selection, and connections for forwarding data to resolve hazards.

#### Components:

The **ALU** is the central component in this stage. It receives two inputs: one from the BusA (usually from a source register) and the other from a MUX that chooses between BusB and the IMM value. The ALU executes an operation specified by control signals (ALUOp[2:0]) such as addition, subtraction, AND, OR, etc. The result is sent to the EX/WB pipeline register.

**MUX4to1** selects between BusB (the second operand from a register) and the sign-extended immediate value (Imm). The selection is controlled by the ALUUseImm signal. If ALUSrc = 1, it means the instruction uses an immediate value (like ADDI), so Imm is selected. If it's 0, BusB is used instead. This MUX ensures that the correct data is sent to the ALU's second input depending on the instruction type.

The **EX/WB Pipeline Register** plays a crucial role in passing data and control signals from the Execution stage to the Write-Back stage. It temporarily stores the ALU result, the value of BusB, and the destination register address, along with control signals. This ensures that the correct data is written back to the register file in the next cycle. By holding these values at the end of the execution phase, it maintains pipeline consistency and supports data forwarding for upcoming instructions, helping prevent hazards and preserving instruction flow accuracy.

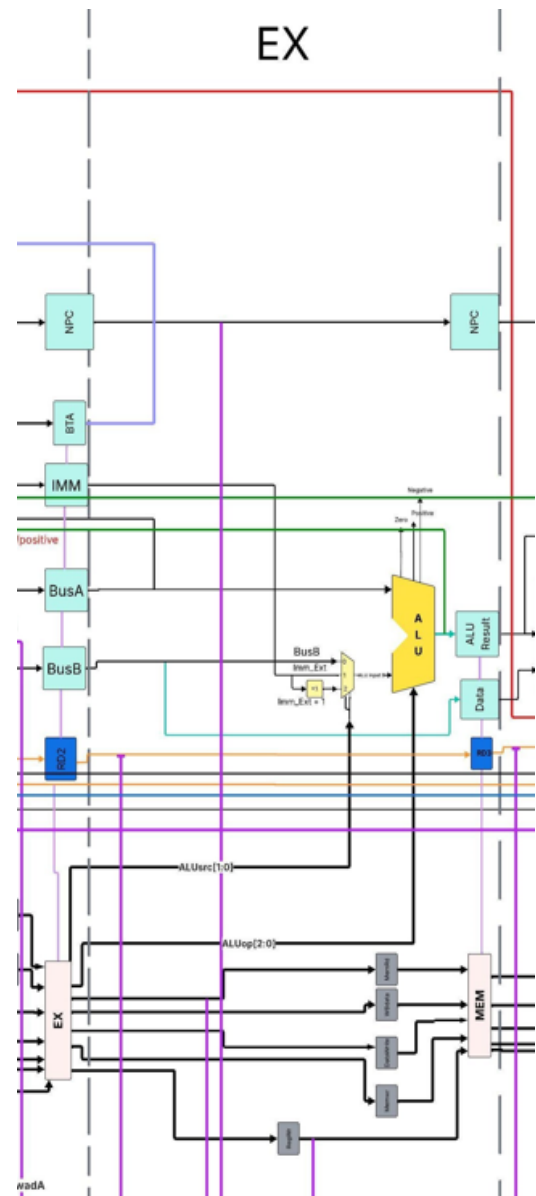


figure 5: pipelined processor (EX).

### 2.4.6 Memory Access (MEM)

The **MEM stage** is responsible for handling memory-related operations such as loading data from memory or storing data into memory. It receives the effective memory address from the ALU result produced in the Execution stage. Based on the instruction type and control signals, it decides whether to read from or write to the data memory. The memory access stage also prepares the data that will be written back to the register file in the next stage.

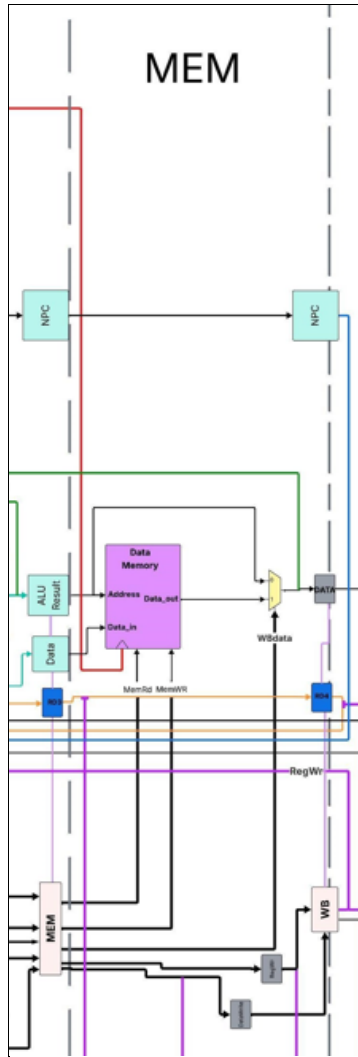


figure 6: pipelined processor (MEM).

#### Components:

**Data Memory** is accessed in this stage using the address output from the ALU. If the instruction is a load (e.g., LDW), data is read from memory. If it is a store (e.g., STW), the value from BusB (which contains the data to be written) is written to the calculated memory address. This memory block operates based on control signals such as MemRead and MemWrite that determine the direction of the operation.

A **2-to-1 multiplexer (MUX)** follows the memory block and is responsible for selecting the correct data to forward to the Write-Back stage. The selection is based on whether the instruction requires the ALU result (e.g., for arithmetic operations) or the data fetched from memory (e.g., for load instructions). The output of this MUX is referred to as WBdata, which will eventually be written to the register file.

The **MEM/WB Pipeline Register** captures the outputs of the MEM stage, including the selected WBdata, the destination register address, and relevant control signals like RegWrite. It ensures the stability and correct timing of these values as they move into the Write-Back stage, allowing proper register updates and maintaining pipeline consistency.

