Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS4370

Computer Architecture. #Project 2

Design and Verification of a Simple Pipelined RISC Processor in Verilog

---

Team Work:

Saleh Shawer **No. :** 1220217.

Abdalraheem Shuaibi **No. :** 1220148.

Mousa Zahran **No. :** 1220716.

Instructor: Aziz Qaroush.

Section: 1.

Date: 8/6/2025.

Team Member Name	Team Member ID	Contributions
Abdalraheem Shuaibi	1220148	Built a fully functional pipeline processor using Logisim, contributed in base blocks implementation, Double LW/SW logic, Forwarding and stall detecting Write and simulate the processor in HDL using Verilog contributed in connecting the data path and implement base blocks logic Write test cases and report with explanation for them Write the RTL for the instructions
Saleh Shawer	1220217	Built a fully functional pipeline processor using Logisim, Contributed in base blocks implementation, all the PC logic, Branches and Stalls/Kills Write and simulate the processor in HDL using Verilog Contributed in test the program and detecting errors then fix them Help explaining data path in the report
Mousa Zahran	1220716.	Built a fully functional pipeline processor using Logisim, contributed in base blocks implementation and control signals Write and simulate the processor in HDL using Verilog contributed in implement base blocks logic Write the report structure, exaplaining data path, Control signals and diagrams/tables, write an abstract and conclution, help in writing the RTL

Processor Implementation (Tick One)	
Single Cycle	
Multi Cycle	
Pipelined	Tick tick (:

In case of pipeline implementation (Tick the correct answer)		
	Implemented in RTL Code?	**Verified and Correctly Worked?**
Data hazards detection	YA	YA
Control hazards detection	YA	YA
Structural hazards detection	YA	YA
Forwarding	YA	YA
Stalling	YA	YA

Tick the correct answer						
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?**	
OR Rd, Rs, Rt	Yes	No	Yes	No	Yes	No
ADD Rd, Rs, Rt	YES		YES		YES	
SUB Rd, Rs, Rt	YES		YES		YES	
CMP Rd, Rs, Rt	YES		YES		YES	
ORI Rd, Rs, Imm	YES		YES		YES	
ADDI Rd, Rs, Imm	YES		YES		YES	
LW Rd, Imm(Rs)	YES		YES		YES	
LDW Rd, Imm(Rs)	YES		YES		YES	
SDW Rd, Imm(Rs)	YES		YES		YES	
BZ Rs, Label	YES		YES		YES	
BGZ Rs, Label	YES		YES		YES	
BLZ Rs, Label	YES		YES		YES	
JR Rs	YES		YES		YES	
J Label	YES		YES		YES	
CALL Label	YES		YES		YES	
Tick one of the following						

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)	YA

Abstract

This project focuses on the design and implementation of a 32-bit pipelined RISC processor using Verilog HDL and Logisim. The processor supports three types of instructions: R-type, I-type, and J-type, and is structured around a five-stage pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The design integrates a complete datapath, a Main Control Unit, ALU Control logic, a 32-register file, and instruction memory modules. To ensure correct pipeline behavior, hazard detection mechanisms were implemented using stall and kill signals to handle control and data hazards. Functionality was verified using testbenches and simulation programs covering a wide range of instruction scenarios by logisim test and simulating in verilog. The results demonstrate correct execution, effective hazard handling, and adherence to the designed instruction set.

Table of Content Contents

Table of figure

Table of table

implementation & Design

Introduction

The objective of this project is to design and verify a simple 32-bit pipelined RISC processor using Verilog HDL and Logisim. These instruction formats allow for arithmetic, logic, memory access, and control flow operations, forming the core functionality of the processor. These instructions are the foundation of a RISC (Reduced Instruction Set Computing) processor, which is known for being simple and efficient. The processor we designed uses a five-stage pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Using pipelining allows the processor to run multiple instructions at the same time, with each one in a different stage. This improves the overall speed and performance. But with pipelining, some problems can happen, like data hazards, control hazards, or the need to stall the pipeline to avoid errors. To solve these issues, we built a complete datapath and control unit, and we also added hazard detection and forwarding logic. These parts help keep the pipeline running correctly and smoothly. The project was done in several steps. First, we designed the datapath and control logic by logisim. Then, we implemented the processor using Verilog HDL. Finally, we tested the design using simulations in Active-HDL and Logisim, with special test programs to make sure the instructions work correctly and the pipeline behaves as expected.

Instruction Format and RTL

Table 1: Instruction Format.

Opcode(6_bits)	Rd(4_bits)	Rs(4_bits)	Rt(bits)	Imm(14_bits)

The format supports 15 key instructions:

- **OR Rd, Rs, Rt**

Performs bitwise OR between Reg(Rs) and Reg(Rt)

→ Reg(Rd) = Reg(Rs) | Reg(Rt)

Opcode = 0

- **ADD Rd, Rs, Rt**

Performs integer addition

→ Reg(Rd) = Reg(Rs) + Reg(Rt)

Opcode = 1

- **SUB Rd, Rs, Rt**

Performs integer subtraction

→ Reg(Rd) = Reg(Rs) - Reg(Rt)

Opcode = 2

- **CMP Rd, Rs, Rt**

Compares the two source registers and sets Reg(Rd) based on the result:

Reg(Rd) = 0 if Reg(Rs) == Reg(Rt)

= -1 if Reg(Rs) < Reg(Rt)

= 1 if Reg(Rs) > Reg(Rt)

Opcode = 3

- **ORI Rd, Rs, Imm**

Performs bitwise OR between Reg(Rs) and the immediate value

→ Reg(Rd) = Reg(Rs) | Imm

Opcode = 4

- **ADDI Rd, Rs, Imm**

Performs integer addition between Reg(Rs) and the immediate

→ Reg(Rd) = Reg(Rs) + Imm

Opcode = 5

- **LW Rd, Offset(Rs)**

Loads a word from memory at address Reg(Rs) + Offset

→ Reg(Rd) = Mem[Reg(Rs) + Offset]

Opcode = 6

- **SW Rd, Offset(Rs)**

Stores the value in Reg(Rd) into memory at address Reg(Rs) + Offset

→ Mem[Reg(Rs) + Offset] = Reg(Rd)

Opcode = 7

- **LDW Rd, Offset(Rs)**

Loads a word using double-word addressing

Opcode = 8

- **SDW Rd, Offset(Rs)**

Stores a word using double-word addressing

Opcode = 9

- **BZ Rs, Label**

Branches to Label if Reg(Rs) == 0

Opcode = 10

- **BGZ Rs, Label**

Branches to Label if Reg(Rs) > 0

Opcode = 11

- **BLZ Rs, Label**

Branches to Label if Reg(Rs) < 0

Opcode = 12

- **JR Rs**

Jumps to the address stored in Reg(Rs)

Opcode = 13

- **JUMP Label**

Unconditional jump to Label

Opcode = 14

- **CALL Label**

Jumps to Label and stores return address , Opcode = 15

- **RTL Of Instructions:**
  - **OR, ADD, SUB, CMP**
    1. **Fetch Stage:**

       Instruction ← ROM[PC]

       PC ← Stall(PC + 1, PC)
    2. **Decode Stage:**

       Data1 ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

       Data2 ← FWB(Reg[Rt], ALU_FW, MEM_FW, WB_FW)

FW is determined by the Forward and Stall Control Unit

3. **Execute Stage:**

   Result ← ALU_OP(Data1, Data2)

   ALU_FW ← Result

   RD2 ← Rd

   ALU_OP is determined by the opcode, (Main Control Unit)

4. **Memory Stage:**

   MEM_FW ← Mem out

   RD3 ← RD2

5. **Write Back Stage:**

   Reg[Rd] ← Result

   WB_FW ← Result

   RD4 ← RD3

   RW ← RD4

- **ORI, ADDI**

  1. **Fetch Stage:**

  Instruction ← ROM[PC]

  PC ← Stall(PC + 1, PC)

  2. **Decode Stage:**

  Data1 ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

  Imm ← (Sign_Extend | Unsign_Ext) (Immediate)

  Data2 ← Imm

  FW is determined by the Forward and Stall Control Unit

  3. **Execute Stage:**

  Result ← ALU_OP(Data1, Data2)

  ALU_FW ← Result

  RD2 ← Rd

  ALU_OP is determined by the opcode, (Main Control Unit)

  4. **Memory Stage:**

  MEM_FW ← Mem out

  RD3 ← RD2

5. **Write Back Stage:**

   Reg[Rd] ← Result

   WB_FW ← Result

   RD4 ← RD3

   RW ← RD4

- **LW (Load Word)**
  1. **Fetch Stage:**

     Instruction ← ROM[PC]

     PC ← Stall (PC + 1, PC)

  2. **Decode Stage:**

     Offset ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

     Imm ← Sign_Extend(Immediate)

     MEM_Data ← Imm

     FW is determined by the Forward and Stall Control Unit

  3. **Execute Stage:**

     EA ← ALU_OP(Data1, Data2)

     ALU_FW ← EA

     RD2 ← Rd

     ALU_OP is determined by the opcode, (Main Control Unit)

  4. **Memory Stage:**

     MEM_FW ← Mem out

     RD3 ← RD2

  5. **Write Back Stage:**

     Reg[Rd] ← Mem out

     WB_FW ← Result

     RD4 ← RD3

     RW ← RD4

- **SW (Store Word)**
  1. **Fetch Stage:**

     Instruction ← ROM[PC]

     PC ← Stall(PC + 1, PC)

2. **Decode Stage:**

   Offset ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

   Imm ← Sign_Extend(Immediate)

   MEM_Data ← Imm

   FW is determined by the Forward and Stall Control Unit

3. **Execute Stage:**

   EA ← ALU_OP(Data1, Data2)

   ALU_FW ← EA

   RD2 ← Rd

   ALU_OP is determined by the opcode, (Main Control Unit)

4. **Memory Stage:**

   MEM_FW ← Mem out

   RD3 ← RD2

5. **Write Back Stage:**

   No stage

- **LDW (Load Double Word)**

1. **Fetch Stage:**

   Instruction ← ROM[PC]

   PC ← Add_PC(Stall(PC + 1, PC), PC)

2. **Decode Stage:**

   Offset ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

   Imm ← Sign_Extend(Add_Imm(Immediate, Immediate + 1))

   MEM_Data ← Imm

   Rd ← Add_RD(Rd, Rd + 1)

   FW is determined by the Forward and Stall Control Unit

3. **Execute Stage:**

   EA ← ALU_OP(Data1, Data2)

   ALU_FW ← EA

   RD2 ← Rd

   ALU_OP is determined by the opcode, (Main Control Unit)

4. **Memory Stage:**

   MEM_FW ← Mem out

   RD3 ← RD2

5. **Write Back Stage:**

   Reg[Rd] ← Mem out

   WB_FW ← Result

   RD4 ← RD3

   RW ← RD4

- **SDW (Store Double Word)**

1. **Fetch Stage:**

   Instruction ← ROM[PC]

   PC ← Add_PC(Stall(PC + 1, PC), PC)

2. **Decode Stage:**

   Offset ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

   Imm ← Sign_Extend(Add_Imm(Immediate, Immediate + 1))

   MEM_Data ← Imm

   Rd ← Add_RD(Rd, Rd + 1)

