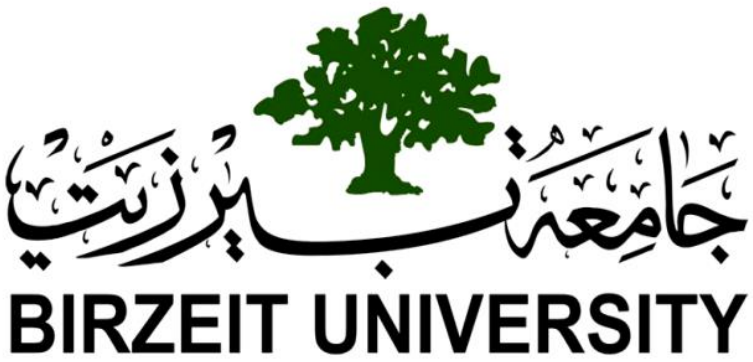


Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Computer Architecture - ENCS4370

Project Two Report

Design and Verification of a Simple Pipelined RISC Processor in Verilog

---

Prepared by:

Partner1_Name: Israa Turkman. **Partner1_Number:** 1221003.

Partner2_Name: Donia Said. **Partner1_Number:** 1222600.

Partner3_Name: Aseel Rabee. **Partner1_Number:** 1210627.

Instructor: Dr.Ayman Hroub Dr. Aziz Qaroush

Section: 1.

Date: 10/06/2025.

Abstract

This project presents the complete design and implementation of a 32-bit multi-cycle RISC processor using Verilog HDL. The processor was developed to execute a predefined instruction set that includes arithmetic, logical, memory, and control-flow operations. The architecture features a unified instruction format, a register file of 16 general-purpose 32-bit registers, and distinct instruction and data memories. The processor operates in a multi-cycle manner, where each instruction is broken down into sequential stages (fetch, decode, execute, memory access, and write-back), allowing efficient reuse of hardware units across different instructions. Special instructions like LDW and SDW are supported and handled across multiple cycles to manage double-word memory operations. The project also incorporates a centralized control unit based on a finite state machine, managing datapath control signals dynamically based on the current instruction. Simulation and waveform analysis were used to verify correctness across all instruction types.

Table of Contents

Table of figures

List of tables

1. Design and Implementation

In this project, the objective was to design and validate a 32-bit multi-cycle RISC processor using Verilog HDL. The development process began with defining a custom instruction set architecture (ISA), which included specifying a unified instruction format and the required operation types. Based on the ISA, key datapath components and control logic were carefully selected to ensure correct functionality and minimal hardware redundancy. The processor was implemented as a multi-cycle design, with each instruction broken down into several sequential execution stages. This report documents the complete design methodology, presents a detailed breakdown of all core components, and explains how each instruction is processed through the datapath cycle-by-cycle.

1.1. Processor Specifications and Overview

The designed processor is based on the following specifications:

1. **Instruction and Word Size:** Both instruction size and data word size are fixed at 32 bits.
2. **Register File:** The processor includes sixteen general-purpose registers (R0 to R15), each 32 bits wide.
3. **Program Counter (PC):** The register R15 is reserved and hardwired to function as the Program Counter, which points to the address of the next instruction to execute.
4. **Return Address Register:** Register R14 is used to store the return address during subroutine calls (CALL/RET instructions).
5. **Memory Architecture:** The design features two physically separated memory modules:
    - Instruction Memory (read-only, word-addressable)
    - Data Memory (read/write, word-addressable)
6. **Instruction Format:** The ISA uses a single fixed-format for all instructions:

$Opcode^6$	Rd^4	Rs^4	Rt^4	$Immediate^{14}$

Table 1: Instructions Format

7. The immediate field is sign-extended for most instructions and zero-extended for specific logical operations.
8. **Instruction Types:** Although the ISA uses one instruction format, it supports multiple categories of operations including:
    - Arithmetic and Logical
    - Memory Access (LW, SW, LDW, SDW)
    - Branching and Control (BZ, BGZ, BLZ, J, JR, CALL)
9. **Memory Addressing:** All memory is word-addressable, and memory accesses are aligned accordingly.
10. **Execution Model:** A multi-cycle execution model was adopted, where each instruction is processed over multiple clock cycles across five distinct stages: Fetch, Decode, Execute, Memory, and Write Back.

These specifications directly influenced the datapath structure and control signal organization, allowing for efficient instruction handling and hardware reuse across execution stages.