### 2.4.7 Write Back (WB)

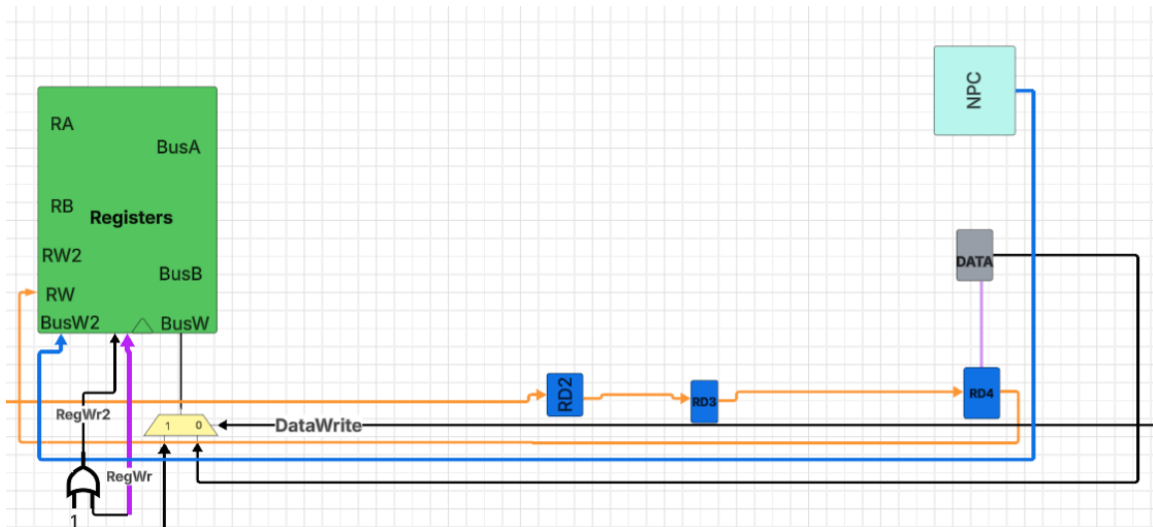


figure 7: pipelined processor (WB).

In the **Write Back (WB)** stage, results from either the ALU or memory are written back to the register file. The standard write-back path uses BusW, which carries the selected result (via a multiplexer) to the destination register specified by the RW line, under the control of the RegWr signal. Additionally, another write-back path, BusW2, has been introduced specifically to update **register R15**, which serves as the **program counter (PC)**. This ensures that the value of the PC stored in the MEM/WB pipeline register is **always written to R15**, independent of regular register writes. The RegWr2 control signal manages this path, allowing the PC to be updated reliably without affecting regular register operations.

## 2.5 Control Signals

### 2.5.1 ALU & Main control unit

Table 4: ALU & Main control unit signals.

Input			Output									
Control Signals Opcode	Exception	Stall	RegDst	ALUSrc	RegReadB	RegWr	MemRd	MemWr	ExtOp	ALUOp	WBdata	Data_write
X	1	X	0	0	0	0	0	0	0	0	0	0
OR = 0	0	0	0	0	0	1	0	0	X	0	1	0
ADD = 1	0	0	0	0	0	1	0	0	X	1	1	0
SUB = 2	0	0	0	0	0	1	0	0	X	2	1	0
CMP = 3	0	0	0	0	0	1	0	0	X	3	1	0
ORI = 4	0	0	0	1	0	1	0	0	0	0	1	0
ADDI = 5	0	0	0	1	0	1	0	0	1	1	1	0
LW = 6	0	0	0	1	0	1	1	0	1	1	1	0
SW = 7	0	0	X	1	1	0	0	1	1	1	X	X
LDW Round 1 = 8	0	1	0	1	0	1	1	0	1	1	1	0
LDW Round 2 = 8	0	0	1	2	0	1	1	0	1	1	1	0
SDW Round 1 = 9	0	1	X	1	1	0	0	1	1	1	X	X
SDW Round 2 = 9	0	0	X	2	2	0	0	1	1	1	X	X
BZ = 10	0	0	X	X	X	0	0	0	1	X	X	X
BGZ = 11	0	0	X	X	X	0	0	0	1	X	X	X
BLZ = 12	0	0	X	X	X	0	0	0	1	X	X	X

<b>JR =13</b>	0	0	X	X	X	0	0	0	1	X	X	X
<b>J = 14</b>	0	0	X	X	X	0	0	0	1	X	X	X
<b>CLL = 15</b>	0	0	2	X	X	1	0	0	1	X	X	1

- **RegDst:** Determines the destination register field (e.g., selects between Rt, Rd, or Rd+1 ).
- **ALUSrc:** Selects the second input to the ALU — either a register, immediate value, or immediate +1.
- **RegReadB:** Selects the source register for the second operand, especially used in memory store instructions.
- **RegWr:** Enables writing the result back to a register (i.e., activates register write).
- **MemRd:** Enables memory read — used in load instructions like LW or LDW.
- **MemWr:** Enables memory write — used in store instructions like SW or SDW.
- **ExtOp:** Controls whether to sign-extend or zero-extend the immediate value.
- **ALUOp:** Specifies the operation the ALU should perform (e.g., OR, ADD, SUB, CMP).
- **WBdata:** Selects the data source to be written back to the register (from ALU or from memory).
- **Data\_write:** Used by special instructions (like CLL) to enable manual data write, such as writing the PC value.

Signal	Boolean Expression
<b>RegDst</b>	$(\neg \text{Exception} \wedge \neg \text{Stall}) \wedge ((\text{Opcode}==0) \vee (\text{Opcode}==1) \vee (\text{Opcode}==2) \vee (\text{Opcode}==3) \vee (\text{Opcode}==15) \rightarrow 2 \vee ((\text{Opcode}==8 \wedge \neg \text{Stall}) \rightarrow 1))$
<b>ALUSrc</b>	$(\neg \text{Exception} \wedge \neg (\text{Stall} \wedge \neg (\text{Opcode}==8 \vee \text{Opcode}==9))) \wedge ((\text{Opcode}==4) \vee (\text{Opcode}==5) \vee (\text{Opcode}==6) \vee (\text{Opcode}==7) \vee (\text{Opcode}==8) \vee (\text{Opcode}==9))$
<b>RegReadB</b>	$(\neg \text{Exception}) \wedge (((\text{Opcode}==7 \wedge \neg \text{Stall}) \rightarrow 1) \vee ((\text{Opcode}==9 \wedge \neg \text{Stall}) \rightarrow 2) \text{ Round2}) \vee ((\text{Opcode}==9 \wedge \text{Stall}) \rightarrow 1) \text{ Round1})$
<b>RegWr</b>	$(\neg \text{Exception} \wedge \neg (\text{Stall} \wedge \neg ((\text{Opcode}==8) \wedge \neg (\text{ForwardA}==1 \vee \text{ForwardB}==1)) \rightarrow (\text{Round2 depend on Round1}))) \wedge ((\text{Opcode}==0) \vee (\text{Opcode}==1) \vee (\text{Opcode}==2) \vee (\text{Opcode}==3) \vee (\text{Opcode}==4) \vee (\text{Opcode}==5) \vee (\text{Opcode}==6) \vee (\text{Opcode}==8) \vee (\text{Opcode}==15))$
<b>MemRd</b>	$(\neg \text{Exception} \wedge \neg (\text{Stall} \wedge \neg ((\text{Opcode}==8) \wedge \neg (\text{ForwardA}==1 \vee \text{ForwardB}==1)) \rightarrow (\text{Round2 depend on Round1}))) \wedge ((\text{Opcode}==6) \vee (\text{Opcode}==8))$
<b>MemWr</b>	$(\neg \text{Exception} \wedge \neg (\text{Stall} \wedge \neg (\text{Opcode}==9))) \wedge ((\text{Opcode}==7) \vee (\text{Opcode}==9))$
<b>ExtOp</b>	$(\neg \text{Exception}) \wedge ((\text{Opcode}==5) \vee (\text{Opcode}==6) \vee (\text{Opcode}==7) \vee (\text{Opcode}==8) \vee (\text{Opcode}==9) \vee (\text{Opcode}==10) \vee (\text{Opcode}==11) \vee (\text{Opcode}==12) \vee (\text{Opcode}==13) \vee (\text{Opcode}==14))$
<b>ALUOp</b>	$(\neg \text{Exception}) \wedge ((\text{Opcode}==0 \rightarrow 000) \vee (\text{Opcode}==1 \vee 5 \vee 6 \vee 8 \vee 9 \rightarrow 001) \vee (\text{Opcode}==2 \rightarrow 010) \vee (\text{Opcode}==3 \rightarrow 011) \vee (\text{Opcode}==10 \vee 11 \vee 12 \rightarrow 100))$
<b>WBdata</b>	$(\neg \text{Exception}) \wedge ((\text{Opcode}==4) \vee (\text{Opcode}==5) \vee (\text{Opcode}==6) \vee (\text{Opcode}==8))$
<b>Data_write</b>	$(\neg \text{Exception}) \wedge (\text{Opcode}==15)$