   FW is determined by the Forward and Stall Control Unit

3. **Execute Stage:**

   EA ← ALU_OP(Data1, Data2)

   ALU_FW ← EA

   RD2 ← Rd

   ALU_OP is determined by the opcode, (Main Control Unit)

4. **Memory Stage:**

   MEM_FW ← Mem out

   RD3 ← RD2

5. **Write Back Stage:**

   No stage

- **BZ, BGZ, BLZ**
  1. **Fetch Stage:**

     Instruction ← ROM[PC]

     PC ← Stall(PC + 1, PC)
  2. **Decode Stage:**

     RsVal ← Reg[Rs]

     Offset ← Sign_Extend(Immediate)

     RsVal ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

     If Cmp_Succ(RsVal) → PC ← PC + 1 + Offset

     Else → PC ← PC + 1
  3. **Execute Stage:**

     (No operation)
  4. **Memory Stage:**

     (No operation)
  5. **Write Back Stage:**

     (No operation)
- **JR (Jump to Register)**
  1. **Fetch Stage:**

     Instruction ← ROM[PC]

     PC ← PC + 1
  2. **Decode Stage:**

     JumpAddr ← FWA(Reg[Rs], ALU_FW, MEM_FW, WB_FW)

     PC ← JumpAddr
  3. **Execute Stage:**

     (No operation)
  4. **Memory Stage:**

     (No operation)
  5. **Write Back Stage:**

     (No operation)

- **J (Unconditional Jump)**

1. **Fetch Stage:**

   Instruction ← ROM[PC]

   PC ← PC + 1

2. **Decode Stage:**

   JumpAddr ← Sign_Extend(Immediate)

   PC ← PC + 1 + JumpAddr

3. **Execute Stage:**

   (No operation)

4. **Memory Stage:**

   (No operation)

5. **Write Back Stage:**

   (No operation)

- **CLL (Call)**

1. **Fetch Stage:**

   Instruction ← ROM[PC]

   PC ← PC + 1

2. **Decode Stage:**

   JumpAddr ← Sign_Extend(Immediate)

   RetAddr ← PC

   Reg[R14] ← PC

   PC ← PC + 1 + JumpAddr

3. **Execute Stage:**

   (No operation)

4. **Memory Stage:**

   (No operation)

5. **Write Back Stage:**

   (No operation)

Data Path

Design of Data Path

The 32-bit pipelined RISC processor in this project is built around two essential parts: the datapath and the control path. Both parts work together to make sure instructions are executed correctly through all five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The datapath handles the flow of data between different units, while the control path generates the necessary control signals to guide the operation of each module. In this section, we explain how the processor is designed by describing the key components of both the datapath and the control unit. We also highlight how each module plays a role in maintaining proper instruction flow and synchronization within the pipeline.

The full of CPU pipelined datapath design is shown below:

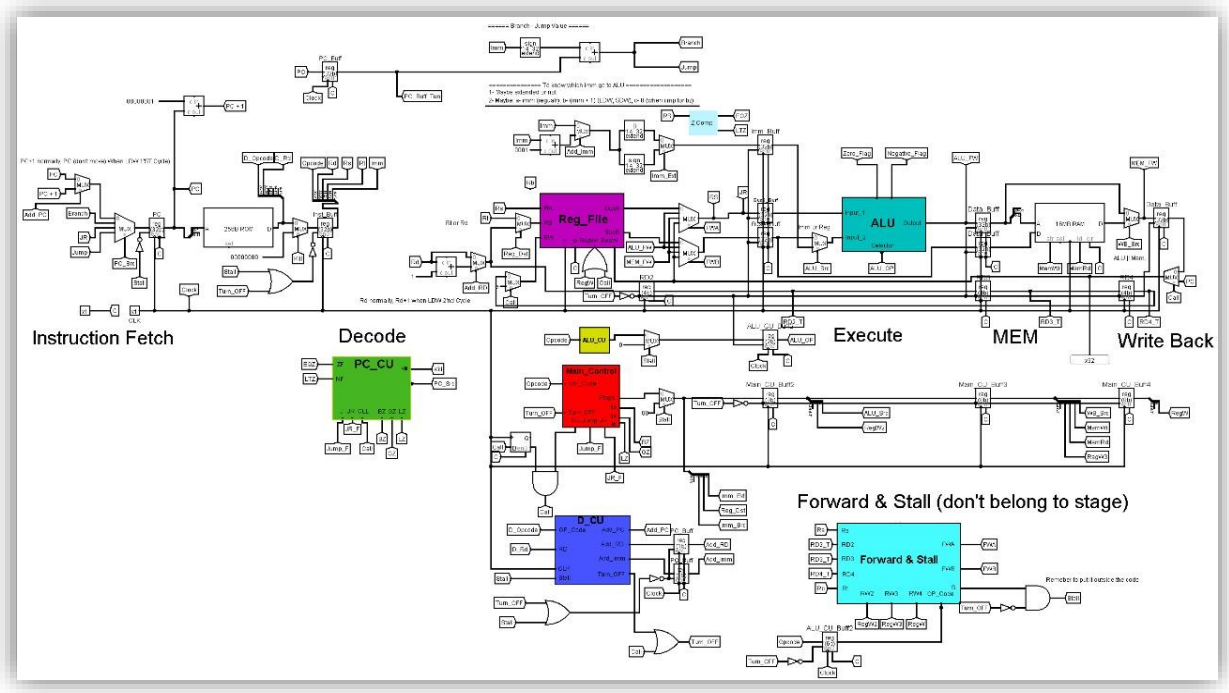

Figure 1: Full Data Path & Control Units.

As illustrated in the figure above, the processor follows a five-stage pipeline structure consisting of: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage is responsible for a specific part of instruction execution, allowing

multiple instructions to be processed simultaneously for improved performance.

1) Instruction Fetch (IF)

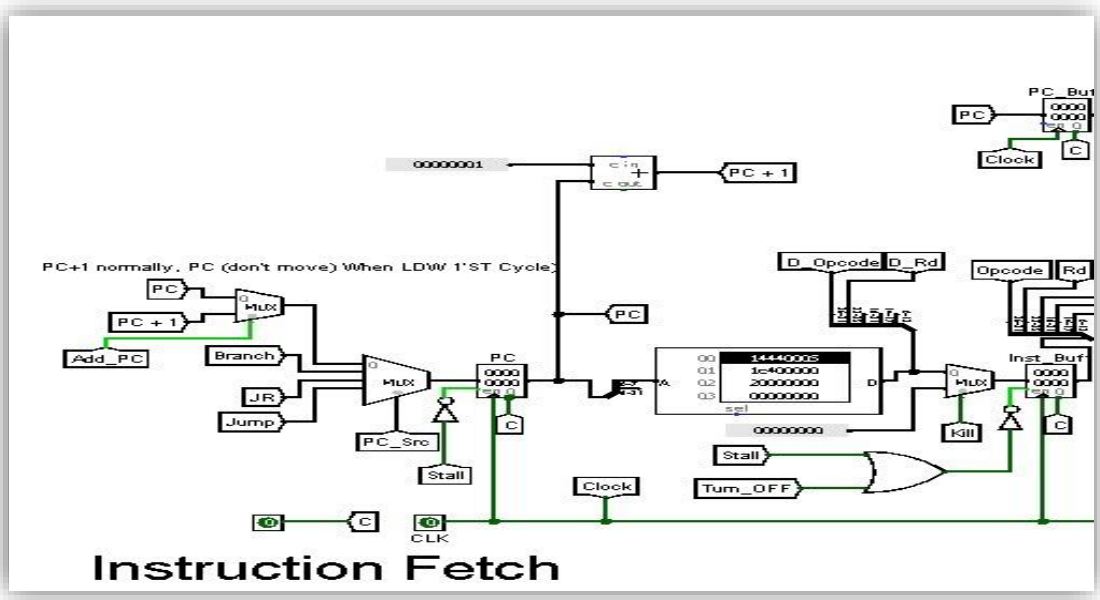

Figure 2: Instruction fetch (IF) stage.

In this stage, the processor retrieves the instruction to be executed from the instruction memory (ROM). The input to the ROM is the current value of the Program Counter (PC), which determines the address of the instruction to fetch. The ROM uses this address to return the corresponding instruction. The output of this stage includes the fetched instruction and the updated PC value, determined through a multiplexer that selects the next address based on a 2-bit PC_Src control signal. This selection allows one of the following:

- PC + 4 (sequential execution) → when PC_Src = 0
- PC + 4 + Imm (for branches) → when PC_Src = 1
- a jump to a register value (for JR instructions) → when PC_Src = 2
- or a jump to a specific address → when PC_Src = 3

This stage plays a critical role in maintaining the continuous flow of instructions through the pipeline.

2) Instruction Decode (ID)

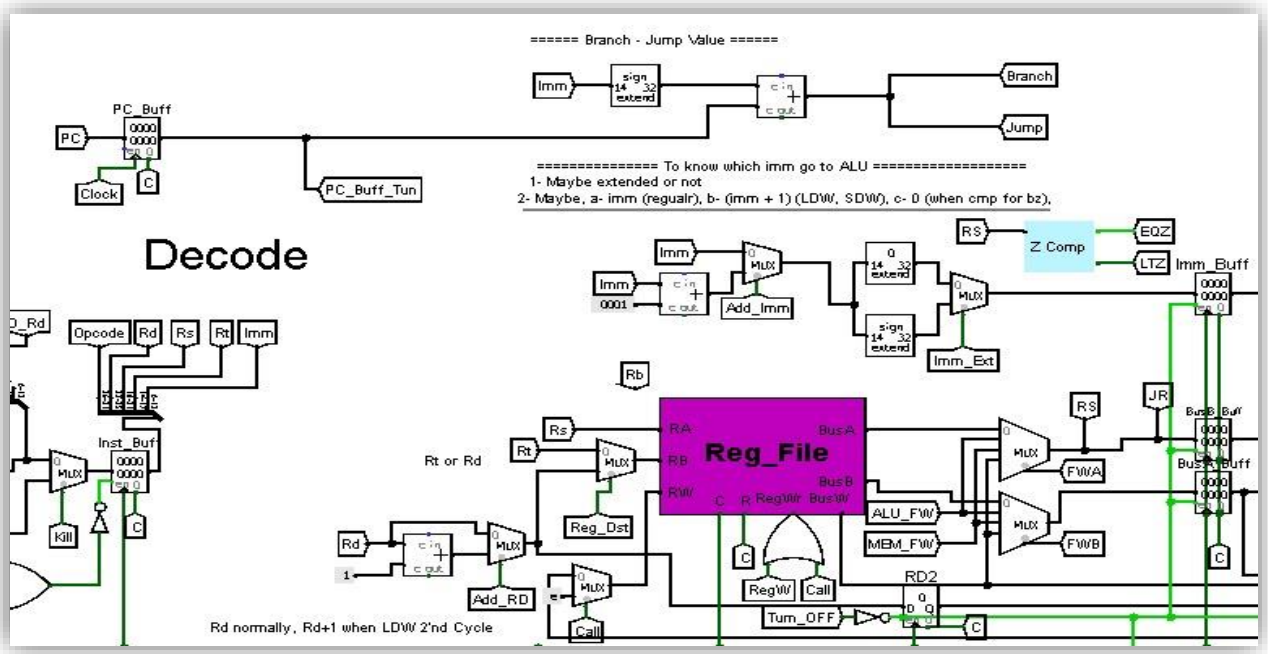

Figure 3: Instruction Decode1 (ID) stage.