In a multi-cycle processor, instruction execution is broken down into five well-defined stages:

1. **Instruction Fetch (IF)**
2. **Instruction Decode (ID)**
3. **Execution (EX)**
4. **Memory Access (MEM)**
5. **Write Back (WB)**

Each of these stages is executed in a separate clock cycle, enabling the processor to reuse internal hardware components such as the ALU and memory units across different stages. Unlike single-cycle designs, which require all operations to complete in one clock cycle, the multi-cycle approach assigns one cycle per stage and allows instructions to occupy the datapath for only as long as needed. Depending on the instruction type, total execution time typically ranges from 2 to 5 cycles.

All instructions begin with the fetch and decode stages. Subsequent stages execution, memory access, and write-back are included as needed based on the instruction's behavior. For example, memory access is only triggered for load and store instructions, while write-back is skipped for

unconditional jumps. These variations are handled through a centralized control unit that manages the datapath according to the current instruction and state. Details of each stage and how they are applied per instruction type are discussed in the following sections.

1.2. Instruction Format & RTL Operations

1.2.1. Instruction Formats and Interpretations

The Instruction Set Architecture (ISA) implemented in this processor adopts a **single unified 32-bit instruction format** that supports all operation types arithmetic, logic, memory access, and control flow. This design choice simplifies the decoding process and minimizes datapath complexity, as all instructions follow the same structural layout.

The instruction format is defined as follows:

$Opcode^{6}$	Rd^{4}	Rs^{4}	Rt^{4}	$Immediate^{14}$

Table 2: Instructions Format

Explanation of each field:

Opcode (6 bits): Specifies the operation to be executed.

Rd (4 bits): Destination register.

Rs (4 bits): First source register.

Rt (4 bits): Second source register, or used as an auxiliary operand depending on the instruction.

Immediate (14 bits): A constant value used in immediate arithmetic, memory addressing, or control instructions.

1.2.2. Instructions' Encoding

The processor implements a carefully selected subset of a custom Instruction Set Architecture (ISA) designed to support fundamental operations across three major instruction categories: **R-Type**, **I-Type**, and **J-Type**. Each instruction is uniquely encoded using a combination of a 6-bit opcode and, for R-Type instructions, an additional 3-bit function code. This encoding scheme allows the control unit to accurately decode and execute operations according to their intended behavior.

Below is a categorized breakdown of the instructions supported by this processor:

Instruction	Operands	Meaning	Opcode Value
OR	Rd, Rs, Rt	Reg(Rd) = Reg(Rs) | Reg(Rt)	000000
ADD	Rd, Rs, Rt	Reg(Rd) = Reg(Rs) + Reg(Rt)	000001
SUB	Rd, Rs, Rt	Reg(Rd) = Reg(Rs) - Reg(Rt)	000010
CMP	Rd, Rs, Rt	Reg(Rd) = 0 if Reg(Rs) == Reg(Rt), -1 if Reg(Rs) < Reg(Rt), 1 if Reg(Rs) > Reg(Rt)	000011
ORI	Rd, Rs, Imm	Reg(Rd) = Reg(Rs) | Imm	000100
ADDI	Rd, Rs, Imm	Reg(Rd) = Reg(Rs) + Imm	000101
LW	Rd, Imm(Rs)	Reg(Rd) = Mem[Reg(Rs) + Imm]	000110
SW	Rd, Imm(Rs)	Mem[Reg(Rs) + Imm] = Reg(Rd)	000111
LDW	Rd, Imm(Rs)	Reg(Rd) = Mem[Reg(Rs) + Imm]; Reg(Rd+1) = Mem[Reg(Rs) + Imm + 1]	001000
SDW	Rd, Imm(Rs)	Mem[Reg(Rs) + Imm] = Reg(Rd); Mem[Reg(Rs) + Imm + 1] = Reg(Rd+1)	001001

BZ	Rs, Label	if Reg(Rs) == 0, PC = PC + Imm	001010
BGZ	Rs, Label	if Reg(Rs) > 0, PC = PC + Imm	001011
BLZ	Rs, Label	if Reg(Rs) < 0, PC = PC + Imm	001100
JR	Rs	PC = Reg[Rs]	001101
J	Label	PC = PC + Imm	001110
CLL	Label	R14 = PC + 1; PC = PC + Imm	001111

Table 2: Instruction's Encoding Table

1.2.3. Instruction RTL Micro Operations

This section defines the Register Transfer Language (RTL) micro-operations for the implemented instructions. RTL captures the internal behavior of the processor by describing how data is transferred between registers, memory, and functional units like the ALU. Each instruction is executed as a sequence of such micro-operations distributed across multiple clock cycles.

In a multi-cycle implementation, each instruction proceeds through five main stages:

IF → ID → EX → MEM → WB

Each stage may include one or more micro-operations, depending on the instruction type and functionality. These RTL steps are essential for controlling the datapath and ensuring correct execution.