### 2.5.1 PC Control Unit

	PCSrc	Kill	zero	negative	positive	Exception
<b>OR = 0</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>ADD = 1</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>SUB = 2</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>CMP = 3</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>ORI = 4</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>ADDI = 5</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>LW = 6</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>SW = 7</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>LDW = 8</b> <b>round1</b>	<b>3</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>LDW = 8</b> <b>round2</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>LDW = 8</b> <b>With Exception</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>1</b>
<b>SDW = 9</b> <b>round1</b>	<b>3</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>SDW = 9</b> <b>round2</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>

<b>SDW = 9</b> <b>With Exception</b>	<b>0</b>	<b>0</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>1</b>
<b>BZ = 10</b>	<b>BT = 0</b> <b>BNT = 2</b>	<b>BT = 0</b> <b>BNT = 1</b>	<b>BT = 0</b> <b>BNT = 1</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>BGZ = 11</b>	<b>BT = 0</b> <b>BNT = 2</b>	<b>BT = 0</b> <b>BNT = 1</b>	<b>x</b>	<b>x</b>	<b>BT = 0</b> <b>BNT = 1</b>	<b>0</b>
<b>BLZ = 12</b>	<b>BT = 0</b> <b>BNT = 2</b>	<b>BT = 0</b> <b>BNT = 1</b>	<b>x</b>	<b>BT = 0</b> <b>BNT = 1</b>	<b>x</b>	<b>0</b>
<b>JR = 13</b>	<b>1</b>	<b>1</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>J = 14</b>	<b>2</b>	<b>1</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>
<b>CLL = 15</b>	<b>2</b>	<b>1</b>	<b>x</b>	<b>x</b>	<b>x</b>	<b>0</b>

The PC Control Unit determines the next value of the Program Counter (PC) based on various instruction types and conditions. Below are the Boolean equations for the PC control signals:

signal	Boolean Expression
<b>PCSrc [0] = PC + 4</b>	<b>If ( (opcode = ADD    opcode = ADDI    opcode = SUB    opcode = ORRI    opcode = ORR    opcode = CMP    opcode = LW    opcode = SW    opcode = LDW(round2 or exception)    opcode = SDW(round2 or exception)    (opcode = BGZ &amp;&amp; POSITIVE == 0)    (opcode = BLZ &amp;&amp; NEGATIVE == 0)    (opcode = BZ &amp;&amp; ZERO == 0) ) &amp;&amp; ( ! stall ) ) Then</b>  <b>PC = PC + 4</b>
<b>PCSrc [1]</b>	<b>if(opcode = JR ) Then</b>  <b>PC = reg_addr R[14]</b>
<b>PCSrc [2]</b>	<b>if(opcode = J    opcode = CLL    (opcode = BGZ &amp;&amp; POSITIVE == 1)    (opcode = BLZ &amp;&amp; NEGATIVE == 1)    (opcode = BZ &amp;&amp; ZERO == 1)) Then</b>  <b>PC = PC + imm_ext</b>
<b>By default PC = PC</b>	<b>if(LDW(Round 1)    stall ) Then</b>  <b>PC = PC</b>

### 2.5.3 Hazard Control Unit

The Hazard Control Unit handles data hazards by generating forwarding and stall signals to ensure correct instruction execution. Below are the Boolean equations for the forwarding and stall signals:

#### Hazard Unit Logic Equations

Table 5: Boolean Expressions for Hazard Control Unit Signals.

signal	Boolean Expression
ForwardA = 0	ForwardA != 1 && ForwardA != 2
ForwardA = 1	if( ID_EX_RegWrite && (ID_EX_Rd != 0) && (ID_EX_Rd == Rs) )
ForwardA = 2	EX_MEM_RegWrite && (EX_MEM_Rd != 0) &&(EX_MEM_Rd ==Rs)
ForwardA = 3	MEM_WB_RegWrite && (MEM_WB_Rd !=0)&&(MEM_WB_Rd==Rs) && Datawrite==0
ForwardB = 0	ForwardB != 1 && ForwardB != 2
ForwardB = 1	ID_EX_RegWrite && (ID_EX_Rd != 0) && (ID_EX_Rd == Rt)
ForwardB = 2	EX_MEM_RegWrite && (EX_MEM_Rd != 0) &&(EX_MEM_Rd ==Rt)
ForwardB = 3	MEM_WB_RegWrite && (MEM_WB_Rd != 0) &&(MEM_WB_Rd == Rt) && Datawrite==0

Signal	Boolean Expression
Exception	if ((opcode == 6'd8    opcode == 6'd9) && RegDest[0] == 1'b1)
Stall	if ( ((ID_EX_MemRd == 1 → EX.MemRd == 1 ) && (ForwardA==1    ForwardB==1))    (((opcode == 6'd8) && (IF_ID_PC != ID_EX_PC))    ((opcode == 6'd9) && (IF_ID_PC != ID_EX_PC)) ) && Exception == 0) → Round1(to go to Round2) )
Disable IR  PC_out <= PC_out and instruction_out = instruction_out (in IF_ID stage)	if( stall == 1 )  PC_out <= PC_out and instruction_out = instruction_out (in IF_ID stage)
Disable PC  PC<= PC (in program counter)	if( stall == 1 )  PC<= PC (in program counter)



### 3.1.1 Analysis

#### 1. Initial Register Values

At the start, the registers (`R0` to `R15`) are initialized with their own index numbers: `R0 = 0`, `R1 = 1`, `R2 = 2`, and so on.