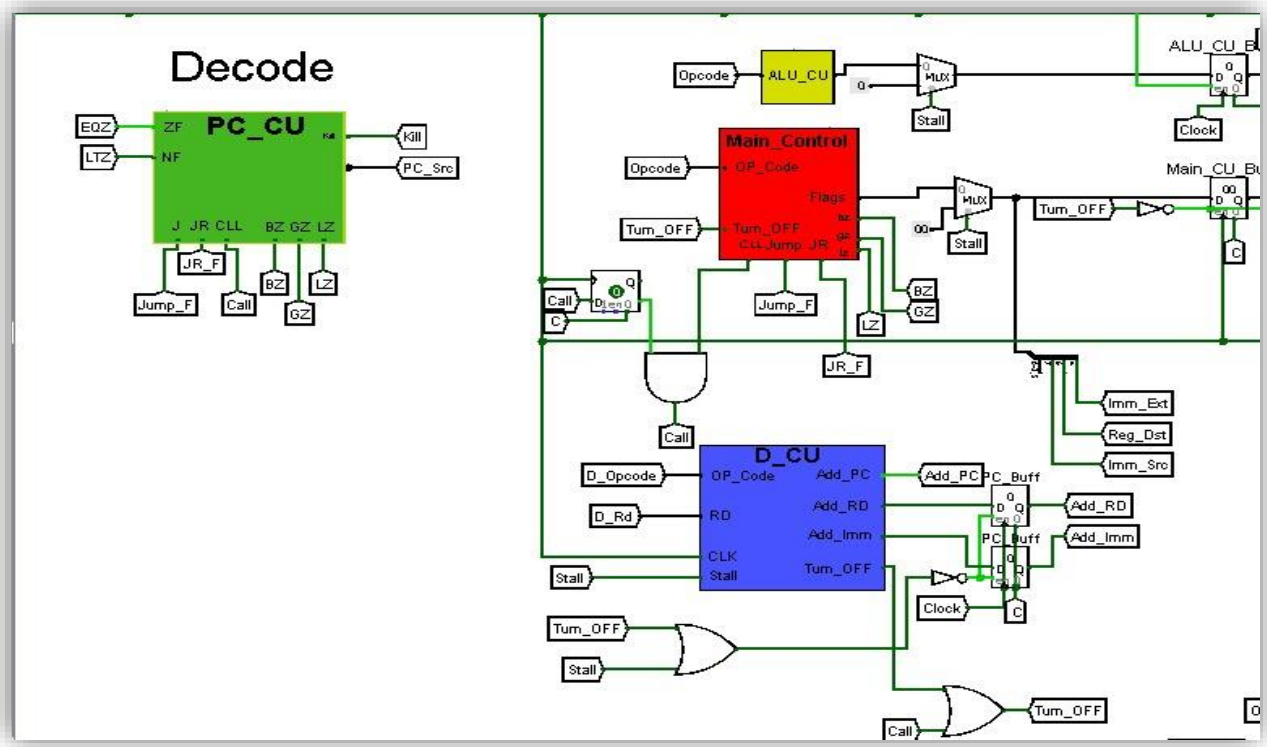

Figure 4: Instruction Decode2 (ID) stage.

In this stage, the instruction fetched from memory is decoded to determine its type whether it is an R-type, I-type, or J-type and to extract its relevant fields such as the source registers (Rs, Rt), destination register (Rd), opcode, and immediate value. The extracted opcode is passed to the Main Control Unit, which generates the appropriate control signals for the next pipeline stages. Based on the opcode, the processor identifies the instruction type. If the instruction is a branch or jump, the branch or jump target address is calculated at this point (as shown in Figure 4). In addition, multiple control signals are determined during this stage if the stall not show (**stall = 0**), including:

- Register Destination (**RegDst**)
- Register Write Enable (**RegWr**)
- ALU Source (**ALUSrc**)
- ALU Operation Code (**ALU_OP**)
- Memory Read Enable (**MemRd**)
- Memory Write Enable (**MemWr**)
- Write Back Data Source (**WBdata**)

Special control flags such as Jump, Call, Jump Register (JR), Branch if Equal (BZ), Branch if Greater Than Zero (BGZ), and Branch if Less Than Zero (BLZ) are also generated depending on the instruction.

At this stage, the register file reads the required operands using the addresses from Rs, Rt, or Rd based on the instruction format. These values are prepared to be forwarded to the Execute (EX) stage.

Furthermore, this stage includes hazard detection logic. If a data or control hazard is detected, the pipeline will respond by inserting a stall or kill signal to prevent incorrect execution. When applicable, forwarding techniques are used to resolve dependencies and keep the pipeline flowing without interruption.

3) Execution (Ex)

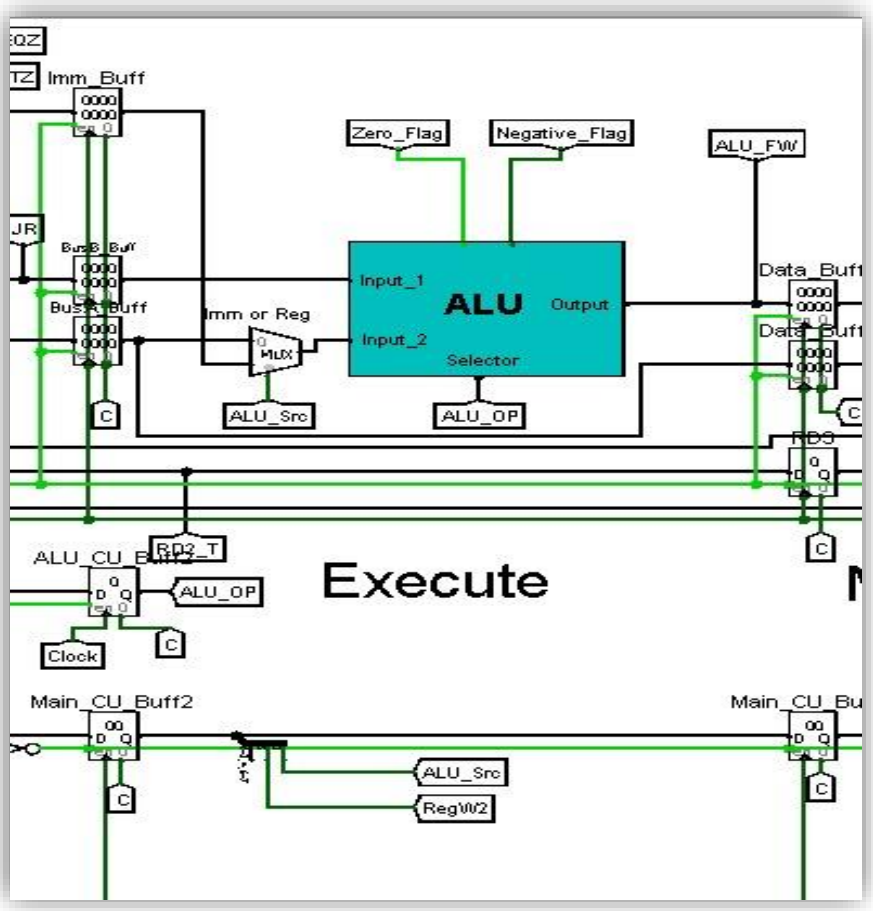

Figure 5: Execution stage.

In the Execute stage, the processor performs the core operation of the instruction. This is where the ALU (Arithmetic Logic Unit) is used to execute arithmetic or logical operations based on the instruction type and control signals generated in the Decode stage. As shown in the figure, the ALU receives two inputs:

- **Input 1 is typically from register Rs.**
- **Input 2 is selected using the control signal ALU_Src:**

  If **ALU_Src = 0:** the second operand is from register Rt.

  If **ALU_Src = 1:** the second operand is the immediate value (after sign-extension).

These operands are passed through multiple buffers (like BusA_Buff, Imm_Buff, BusB_Buff) to synchronize the pipeline stage. The control signal ALU_OP, generated from the Control Unit

and passed through the ALU_CU_Buff, selects the operation (e.g., addition, subtraction, comparison, OR) that the ALU must perform. The result of this operation is stored in the Data_Buff. The ALU also produces status flags:

- **Zero_Flag (ZF)** is set when the result is zero.
- **Negative_Flag (NF)** is set when the result is negative.

These flags are used in the Program Counter Control Unit (PC_CU) to evaluate conditional branches. This stage may also include forwarding logic (ALU_FW) to handle data hazards, allowing the ALU to use the most recent value even if it has not yet been written back to the register file.

4) Memory and write back Stages

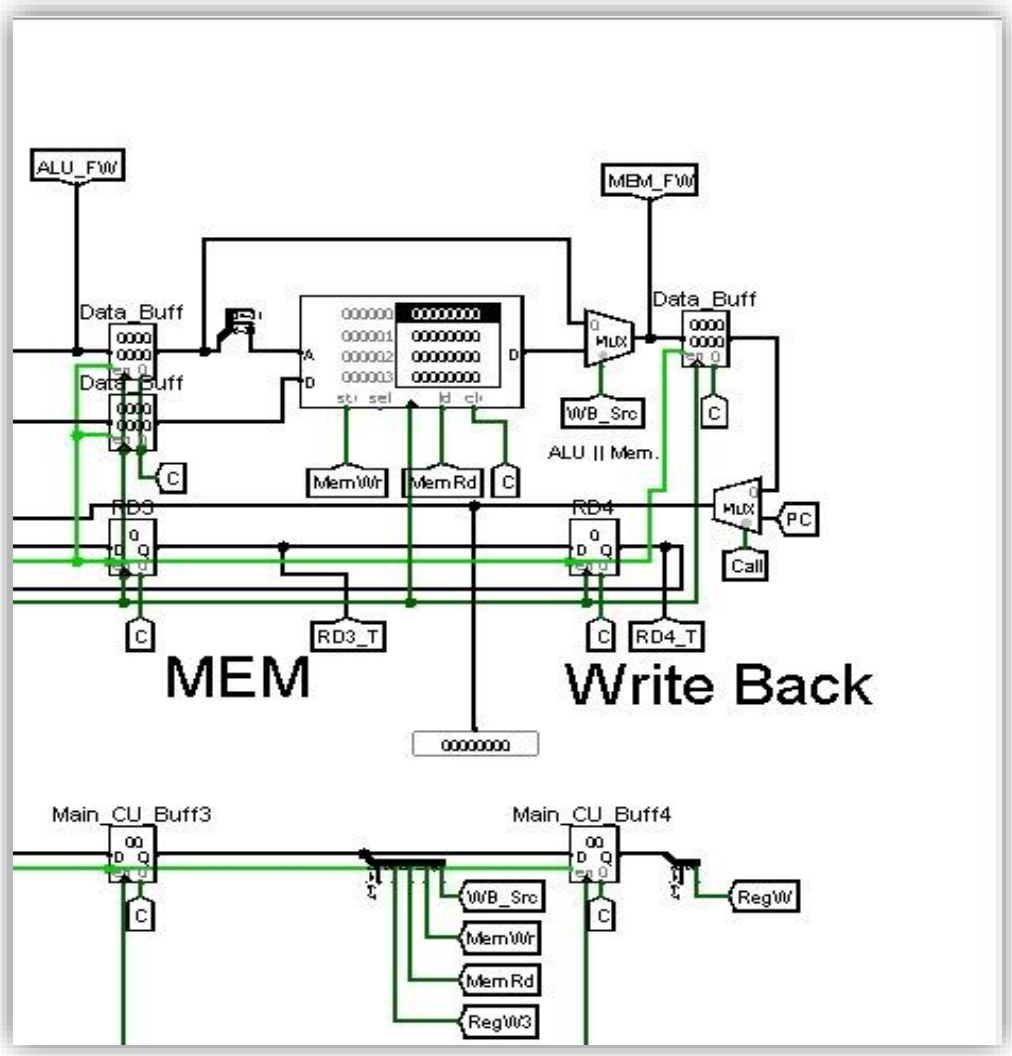

Figure 6: Memory and write back Stages.