The processor uses:

- A **32-bit word size**
- A **word-addressable memory**
- A **32-bit Program Counter**, which is hardwired to register **R15**
- A dedicated return address register (**R14**) for function calls and returns

Instructions are grouped according to their operational behavior. Each group shares similar control and data movement patterns. The RTL operations below describe the sequence of actions per group, step-by-step, across the processor's datapath. Unless otherwise noted, multiple micro-operations listed on the same step are assumed to occur in the same clock cycle.

Group 1: R-Type

1. IR ← Mem[PC]
2. PC ← PC + 1
3. A ← Reg[Rs], B ← Reg[Rt]
4. ALUOut ← ALU(A, B)
5. Reg[Rd] ← ALUOut

Group 2: I-Type (ADDI, ORI)

1. IR ← Mem[PC]

2. PC ← PC + 1

3. A ← Reg[Rs], Imm ← (Sign or Zero)-Extended Immediate

4. ALUOut ← ALU(A, Imm)

5. Reg[Rd] ← ALUOut

Group 3: Load Word (LW)

1. IR ← Mem[PC]

2. PC ← PC + 1

3. A ← Reg[Rs], Imm ← Sign-Extended Immediate

4. ALUOut ← A + Imm

5. MDR ← Mem[ALUOut]

6. Reg[Rd] ← MDR

Group 4: Store Word (SW)

1. IR ← Mem[PC]

2. PC ← PC + 1

3. A ← Reg[Rs], B ← Reg[Rd], Imm ← Sign-Extended Immediate

4. ALUOut ← A + Imm

5. Mem[ALUOut] ← B

Group 5: Load Double Word (LDW)

1. IR ← Mem[PC]

2. PC ← PC + 1

3. A ← Reg[Rs], Imm ← Sign-Extended Immediate

4. ALUOut ← A + Imm

5. MDR ← Mem[ALUOut]

6. Reg[Rd] ← MDR

7. MDR ← Mem[ALUOut + 1]

8. Reg[Rd + 1] ← MDR

Group 6: Store Double Word (SDW)

1. IR ← Mem[PC]

2. PC ← PC + 1

3. A ← Reg[Rs], B1 ← Reg[Rd], B2 ← Reg[Rd + 1], Imm ← Sign-Extended Immediate

4. ALUOut ← A + Imm

5. Mem[ALUOut] ← B1

6. Mem[ALUOut + 1] ← B2

Group 7: Branch (BZ, BGZ, BLZ)

1. IR ← Mem[PC]

2. A ← Reg[Rs], Imm ← Sign-Extended Immediate

3. if condition met → PC ← PC + Imm

4. else → PC ← PC + 1

Group 8: JR

1. IR ← Mem[PC]

2. PC ← Reg[Rs]

Group 9: JUMP

1. IR ← Mem[PC]

2. PC ← PC + Sign-Extended Immediate

Group 10: CALL

1. IR ← Mem[PC]

2. Reg[14] ← PC + 1

3. PC ← PC + Sign-Extended Immediate

Group 11: RET

1. IR ← Mem[PC]

2. PC ← Reg[14]

1.3. Functional Units and Components

In the design of our processor, several functional units and components are essential for ensuring the correct execution of instructions. These units work together to process data, perform arithmetic and logical operations, handle memory access, and manage control flow.

1. Program Counter (PC)

In this processor, the **Program Counter (PC)** is implemented as a dedicated 32-bit register responsible for holding the address of the next instruction to fetch. It is connected directly to the **Instruction Memory** as its address input. The PC register supports multiple input sources via a multiplexer (8-to-1 MUX), which allows updating its value from various paths:

- PC + 1 (for sequential execution)
- Reg[Rs] + Imm (e.g., for branching)
- Jump/Call target (constructed using PC[15:9] concatenated with a 9-bit offset)
- RR (for RET)
- Reg[Rs] (for JR)

The PC is clocked and includes an **enable signal (PCwrite)**, controlled by the PC Control unit. The correct source is selected through the **PCsrc** signal, which also comes from the control unit based on instruction type and ALU/branch results.

2. Instruction Register (IR)

The **Instruction Register (IR)** is a 32-bit register that stores the instruction fetched from the **Instruction Memory**. It connects directly to the output of instruction memory and is written during the **IF stage** when the signal IRwrite is active. From the IR, all instruction fields are extracted (opcode, func, Rd, Rs, Rt, Imm), and sent to various components, especially the **Decoding and Multiplexing Unit**.

Its output drives:

- Opcode and Func inputs to the **Main Control Unit** and **ALU Control**
- Register addresses and immediate fields
- Multiplexer selectors and Extender inputs

3. Return Register (RR)

The **Return Register (RR)** is implemented as a standalone 32-bit register, labeled clearly in the datapath. It is used to store the return address during a CLL (call) instruction and is read during RET. The value written into RR is typically PC + 1, computed by a dedicated **adder** connected to the PC.