#### 2. Avoiding Forwarding Hazards (Data hazard)

We successfully solved the data forwarding hazard problem, which greatly improved performance by reducing stall cycles from 2 to 0. For example, when the first instruction (Memory[0] ADDI R1,R0,5) is in the memory stage and the third instruction (Memory[2] ADD R3,R1,R2) is in the decode stage, the third instruction would normally wait 2 cycles for the first instruction to write its result to the register file. With only structural hazard solutions, this wait time reduces to 1 cycle. However, with data forwarding and structural hazard implemented, there are no stall cycles at all because the data is passed directly between pipeline stages without waiting for the register file write-back. Same that for Memory[1] & Memory[3] .

#### 3. Single-Cycle Read & Write (Structural hazard)

The processor seems to be single-cycle, meaning each instruction finishes in one clock cycle. This avoids complex hazards because: Reads and writes happen in the same cycle. In the first half of the cycle Writing in the register file , and in the second half there is reading from the register file (Memory[0] ADDI R1,R0,5, → Writing , Memory[3] SUB R4,R2,R1 → Reading ).

### 3.1.2 Instruction Execution Profile Report

```
◦ # KERNEL: -----
◦ # KERNEL: =====
◦ # KERNEL: Total Number of Cycles = 13
◦ # KERNEL: Total Executed Instructions = 9
◦ # KERNEL: Total Stall Cycles = 0
◦ # KERNEL: Total Load Instructions = 0
◦ # KERNEL: Total Store Instructions = 0
◦ # KERNEL: Total Load Instructions Double = 0
◦ # KERNEL: Total Store Instructions Double = 0
◦ # KERNEL: Total ALU Instructions = 9
◦ # KERNEL: Total Control Instructions = 0
◦ # KERNEL: =====
```

Figure 9: Instruction Execution Analysis

### 3.1.3 Contact of registers

```

> # KERNEL: R1 = 5
> # KERNEL: R2 = 10
> # KERNEL: R3 = 15
> # KERNEL: R4 = 5
> # KERNEL: R5 = 15
> # KERNEL: R6 = 7
> # KERNEL: R7 = -1
> # KERNEL: R8 = 0
> # KERNEL: R9 = 1
> # KERNEL: R10 = 10
> # KERNEL: R11 = 11
> # KERNEL: R12 = 12
> # KERNEL: R13 = 13
> # KERNEL: R14 = 14
> # KERNEL: R15 = 32