Memory Access (MEM) Stage:

In this stage, the processor interacts with data memory (RAM). Even though all instructions flow through this stage, only load and store instructions actually perform memory operations:

- **Load (LW, LDW):** If the control signal MemRd is active (MemRd = 1), the processor reads data from memory at the address provided by the ALU result. This data is saved in a buffer (Data_Buff) to be sent to the Write Back stage.
- **Store (SW, SDW):** If the control signal MemWr is active (MemWr = 1), the processor writes data to memory. The address is again determined by the ALU output, and the data to be written comes from the register value (often from BusB).

This stage is also connected to forwarding logic (MBM_FW) to handle hazards and ensure up-to-date values are used when needed.

Write Back (WB) Stage:

This is the final pipeline stage where the result of the instruction is written back into the register file. If the signal RegW (Register Write) is set to 1, the processor writes the result into the register specified by the control path.

The value to write can be either:

ALU result (for arithmetic instructions).

Memory read data (for load instructions).

5) Forward and Stall

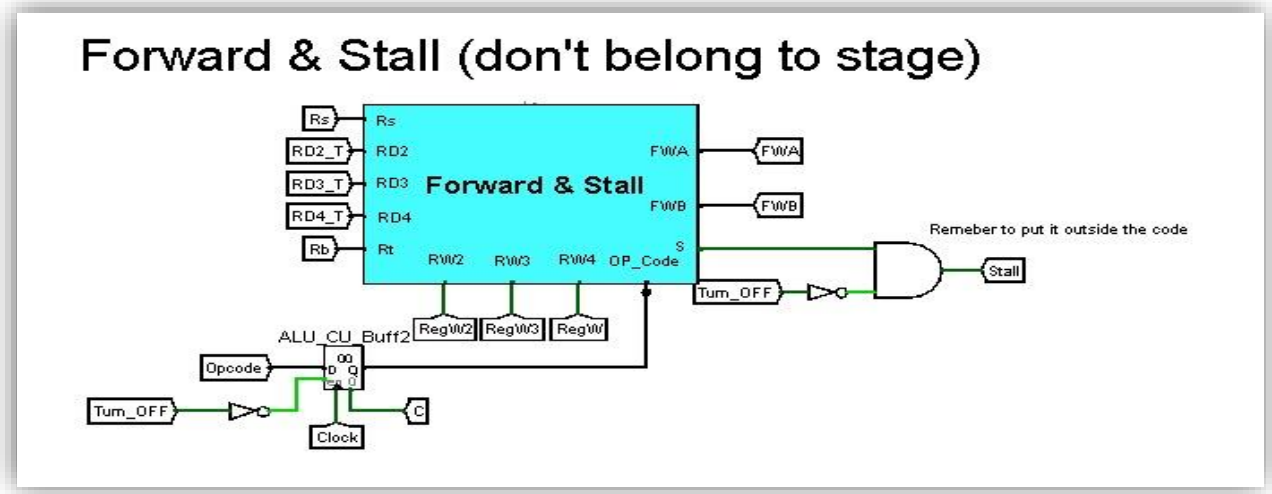

Figure 7: forward & Stall.

This unit is used to keep the pipeline working correctly and without mistakes. It checks if the current instruction needs data from a previous instruction that didn't finish yet. It does that by comparing the source registers (Rs, Rt) with the destination registers from the later pipeline stages (RD2, RD3, RD4). If the data is ready, it will use forwarding, and the signals FWA and FWB will be active to send the needed value directly to the ALU without waiting. But if the data is not ready yet, like after a load instruction, then the unit will activate the Stall signal to stop the pipeline for one cycle until the data is ready. This helps the processor stay correct while still trying to be fast.

Control Signals

Design of Control Signals:

The control path is what tells the datapath what to do at every step. It sends the right control signals to handle things like writing to registers, reading or writing from memory, and deciding when and where to branch or jump. In this project, the control path is split into three main parts:

- **PC Control**: Controls the program counter and jump/branch logic.
- **Main Control**: Generates most of the control signals based on the instruction type.
- **ALU Control**: Decides what operation the ALU should perform based on the instruction.

1) PC Control Signal

The PC Control unit is responsible for deciding where the next instruction should come from. It controls how the Program Counter (PC) is updated whether to just go to the next instruction (PC + 4), or jump, or take a branch. This unit takes inputs like flags and control signals (Jump, Branch, JR) and based on that, chooses the correct next address for the PC.

A summary of the inputs and outputs of this unit is shown in Table , explaining how each instruction type affects the update of the PC.

Table 2: PC Control Signals.

Instruction	Inputs								Outputs	
	ZF	NF	Jump_F	JR_F	Call	BZ	GZ	LZ	kill	PC_Source
OR Rd, Rs, Rt	x	x	0	0	0	0	0	0	0	00
ADD Rd, Rs, Rt	x	x	0	0	0	0	0	0	0	00
SUB Rd, Rs, Rt	x	x	0	0	0	0	0	0	0	00
CMP Rd, Rs, Rt	x	x	0	0	0	0	0	0	0	00
ORI Rd, Rs, Imm	x	x	0	0	0	0	0	0	0	00
ADDI Rd, Rs, Imm	x	x	0	0	0	0	0	0	0	00
LW Rd, Imm(Rs)	x	x	0	0	0	0	0	0	0	00
SW Rd, Imm(Rs)	x	x	0	0	0	0	0	0	0	00
LDW Rd, Imm(Rs)	x	x	0	0	0	0	0	0	0	00
SDW Rd, Imm(Rs)	x	x	0	0	0	0	0	0	0	00
BZ Rs, Label	1	0	0	0	0	1	0	0	1	01
	0	x	0	0	0	1	0	0	0	00
BGZ Rs, Label	0	0	0	0	0	0	1	0	1	01
	x	1	0	0	0	0	1	0	0	00
	1	x	0	0	0	0	1	0	0	00
BLZ Rs, Label	0	0	0	0	0	0	1	0	1	01
	x	1	0	0	0	0	1	0	0	00
JR Rs	x	x	0	1	0	0	0	0	0	10
J Label	x	x	1	0	0	0	0	0	0	11
CLL Label	x	x	0	0	1	0	0	0	0	11

As shown in Figure 7, this is the implementation of the PC control logic, which decides if a normal PC increment should happen or if the program should jump or branch. There are four main conditions handled in this logic:

- BZ_True is activated when the instruction is a branch if zero and the zero flag (ZF) is set: **BZ_True = BZ & ZF**

- BGZ_True is for branch if greater than zero, which happens when ZF is 0 and NF is 0:
  BGZ_True = GZ & ~ZF & ~NF
- BLZ_True handles the branch if less than zero, so it's active when the negative flag (NF) is set:
  BLZ_True = LZ & NF

These are combined into one signal:

Branch_True = BZ_True | BGZ_True | BLZ_True

For jump-related instructions, we use:

J_True = Jump_F | CLL

Finally, the Kill signal is used to flush the pipeline in case of control changes (jump, call, or branch):

Kill = JR_F or J_True or Branch_True

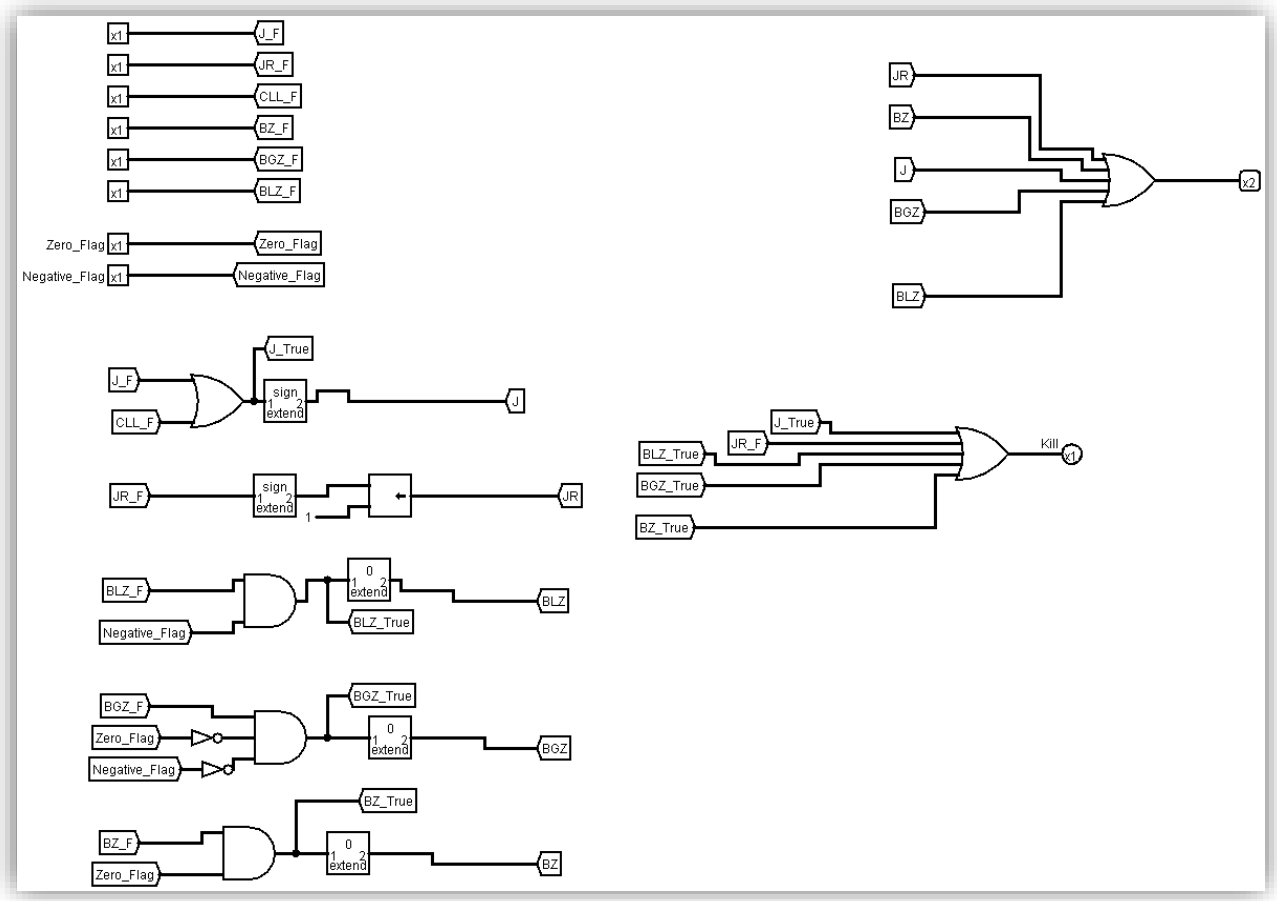

Figure 8: PC control unit implementation.