```

Figure 10 : Register values after program execution

### 3.1.4 Waveform

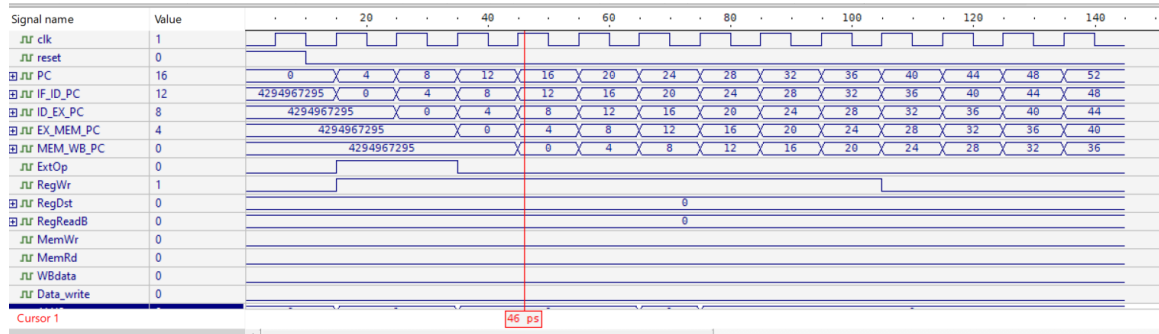


Figure 11 : Test 1 waveform 1

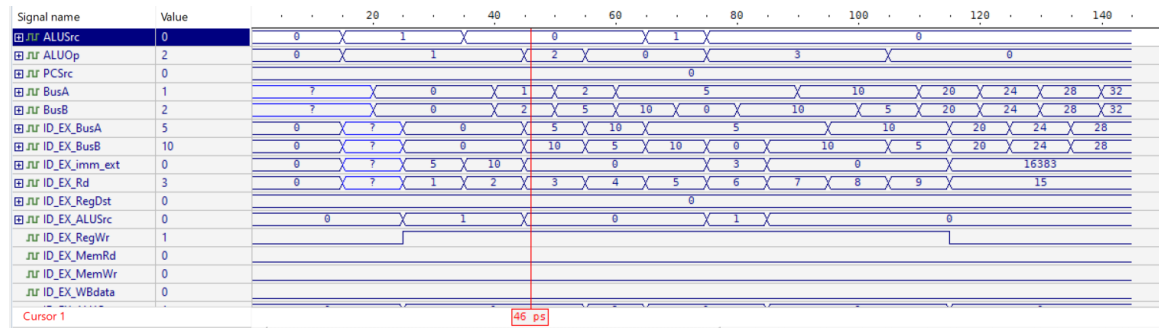


Figure 12 : Test 1 waveform 2

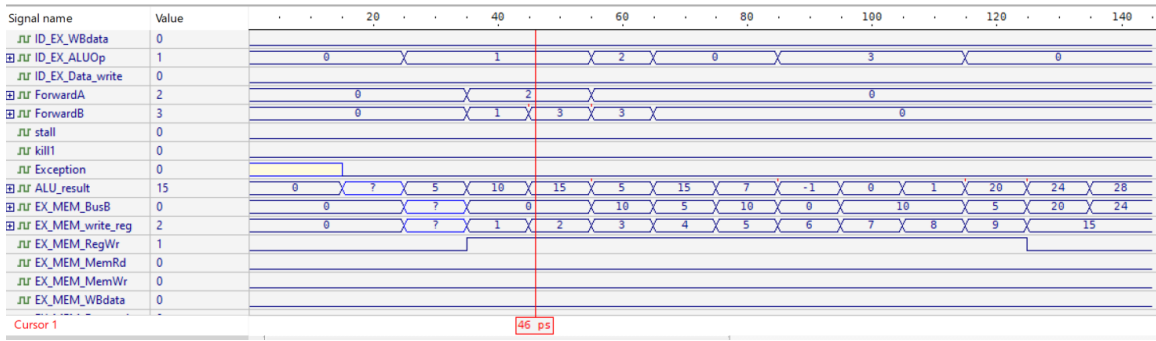


Figure 13 : Test 1 waveform 3

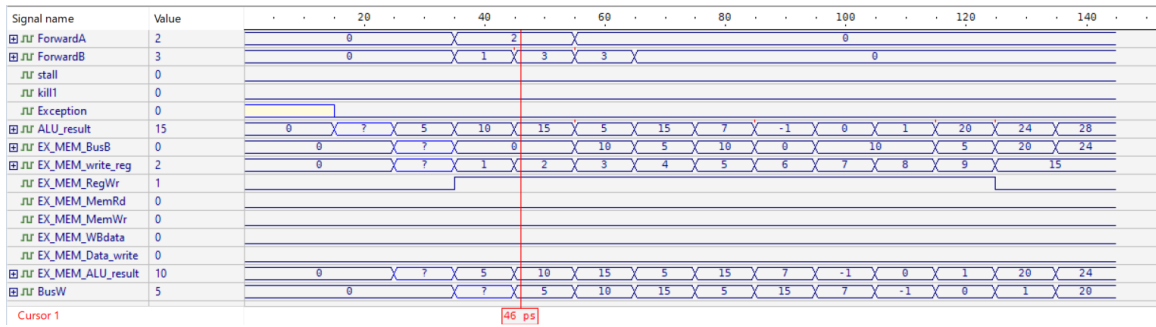


Figure 14 : Test 1 waveform 4

### 3.2 Test 2

```
// Example program with memory load/store and ALU ops
// Sample program 2
// LWD & SDW AND Rd is odd
memory[0] = {6'b000101, 4'd1, 4'd0, 4'd0, 14'd8}; // ADDI R1, R0, 8 ; R1 = 8
memory[1] = {6'b000101, 4'd2, 4'd0, 4'd0, 14'd20}; // ADDI R2, R0, 20 ; R2 = 20
memory[2] = {6'b001001, 4'd8, 4'd1, 4'd0, 14'd4}; // SDW R8, 4(R1) ; MEM[R1 + 4] = R8 MEM[5+R1] = R9
memory[3] = {6'b001001, 4'd6, 4'd0, 4'd0, 14'd0}; // SDW R6, 0(R0) ; MEM[0] = R6 MEM[1+R0] = R7
memory[4] = {6'b001000, 4'd3, 4'd0, 4'd0, 14'd0}; // LDW R3, 0(R0) ; Exception
memory[5] = {6'b001000, 4'd4, 4'd1, 4'd0, 14'd4}; // LDW R4, 4(R1) ; R4 = MEM[8+4] R5 = MEM[R1+5]
memory[6] = {6'b000001, 4'd5, 4'd2, 4'd3, 14'd0}; // ADD R5, R2, R3 ; R5 = R2 + R3
memory[7] = {6'b000010, 4'd6, 4'd5, 4'd4, 14'd0}; // SUB R6, R5, R4 ; R6 = R5 - R4
memory[8] = {6'b000000, 4'd7, 4'd3, 4'd4, 14'd0}; // OR R7, R3, R4 ; R7 = R3 | R4
memory[9] = {6'b000011, 4'd8, 4'd3, 4'd4, 14'd0}; // CMP R8, R3, R4 ; R8 = R3 & R4
```

Figure 15 : Test 2

The code in in assembly language :

ADDI R1, R0, 8

ADDI R2, R0, 20

SDW R8, 4(R1)

SDW R6, 0(R0)

LDW R3, 0(R0)

LDW R4, 4(R1)

ADD R5, R2, R3

SUB R6, R5, R4

OR R7, R3, R4

CMP R8, R3, R4

The code in machine code :

000101 0001 0000 0000 000000000001000

000101 0010 0000 0000 000000000010100

001001 1000 0001 0000 000000000000100

001001 0110 0000 0000 0000000000000000

001000 0011 0000 0000 0000000000000000

001000 0100 0001 0000 0000000000000100

000001 0101 0010 0011 0000000000000000

000010 0110 0101 0100 0000000000000000

000000 0111 0011 0100 0000000000000000

000011 1000 0011 0100 0000000000000000

### 3.2.1 Analysis

#### 1. Initial Register Values

At the start, the registers (`R0` to `R15`) are initialized with their own index numbers: `R0 = 0`, `R1 = 1`, `R2 = 2`, and so on.

#### 2. Load Double Word & Store Double Word Pipeline Handling

When a (**Load Double Word** / **Store Double Word**) instruction enters the decode stage, the processor performs a critical check to prevent pipeline hazards. Here's how this process works step by step:

##### Initial Check and Stall Decision

The system compares the Program Counter (PC) values between the execution stage and decode stage. If the PC in the execution stage does not equal the PC in the decode stage, a stall occurs. However, this stall has a unique behavior - instead of passing control signals as zeros, it passes the control signals from the first round of the (Load Double Word / Store Double Word) instruction.

##### First Round Processing

During the stall, the first round moves from the decode stage to the execution stage. Since a stall has occurred, the Instruction Register (IR) is disabled, which causes the processor to fetch the same (Load Double Word / Store Double Word) instruction again, but this time it processes the second round.

##### PC Comparison and Resolution

The processor then checks the PC values in both the decode and execution stages again. At this point, they are equal because:

- **Decode stage:** Contains (Load Double Word / Store Double Word ) (second round)
- **Execution stage:** Contains (Load Double Word / Store Double Word ) (first round)

### Completion of Operation

Once the PC values match, the second round passes to the execution stage with its proper control signals. The PC is incremented by 4, and the next instruction is fetched since no stall occurs.

### Stall Conditions Summary

A stall happens when:

- The PC in decode stage equals the PC in execution stage and the operation code is Load Double Word or Store Double Word (opcodes 8 or 9) , so here is no first-type forwarding available (since a stall is needed to resolve the hazard)

**Note:** If there is a Load Double Word instruction where the destination register matches a source register, forwarding occurs but requires 2 stalls total - one for the forwarding operation and one for the second round processing(e.g LWD R8,4[R8] ).

### 3. Data Hazards (RAW - Read After Write)

Instructions depend on results from previous instructions that haven't completed yet.

#### Examples in the code:

- **Line 4-6:** LDW R3, 0(R0) followed by ADD R5, R2, R3 (Forward = 2 , stall = 1 → 0)
  - The ADD instruction needs R3, but LDW hasn't finished loading it yet
- **Line 5-7:** LDW R4, 4(R1) followed by SUB R6, R5, R4 (Forward = 2 , stall = 1 → 0)
  - SUB needs R4 from the previous LDW instruction
- **Line 6-7:** ADD R5, R2, R3 followed by SUB R6, R5, R4 (Forward = 1 , stall = 2 → 0)
  - SUB needs R5 which is being calculated by ADD
- **Data Forwarding:** Pass results directly between pipeline stages
- **Pipeline Stalls:** Insert wait cycles when forwarding isn't possible (Decode stage and Execution stage )

### 4 . Exception Handling for Odd Register Destinations (LWD,SWD)

When a Load Double Word or Store Double Word instruction has an odd-numbered destination register, an exception handler is triggered. In this case, the instruction is handled differently: it is not executed at all. Instead, zero control signals are passed when moving from the decode stage to the execution stage, and the Program Counter (PC) is incremented by 4. This means no stall occurs - only zero signals are passed through the pipeline, and the program continues without modifying any registers or memory locations.

### 3.2.2 Instruction Execution Profile Report

```

◦ # KERNEL: =====
◦ # KERNEL: Total Number of Cycles = 17
◦ # KERNEL: Total Executed Instructions = 9
◦ # KERNEL: Total Stall Cycles = 4
◦ # KERNEL: Total Load Instructions = 0
◦ # KERNEL: Total Store Instructions = 0
◦ # KERNEL: Total Load Instructions Double = 1
◦ # KERNEL: Total Store Instructions Double = 2
◦ # KERNEL: Total ALU Instructions = 6
◦ # KERNEL: Total Control Instructions = 0
◦ # KERNEL: =====

```

Figure 16: Instruction Execution Analysis

### 3.2.3 Contact of registers

```

◦ # KERNEL: =====
◦ # KERNEL: R1 = 8
◦ # KERNEL: R2 = 20
◦ # KERNEL: R3 = 3
◦ # KERNEL: R4 = 8
◦ # KERNEL: R5 = 23
◦ # KERNEL: R6 = 15
◦ # KERNEL: R7 = 11
◦ # KERNEL: R8 = -1
◦ # KERNEL: R9 = 9
◦ # KERNEL: R10 = 10
◦ # KERNEL: R11 = 11
◦ # KERNEL: R12 = 12
◦ # KERNEL: R13 = 13
◦ # KERNEL: R14 = 14
◦ # KERNEL: R15 = 36
◦ # KERNEL: =====