2) Main Control Signals

The Main Control Unit is responsible for producing all the essential control signals that guide the processor's operation. It manages register selection, enables or disables memory read/write, determines the ALU source and operation, and controls the write-back process.
A detailed breakdown of the control signals and how they behave for each instruction type is shown in Table 5 and 6.
This Table 5 show the value of Flags(selection of multiplexer).

Table 3:Main Control Signals(Flags).

Instruction	Inputs	Outputs						
	Opcode	Reg_ Dst	RegW	Imm_ Ext	ALU_Src	MemRd	MemWr	WB_Src
OR Rd, Rs, Rt	0	0	1	0	0	0	0	0
ADD Rd, Rs, Rt	1	0	1	0	0	0	0	0
SUB Rd, Rs, Rt	2	0	1	0	0	0	0	0
CMP Rd, Rs, Rt	3	0	1	0	0	0	0	0
ORI Rd, Rs, Imm	4	0	1	1	1	0	0	0
ADDI Rd, Rs, Imm	5	0	1	1	1	0	0	0
LW Rd, Imm(Rs)	6	0	1	1	1	1	0	1
SW Rd, Imm(Rs)	7	x	0	1	1	0	1	x
LDW Rd, Imm(Rs)	8	0	1	1	1	1	0	1
SDW Rd, Imm(Rs)	9	x	0	1	1	0	1	x
BZ Rs, Label	10	x	0	1	1	0	0	x

BGZ Rs, Label	**11**	**x**	**0**	**1**	**1**	**0**	**0**	**x**
BLZ Rs, Label	**12**	**x**	**0**	**1**	**1**	**0**	**0**	**x**
JR Rs	**13**	**x**	**0**	**0**	**0**	**0**	**0**	**x**
J Label	**0**	**x**	**0**	**0**	**0**	**0**	**0**	**x**
CLL Label	**0**	**x**	**0**	**0**	**0**	**0**	**0**	**x**

Table 4: Main Control Signals.

Instruction	**Inputs**	**Outputs**					
	Opcode	**Call**	**Jump_F**	**JR_F**	**LZ**	**GZ**	**BZ**
OR Rd, Rs, Rt	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ADD Rd, Rs, Rt	**1**	**0**	**0**	**0**	**0**	**0**	**0**
SUB Rd, Rs, Rt	**2**	**0**	**0**	**0**	**0**	**0**	**0**
CMP Rd, Rs, Rt	**3**	**0**	**0**	**0**	**0**	**0**	**0**
ORI Rd, Rs, Imm	**4**	**0**	**0**	**0**	**0**	**0**	**0**
ADDI Rd, Rs, Imm	**5**	**0**	**0**	**0**	**0**	**0**	**0**
LW Rd, Imm(Rs)	**6**	**0**	**0**	**0**	**0**	**0**	**0**
SW Rd, Imm(Rs)	**7**	**0**	**0**	**0**	**0**	**0**	**0**
LDW Rd, Imm(Rs)	**8**	**0**	**0**	**0**	**0**	**0**	**0**
SDW Rd, Imm(Rs)	**9**	**0**	**0**	**0**	**0**	**0**	**0**
BZ Rs, Label	**10**	**0**	**0**	**0**	**0**	**0**	**1**
BGZ Rs, Label	**11**	**0**	**0**	**0**	**0**	**1**	**0**
BLZ Rs, Label	**12**	**0**	**0**	**0**	**1**	**0**	**0**
JR Rs	**13**	**0**	**0**	**1**	**0**	**0**	**0**
J Label	**14**	**0**	**1**	**0**	**0**	**0**	**0**
CLL Label	**15**	**1**	**0**	**0**	**0**	**0**	**0**

As shown in Figure 8, this diagram represents the Main Control Unit (MCU) implementation. Its

role is to decode the opcode of the current instruction and generate the necessary control signals to coordinate the processor's operations during the execution stage. The unit ensures that the correct flags are activated based on the instruction type, guiding data flow and operations across the processor.

- **Turn_OFF Signal:**

  When set to 1, this signal acts as a safety mechanism, raising an exception and disabling all control flags. This halts further operations, typically used for error handling.

- **Opcode Inputs:**

  Opcode x6: These represent segments of the opcode being decoded. The MCU uses these segments to determine the instruction type (e.g., Call, Jump_F, JR_F, LZ, GZ, BZ) and generate the appropriate control signals.

Condition and Flags:

- **Call**

  Call = 1 → when Opcode = 12 & Turn_OFF = 0.

- **JR_F**

  JR_F = 1 → when Opcode = 13 & Turn_OFF = 0.

- **Jump_F**

  Jump_F = 1 → when Opcode = 14 & Turn_OFF = 0.

- **LZ (Less Than Zero):** Activated if the ALU result is negative.

  LZ = 1 → when Opcode = 12 & Turn_OFF = 0.

- **GZ (Greater Than Zero):** Activated if the ALU result is positive.

  GZ = 1 → when Opcode = 11 & Turn_OFF = 0.

- **BZ (Branch Zero):** Used for conditional branching.

  BZ = 1 → when Opcode = 10 & Turn_OFF = 0.

Flags:

- **Reg_Dst:** Selects the destination register for write-back.

  Reg_Dst to select the Rd or Rt to write to the register.

  If Reg_Dst = 0 → Rd write to the register.

  If Reg_Dst = 1→ Rt write to the register.

- **RegW:** Enables writing to the register file.

  RegW to select if write to the register or no.

If RegW = 0 → doesn't write to the register.

If RegW = 1→ write to the register.

- **Imm_Ext:** Controls sign-extension of immediate values.

  Imm_Ext to select if Extension to Immediate or no.

  If RegW = 0 → doesn't Extension to Immediate.

  If RegW = 1→ Extension to Immediate.

- **ALU_Src:** Selects between a register or immediate value as the ALU input.

  If ALU_Src = 0 → Select the register to second input in Execution.

  If ALU_Src = 0 → Select the Imm to second input in Execution.

- **MemRd:** Enables memory read operations.

  If MemRd = 0 → memory read operations disable.

  If MemRd = 1 → memory read operations enable.

- **MemWr:** Enables memory write operation

  If MemWr= 0 → memory write operations disable.

  If MemWr= 1 → memory write operations enable.

- **WB_Src:** Selects the source of data for write-back (e.g., ALU result or memory data).

  If WB_Src= 0 → write ALU result.

  If WB_Src= 1 → write memory data.

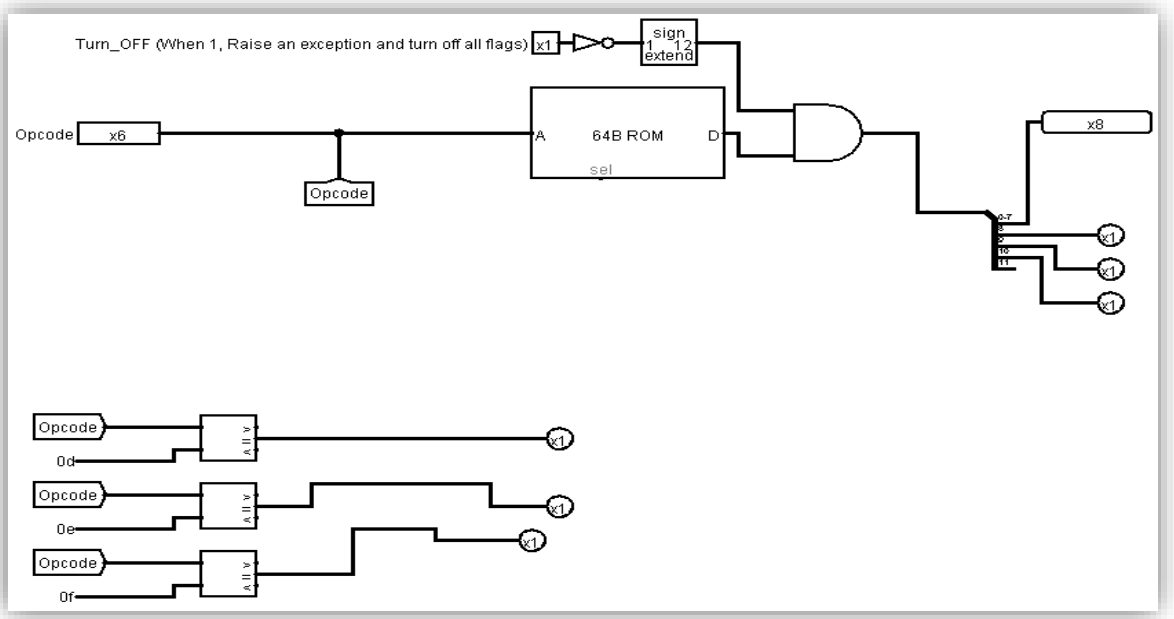

Figure 9: Main control unit implementation.

3) ALU Control Signals

The ALU Control unit is responsible for telling the ALU which operation to perform depending on the instruction. It looks at the opcode to decide if the ALU should do an addition, subtraction, comparison, or bitwise operation like OR. This unit sends a 2-bit signal (ALU_OP) to control the ALU behavior.

Table 7 shows how each opcode is mapped to the correct ALU operation.

Table 5: ALU Control Signals.

Instruction	Inputs	Outputs
	Opcode	ALU_OP
OR Rd, Rs, Rt	000000	00
ADD Rd, Rs, Rt	000001	01
SUB Rd, Rs, Rt	000010	11
CMP Rd, Rs, Rt	000011	11
ORI Rd, Rs, Imm	000100	00
ADDI Rd, Rs, Imm	000101	01
LW Rd, Imm(Rs)	000110	01
SW Rd, Imm(Rs)	000111	01
LDW Rd, Imm(Rs)	001000	01
SDW Rd, Imm(Rs)	001001	01
BZ Rs, Label	001010	11
BGZ Rs, Label	001011	11
BLZ Rs, Label	001100	11
JR Rs	001101	xx
J Label	001110	xx
CLL Label	001111	xx

As shown in Figure 9, this diagram represents the ALU Control Unit implementation. Its role is to generate the correct control signals to select the type of operation that the ALU should perform during the execution stage. The input to this unit is the opcode, which is decoded and passed through logic gates to decide the appropriate ALU_OP value.

This control logic can choose one of the four operations:

- ADD
- SUB
- OR
- CMP

Each operation is mapped to a specific combination of bits in the ALU_OP signal. The final selected operation is passed to the ALU, which then uses it to perform the required calculation.
The ALU Control Unit maps opcode segments to one of four operations, encoded as 2-bit ALU_OP signals:

- 2'b00 (OR):

  Activated for opcode values 5'd0 and 5'd4.