```

Figure 17: Register values after program execution

### 3.2.4 Waveform

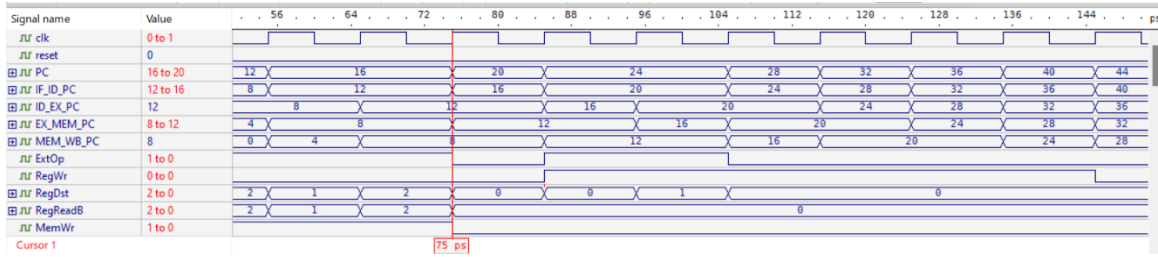


Figure 18:Test 2 waveform 1

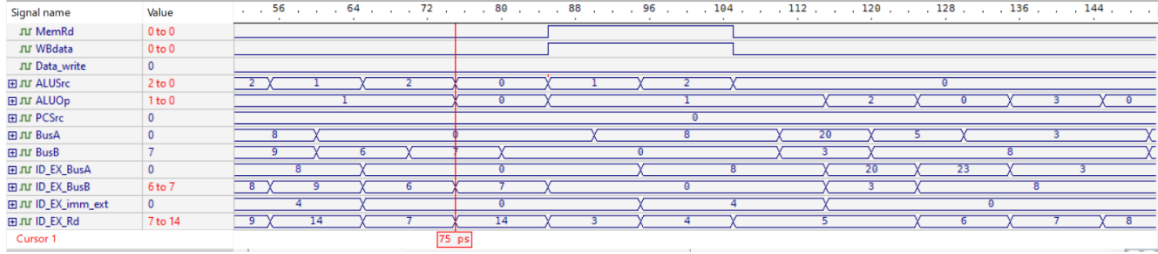


Figure 19:Test 2 waveform 2

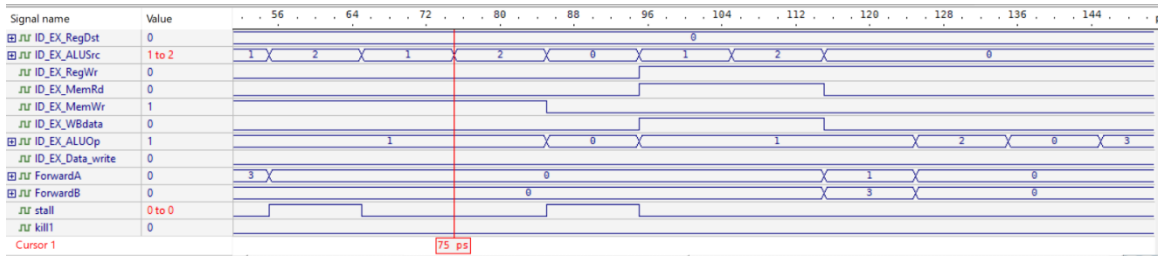


Figure 20:Test 2 waveform 3

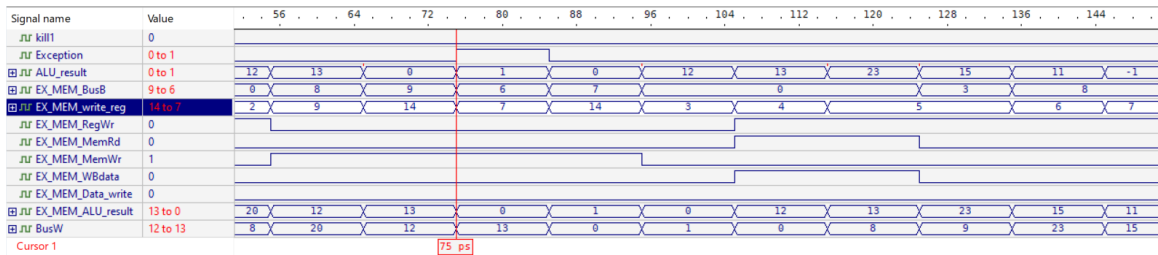


Figure 21:Test 2 waveform 4

### 3.3 Test 3

```
// Example program with memory load/store and ALU ops and jump
// Sample program 3
// LWD , SWD, Jump , Branch (predict false (true predict)) , Forwarding
memory[0] = {6'd14, 4'd0, 4'd0, 4'd0, 14'd3}; // J + 3 <- unconditional jump
memory[1] = {6'd5, 4'd1, 4'd0, 4'd0, 14'd20}; // ADDI R1, R0, 20
memory[2] = {6'd5, 4'd2, 4'd0, 4'd0, 14'd40}; // ADDI R2, R0, 40
memory[3] = {6'd7, 4'd4, 4'd1, 4'd0, 14'd4}; // SW R4, [R1 + 4] <- normal store
memory[4] = {6'd6, 4'd3, 4'd1, 4'd0, 14'd4}; // LD R3, [R1 + 4] <- load
memory[5] = {6'd0, 4'd4, 4'd3, 4'd2, 14'd0}; // OR R4, R3, R2 <- causes stall (uses result of load)
memory[6] = {6'd10, 4'd0, 4'd6, 4'd0, 14'd3}; // BZ R6, 3 <- if R6 == 0, skip next 3
memory[7] = {6'd8, 4'd6, 4'd1, 4'd0, 14'd4}; // LWD R6, [R1 + 4] <- double load
memory[8] = {6'd2, 4'd8, 4'd6, 4'd2, 14'd0}; // SUB R8, R6, R2 <- causes stall (uses double load)
memory[9] = {6'd9, 4'd6, 4'd1, 4'd0, 14'd8}; // SWD R6, [R1 + 8] <- double store
memory[10] = {6'd7, 4'd4, 4'd1, 4'd0, 14'd12}; // SW R4, [R1 + 12] <- normal store
memory[11] = {6'd5, 4'd7, 4'd0, 4'd0, 14'd100}; // ADDI R7, R0, 100 <- skipped if branch taken
memory[12] = {6'd4, 4'd8, 4'd0, 4'd0, 14'd200}; // ORI R8, R0, 200 <- skipped by jump
memory[13] = {6'd5, 4'd9, 4'd8, 4'd0, 14'd255}; // ADDI R9, R8, 255 <- executed
```

Figure 22:Test 3

The code in in assembly language :

J +3

ADDI R1, R0, 20

ADDI R2, R0, 40

SW R4, [R1 + 4]

LD R3, [R1 + 4]

OR R4, R3, R2

BZ R6, 3

LWD R6, [R1 + 4]

SUB R8, R6, R2

SWD R6, [R1 + 8]

SW R4, [R1 + 12]

ADDI R7, R0, 100

ORI R8, R0, 200

ADDI R9, R8, 255

The code in machine code :

```
00111000000000000000000000000011
000101000100000000000000000010100
0001010010000000000000000000101000
000111010000010000000000000000100
000110001100010000000000000000100
00000001000011001000000000000000
00101000000110000000000000000011
001000011000010000000000000000100
00001010000110001000000000000000
0010010110000100000000000000001000
0001110100000100000000000000001100
00010101110000000000000000001100100
000100100000000000000000000011001000
000101100110000000000000000011111111
```

### 3.3.1 Analysis

#### Branch and Jump Instruction Handling

##### Branch Predicted False (Branch Value is True)

When a jump or branch instruction enters the decode stage, the processor fetches the next sequential instruction. However, since the branch will actually be taken, the fetched instruction needs to be canceled. The system kills this instruction by passing zero control signals through the pipeline, effectively making the instruction value zero while keeping

the PC unchanged. The branch or jump instruction then moves from decode to execution stage. To optimize performance, branch processing uses a comparator in the decode stage instead of waiting for the execution stage, reducing the penalty from 2 stall cycles to just 1 stall cycle.

### Branch Predicted True (Branch Value is False)

When the branch prediction is true but the actual branch value is false, the processor handles this more efficiently. In this case, there is no need to kill the instruction that was fetched from the decode stage, and no stall cycles are required since the prediction was correct and the sequential execution can continue normally.

### 3.3.2 Instruction Execution Profile Report

```

◦ # KERNEL: Total Number of Cycles = 20
◦ # KERNEL: Total Executed Instructions = 12
◦ # KERNEL: Total Stall Cycles = 3
◦ # KERNEL: Total Load Instructions = 1
◦ # KERNEL: Total Store Instructions = 2
◦ # KERNEL: Total Load Instructions Double = 1
◦ # KERNEL: Total Store Instructions Double = 1
◦ # KERNEL: Total ALU Instructions = 5
◦ # KERNEL: Total Control Instructions = 2
◦ # KERNEL: =====

```

Figure 23: Instruction Execution Analysis

### 3.3.3 Contact of registers

```

◦ # KERNEL: =====:
◦ # KERNEL: R1 = 1
◦ # KERNEL: R2 = 2
◦ # KERNEL: R3 = 4
◦ # KERNEL: R4 = 6
◦ # KERNEL: R5 = 5
◦ # KERNEL: R6 = 4
◦ # KERNEL: R7 = 100
◦ # KERNEL: R8 = 200
◦ # KERNEL: R9 = 455
◦ # KERNEL: R10 = 10
◦ # KERNEL: R11 = 11
◦ # KERNEL: R12 = 12
◦ # KERNEL: R13 = 13
◦ # KERNEL: R14 = 14
◦ # KERNEL: R15 = 52