  ALU_OP = OR instruction | ORI instruction.

- 2'b01 (ADD):

  Activated for opcode values 5'd1, 5'd5, 5'd6, 5'd7, 5'd8, and 5'd9.

  ALU_OP = ADD instruction | ADDI instruction | ADD instruction | LW instruction | LDW instruction | SW instruction | SDW instruction .

- 2'b10 (SUB):

  Activated for opcode values 5'd3, 5'd10, 5'd11, and 5'd12.

  ALU_OP = SUB instruction | BZ instruction | BLZ instruction | BGZ instruction.

- 2'b11 (CMP):

  Activated exclusively for opcode value 5'd2.

  ALU_OP = CMP instruction.

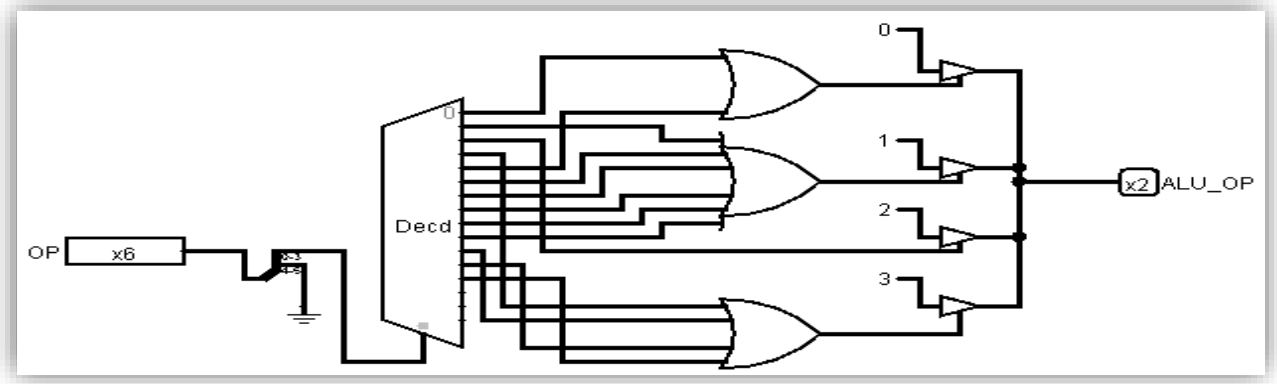

Figure 10: ALU control unit implementation.

Testing & Simulation results

Test case#1: LDW/SDW Hazard Detection with Loops

This test case will show how program deal with loops for a certain number using **BLZ,** Also how **LDW/SDW** will do two clocks and detect when odd Register is added.

Instruction Memory

Table 6: LDW/SDW Hazard Detection with Loops

Index	Assembly Instruction	HEX. Format
0X0	**ADDI R1, R1, 5**	**14440005**
0X1	**SW R1, [R0 + 0]**	**1C400000**
0X2	**ADDI R0, 1**	**14000001**
0X3	**SWD R0, [R0 + 1]**	**24000001**
0X4	**ADDI R2, R0, -3**	**14803FFD**
0X5	**BLZ R2, -6**	**30083FFB**
0X6	**LWD R0, [R0 + 0]**	**20000000**
0X7	**LWD R1, [R0 + 0]**	**20400000**
0X8	**SWD R0, [R0 + 4]**	**24000004**

Excepted Results:

The loop will continue iterating through R0, R1 and increasing them by 1, 5 three times, then do LDW twice which one of them should be rejected so the final results should be:

Table 7: Excepted Results LDW/SDW Hazard.

Index	Value
0X0	5
0X1	A
0X2	F
0X3	2
0X4	3
0X5	F
0X6	2
0X7	2

Waveforms:

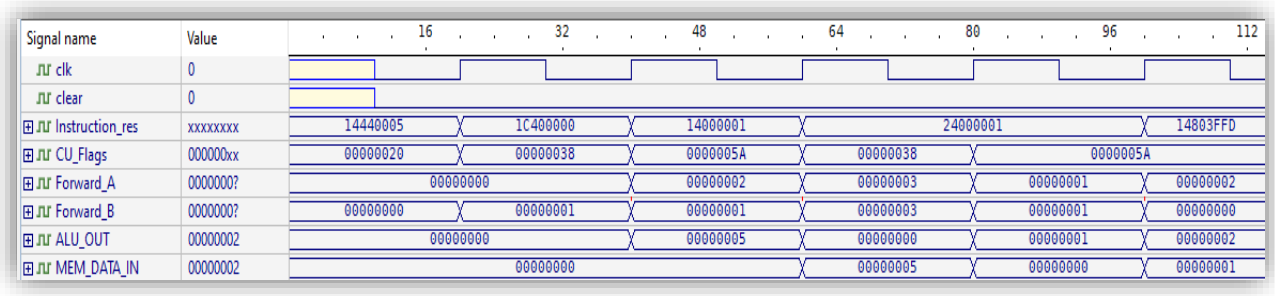

Figure 11: Case 1 - First Cycle.

Can notice how the **SDW** (2400001) got two cycles so store **R0** then **R(0 + 1) values**

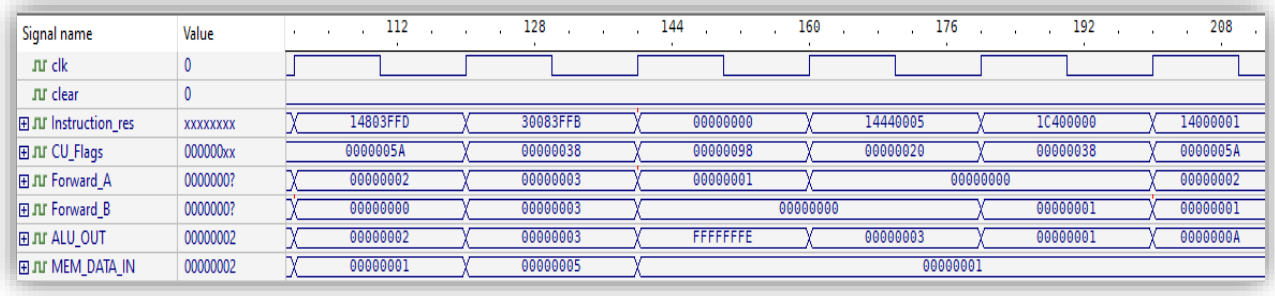

Figure 12: case 1 - Second iteration

The iteration by **BLZ** was successful.

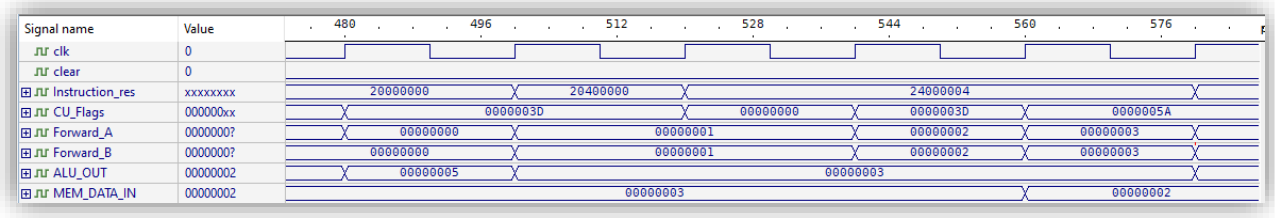

Figure 13: Case 1 - Last loop

In the last execution cycles, **SDW** got three cycles because it requires two originally and one added by stall causing by the **LDW** instruction before it.

Actual Results:

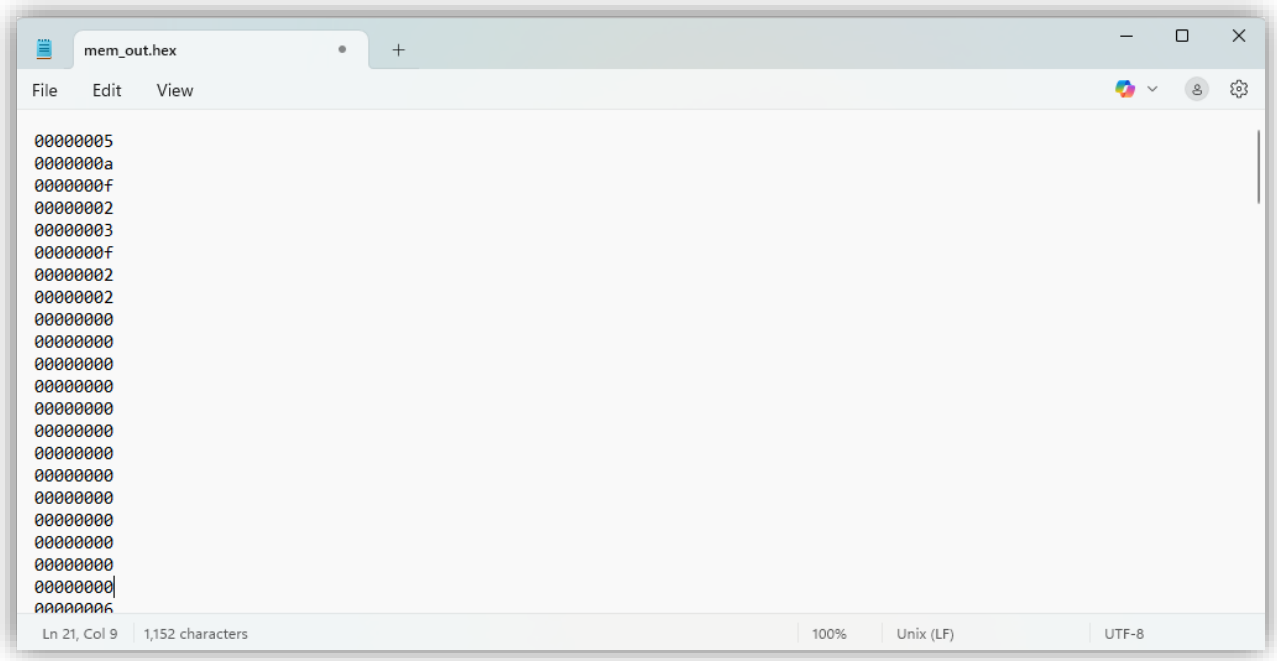

Figure 14: Case 1 - MEM. Out

Test case#2: Nested Loops

This test case will show how program can deal with loops efficiently even if nested calls done for a certain number for each loop using **BGZ** for two register **R5, R6** which will determine the iterations number**,** Showing the efficiency of forwarding logic.

Instruction Memory

Table 8: Nested Loops.

Index	Assembly Instruction	HEX. Format
0X0	ADDI R6, 3	15980003
0X1	ADDI R5, 3	15540003
0X2	ADDI R1, R1, 5	14440005
0X3	SW R1, [R0 + 0]	1C400000
0X4	ADDI R0, 1	14000001
0X5	SW R0, [R0 + 1]	1C000000
0X6	ADDI R5, -1	15543FFF

0X7	**BGZ R5, -3**	**2C143FFD**
0X8	**ADDI R0, 1**	**14000001**
0X9	**ADDI R6, -1**	**15983FFF**
0XA	**BGZ R6, -8**	**2C183FF7**

Excepted Results:

The two loops will continue iterating through R0, R1 and increasing them by 1, 5 three times in nested way, which will cause the output to be, (Where R0 is the value iterated through nested loops):

Table 9: Excepted Results of nested loops.

Index	Value
0X0	**5**
0X1	**1**
0X2	**2**
0X3	**3**
0X4	**A**
0X5	**5**
0X6	**6**
0X7	**7**
0X8	**F**
0X9	**9**
0XA	**A**
0XB	**B**

Waveforms:

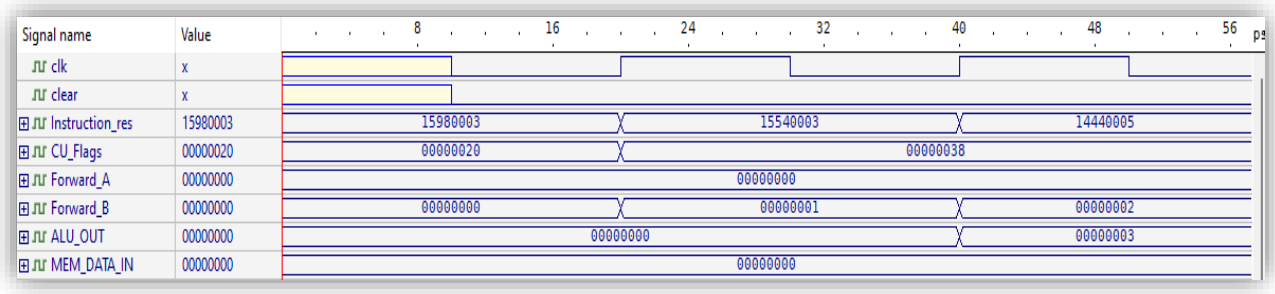

Figure 15: Case 2 - First Cycle

First Cycle and process initialization

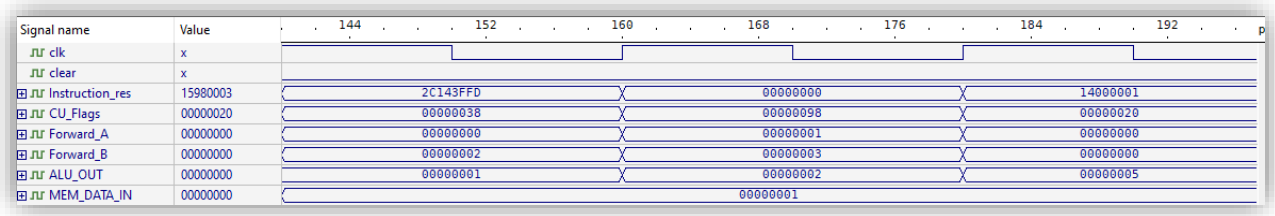

Figure 16: Case 2 - BGZ Success

The iteration by **BGZ** was successful, returned to **ADDI R0, R0, 1**.

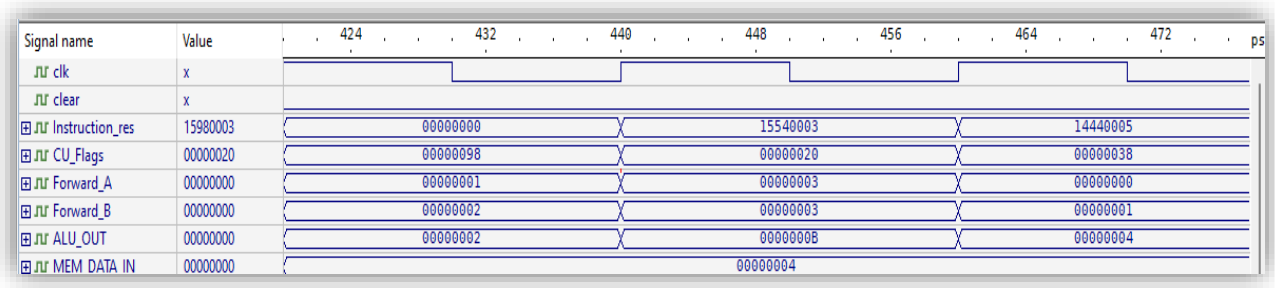

Figure 17: Case 2 - second BGZ success

The Second **BGZ** Successfully returned the program to **ADDI R1, R1, 5**.

Actual Results:

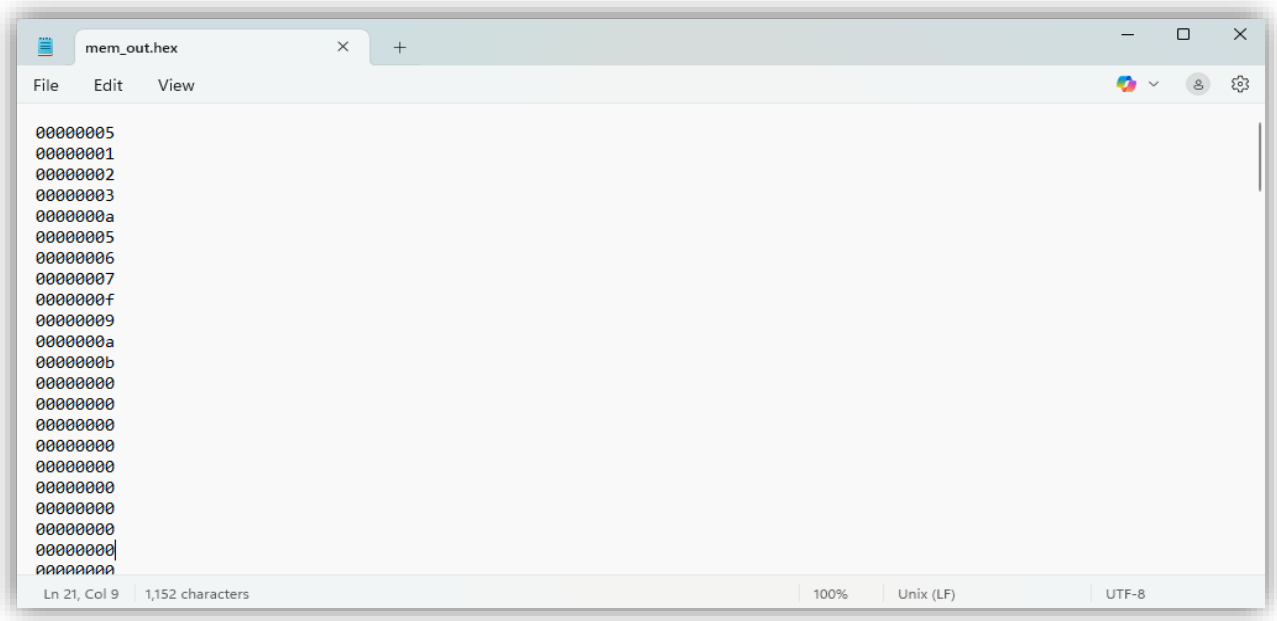

Figure 18: Case 2 – MEM. Out

Test case#3: Function Calls and Return

This test case will show how program can deal with function Calls using **CLL** and then return to a certain point using **JR R14** which saved earlier by **CLL** instruction, this process will call a function to loop through **R1** increasing it by 5 each time and saving it to the memory creating an Array, then loop through the created array which its address saved earlier by the first function, to do a summation through it and save its values in MEM out.

Instruction Memory

Index	Assembly Instruction	HEX. Format
0X0	CLL 10	3C000010
0X1	CLL 20	3C00001F
0X10	ADDI R1, R1, 5	14440005
0X11	SW R1, [R0 + 0]	1C400000
0X12	ADDI R0, R0, 1	14000001
0X13	ADD R2, R0, -3	14803FFD
0X14	BLZ R2, -4	30083FFC
0X15	JR R14	34380000

0X20	LW R3, [R0 + 0]	18C00000
0X21	ADD R4, R4, R3	0510C000
0X22	ADDI R0, -1	14003FFF
0X23	BGZ R0, -3	2C003FFD
0X24	SW R4, [R8 + 5]	1D200005
0X25	JR R14	34380000

Excepted Results:

The first loops called by the first function will create an array with 5, A, F then another function to sum it.

Index	Value
0X0	5
0X1	A
0X2	F
0X5	1E

Waveforms:

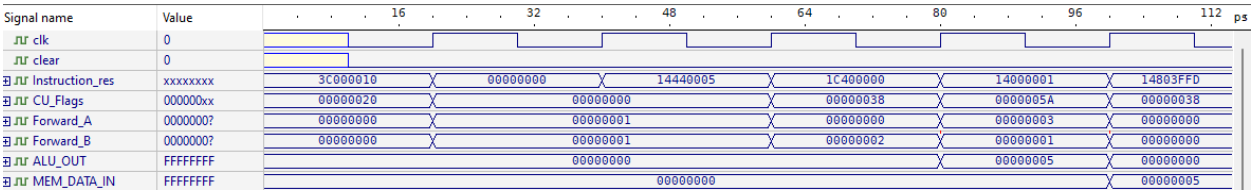

Figure 19: Case 3 - First clock

The instruction **CLL 10** Actually moved the PC to 0x10.

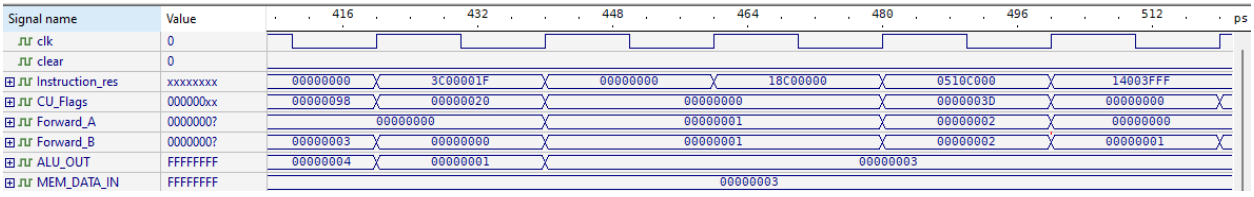

Figure 20: Case 3 – JR 14 Successes

The instruction **JR 14** returned the program to the function call which prove **CLL** and **JR** are executed successfully.

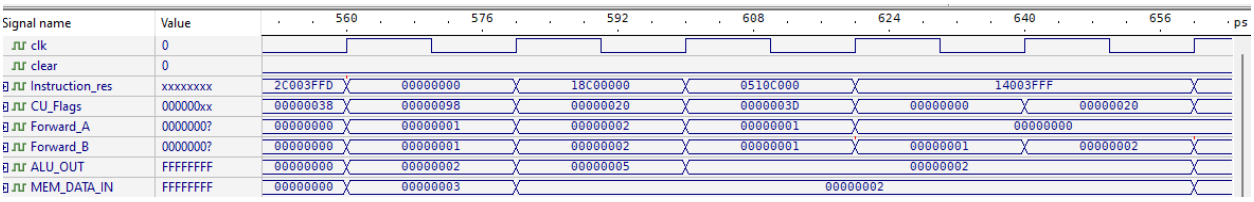

Figure 21: Case 3 – stalls

Stalls are happening Also.