```

Figure 24: Register values after program execution

### 3.3.4 Waveform

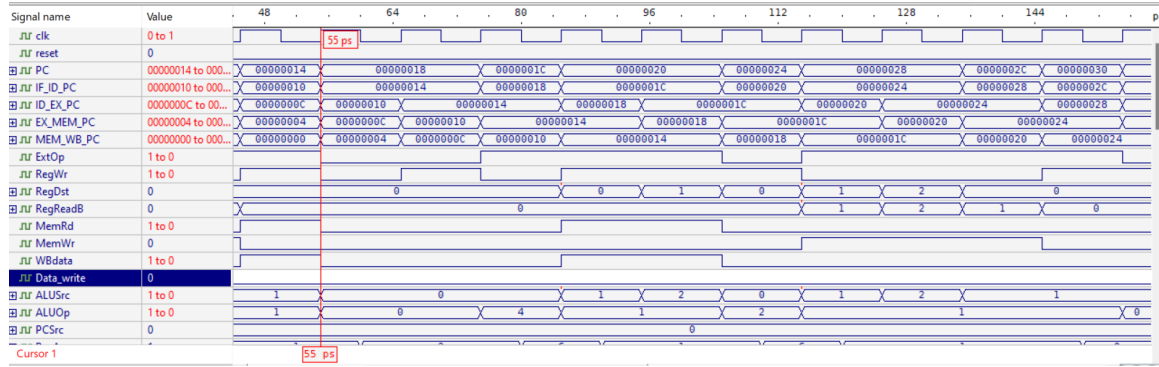


Figure 25:Test 2 waveform 1

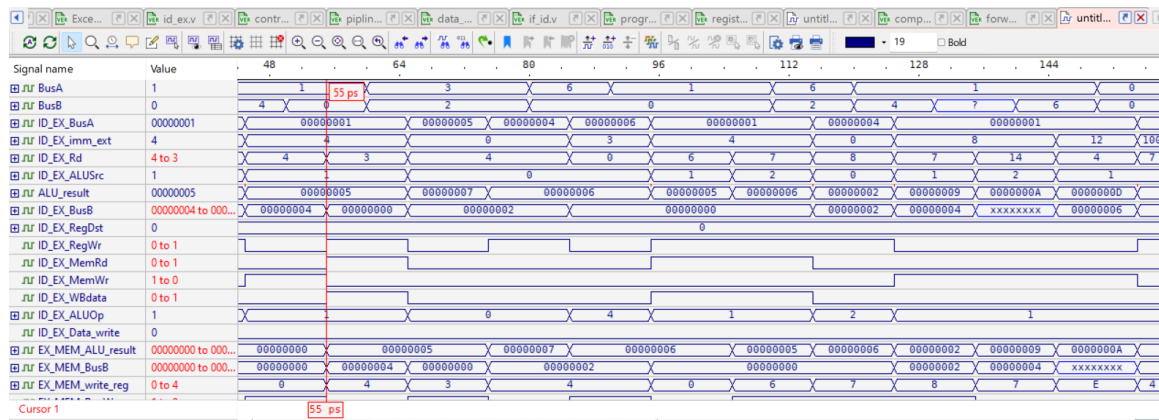


Figure 26:Test 2 waveform 2

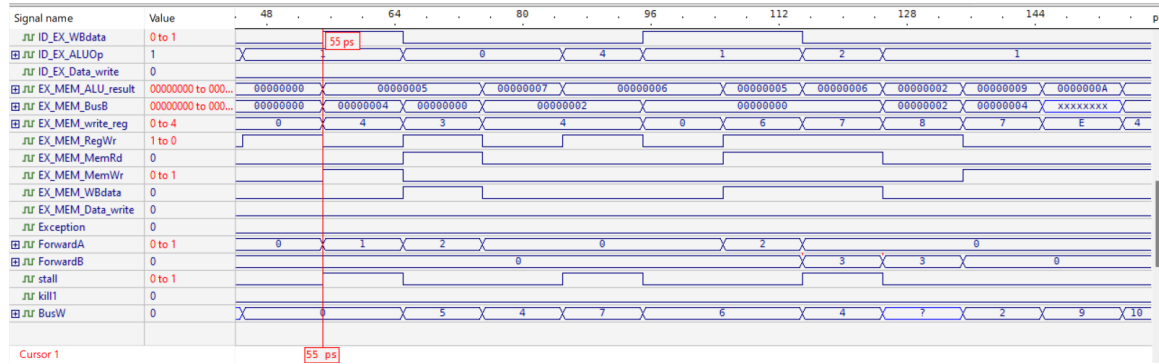


Figure 27:Test 2 waveform 3

## 4. Conclusion

This project successfully designed and built a complete 5-stage pipelined RISC processor using Verilog. The processor works as a real computer that can run programs and execute different types of instructions.

The processor supports 15 different instructions including basic arithmetic operations (ADD, SUB), logic operations (OR), memory operations (load and store), and control instructions (jumps and branches). All instructions use the same 32-bit format, making the design simple and efficient.

The main achievement was creating a pipelined processor that can work on multiple instructions at the same time. This pipeline has five stages: fetch, decode, execute, memory access, and write back. This design is much faster than a single-cycle processor because it can process several instructions simultaneously. To make the pipeline work correctly, we implemented important features like data forwarding to handle dependencies between instructions, hazard detection to prevent errors, and stalling when needed. The processor also handles special cases like double-word operations and includes proper exception handling.

The final result is a working processor that can execute complete programs efficiently. This project demonstrates our understanding of computer architecture principles and our ability to implement complex digital systems. It provides a strong foundation for future work in computer engineering and processor design.

## 5. Teamwork

Our team collaborated closely throughout the entire project, beginning with the datapath design. We used a shared Lucidchart file to collectively draft the architecture, ensuring that all members contributed equally to the design process. Each team member designed 5–6 instructions using RTN/RTL notation to build a solid foundation for implementation.

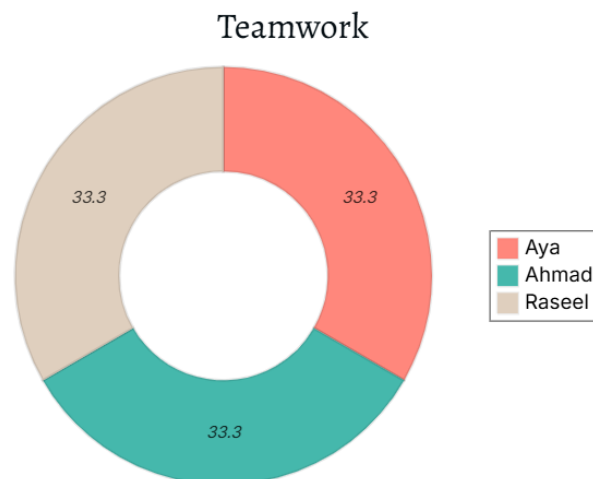
For the Verilog implementation:

- **Raseel** developed the Instruction Memory and Data Memory modules.
- **Aya** implemented the Control Unit and the Program Counter (PC).
- **Ahmad** designed the Register File and the ALU.

Initially, we implemented the system as a single-cycle datapath. After completing and verifying this stage, we worked together to refactor the design into a pipelined architecture, ensuring correct stage separation and data flow.

As a team, we jointly implemented and tested: the forwarding units the pipeline buffers the exception handler.

We also collaborated extensively on debugging and problem-solving, conducting thorough testing at each development phase to ensure functional correctness and performance. tested them together, and worked together on the debugging and solving the problems, The entire team collaborated on writing the report, we regularly held online meeting to track our work together.



*figure 28: Team Contribution Distribution.*