Actual Results:

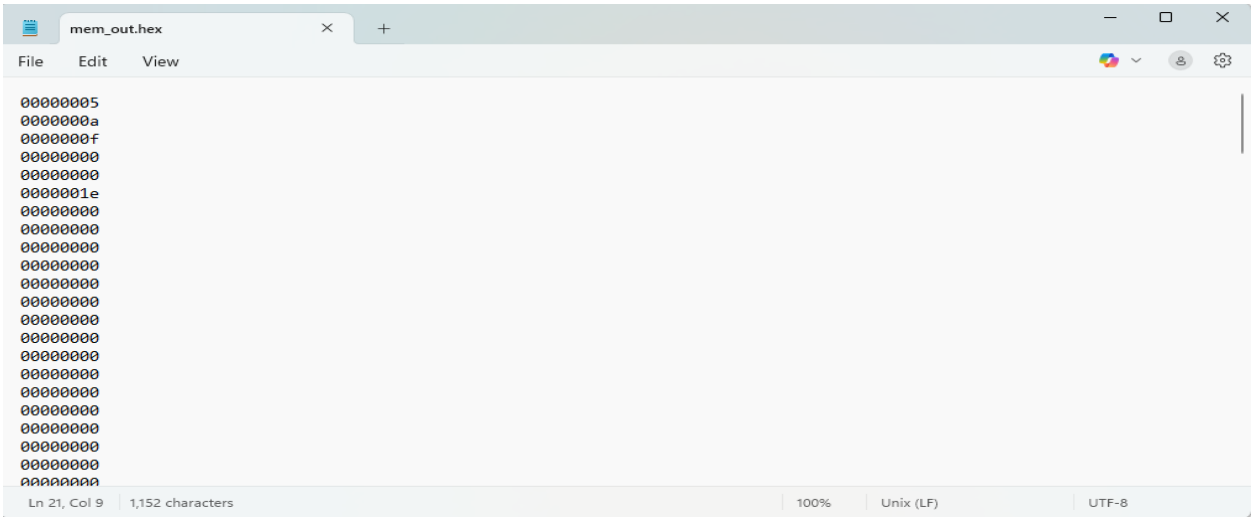

Figure 22: Case3 – MEM. Out.

Test case#4: Duplicated Arrays Checker

This test case is a large case which takes two arrays from the memory, iterate through each of them in a nested way, and store **1** in **0x5** if there's any duplicate number, The two memories addressees will be load from the memory using **LDW** to registers **R0, R1** and the length of them both also will be load in **R2**.

Initial Data Memory

```
66        mem[0] = 32'h0000000a; // Array1 Address
67        mem[1] = 32'h00000014; // Array2 Address
68        mem[2] = 32'h00000005; // Arrays Length
69
70        // Array#1
71        mem[10] = 32'h00000001;
72        mem[11] = 32'h00000002;
73        mem[12] = 32'h00000003;
74        mem[13] = 32'h00000004;
75        mem[14] = 32'h00000005;
76
77        // Array#2
78        mem[20] = 32'h0000000A;
79        mem[21] = 32'h0000000B;
80        mem[22] = 32'h0000000C;
81        mem[23] = 32'h0000000D;
82        mem[24] = 32'h0000000E;
```

Figure 23: Initial Data Memory.

Instruction Memory

Table 10: Duplicated Arrays Checker.

Index	Assembly Instruction	HEX. Format
0X0	LDW R0, [R0 + 0]	20000000
0X1	CLL 20	3C000020
0X21	LDW R2, [R13 + 0]	18B40002
0X22	ADDI R2, R2, 1	14880001
0X23	LW R6, [R0 + 0]	19800000
0X24	CLL 15	3C00000F
0X25	LW R6, [R0 + 0]	19800000
0X26	ADDI R0, R0, 1	14000001
0X27	ADDI R2, R2, -1	14883FFF
0X28	BGZ R2, -4	2C083FFC
0X29	J 71	38000021
0X33	LW R3, [R13 + 2]	18F40002
0X34	LW R1, [R13 + 1]	18740001
0X35	LW R7, [R1 + 0]	19C40000
0X36	CMP R8, R7, R6	0E1D8000
0X37	BZ R8, 0X11	28200011
0X38	ADDI R1, R1, 1	14440001
0X39	ADDI R3, R3, -1	14CC3FFF
0X3A	BGZ R3, -5	2C0C3FFB
0X2B	JR R14	34380000
0X4A	ADDI R10 , R10, 1	16A80001
0X4B	SW R10, [R13 + 0]	1EAC0005

Excepted Results:

The two loops will give on 0x5 1 if there's a problem, 0x5 will remain 0 if no problem, for the mem above will remain 0.

Table 11: Excepted Result of Duplicated Arrays Checker.

Index	Value

0X0	10
0X1	20
0X2	5
0X5	0

Waveforms:

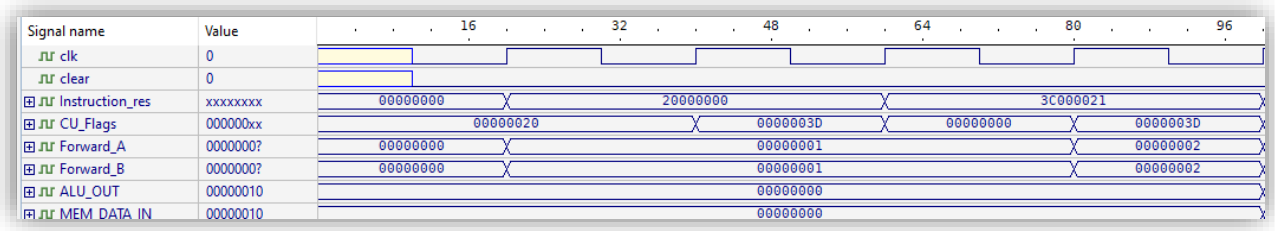

Figure 24: case 4 - LDW then Call

Call function take two cycles to insure no error

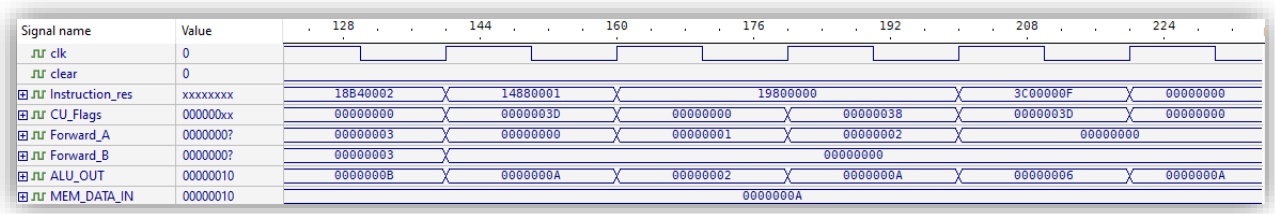

Figure 25: Case 4 - BGZ Success

Jump Successfully done

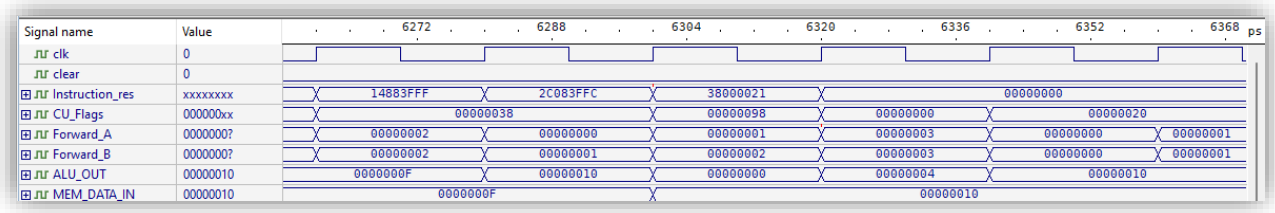

Figure 26: Case 4 – store r10

R10 value stored to determine if **zero** or **one** (if it was zero the function will skip **ADDI R10, R10, 1**)

Actual Results:

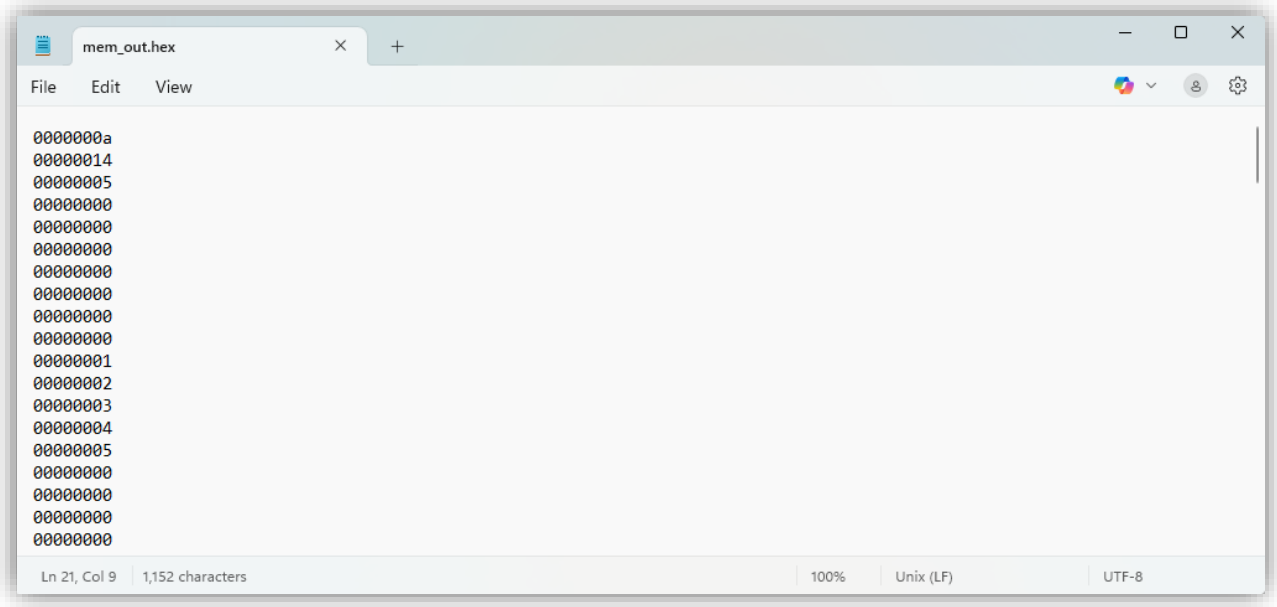

Figure 27: Case 4 – MEM. Out

Conclusion

In this project, we successfully designed and implemented a 32-bit pipelined RISC processor

using Verilog and Logisim. The processor is based on a five-stage pipeline architecture, which includes instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write-back (WB). This pipeline structure enhances performance by allowing multiple instructions to be processed simultaneously across different stages. During the implementation, we developed both the datapath and the control path, ensuring correct coordination between different units of the processor. To manage potential pipeline hazards such as structural hazards, data hazards, and control hazards we integrated stalling and forwarding mechanisms. These features were essential for maintaining the correct sequence of instruction execution and preventing errors due to data dependencies or control changes. The processor was tested with different instruction types including arithmetic operations like ADD and SUB, logic operations such as OR, memory-related instructions like load (LW) and store (SW), and various branching and jumping instructions. All tests were executed successfully, which confirms that our design and Verilog implementation were correct and reliable. This project provided valuable experience in understanding how pipelined processors function, how control signals must be synchronized, and how to effectively manage hazards to build an efficient and accurate processor architecture.