The RR includes:

- A write enable signal (RRwrite)
- A direct path back to the PC via the PC MUX (for RET execution)

4. Instruction Memory

The **Instruction Memory** is a dedicated, read-only memory block used solely for storing the processor's program instructions. It is physically separated from the data memory, as shown in the datapath, which aligns with the Harvard architecture principle.

The memory:

- Is **word-addressable**, where each address points to a full **32-bit instruction**.
- Receives its address input directly from the **Program Counter (PC)**.
- Outputs a 32-bit instruction that is immediately loaded into the **Instruction Register (IR)** during the **Fetch (IF)** stage.

No write operations are performed on the instruction memory during execution — it acts strictly as a **read-only** component. This organization allows for efficient and reliable instruction retrieval, with each instruction occupying exactly one memory location (one word), simplifying the fetch logic and reducing control complexity.

The interface between Instruction Memory and other components is tightly synchronized via the control signal PCwrite, ensuring instructions are fetched only when needed. The output of the memory is registered into the IR when the signal IRwrite is asserted by the control unit.

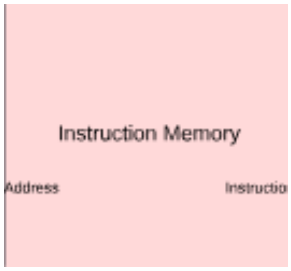


Figure 1: Instruction Memory

5. Data Memory

The **Data Memory** is a critical component used during the **Memory Access (MEM)** stage of instruction execution. It provides read and write capabilities for load and store instructions, such as LW, SW, LDW, and SDW. This memory block is physically separate from the instruction memory and is explicitly designed to handle 32-bit **word-addressable** data operations.

Functional Characteristics:

- **Address Input:**
  A 32-bit address generated by the ALU (from Reg[Rs] + Imm) is provided via the signal address.
- **Data Input:**
  For store operations (SW, SDW), the value from the register file (bus_write) is routed to the memory through the dataIn line.
- **Control Signals:**
  - MemRd: Activates memory read operations.
  - MemWr: Enables write operations to memory.
    These signals are controlled by the **Main Control Unit** depending on the instruction type and current FSM state.
- **Data Output:**
  During load operations, the retrieved 32-bit word from memory is passed to the **Memory Data Register (MDR)**, then written back to the register file.
- **Clocked Writes:**
  Write operations (MemWr = 1) are synchronized with the system clock to ensure stable and accurate memory updates.

This design allows smooth interaction with extended instructions like LDW and SDW, which access **two consecutive memory words** by issuing two memory operations in successive cycles. The Address MUX and dedicated increment logic in the datapath ensure that the second word access occurs at Address + 1.

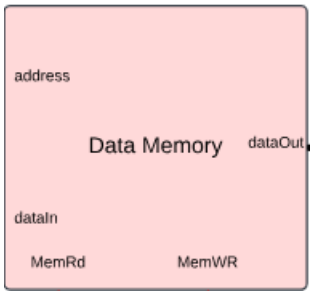


Figure 2: Data Memory

6. Register File

The **Register File** is a core component of the processor, providing fast and parallel access to general-purpose registers required during instruction execution. As specified in the processor architecture, the design includes **16 registers (R0–R15)**, each **32 bits wide**.

- **R15** is hardwired to serve as the **Program Counter (PC)**.
- **R14** is used to store the **return address** during function calls (CLL) and is read during returns (RET).
- **R0** may be hardwired to zero depending on implementation choice (optional), but no write protection logic is enforced in the datapath unless explicitly added.

Structural Details:

- **Read Ports:**
  The register file provides two simultaneous **read ports**, driven by 4-bit addresses addr_read1 and addr_read2. The outputs appear on bus_read1 and bus_read2 respectively.
  - **Reading is purely combinational**, requiring no clock signal.
- **Write Port:**
  A single **write port** exists, controlled via:
  - addr_write (4-bit write address)
  - bus_write (32-bit data input)
  - RegWr (write enable signal)

  Writing is **clocked**, and data is only written on the active edge of the clock **when RegWr is asserted**.

This design allows flexible operand selection and result storage, supporting a wide variety of instruction types including arithmetic, memory, and branching operations. The register file is accessed during the **Decode (ID)** and **Write-Back (WB)** stages of the multi-cycle execution flow.

Figure 3: Register File

7. ALU

The Arithmetic Logic Unit (ALU) is a combinational component responsible for executing arithmetic and logical operations during the execution stage of each instruction. In this processor, the ALU supports five primary operations: addition, subtraction, bitwise AND, bitwise OR, and comparison, based on the instruction set. It receives two 32-bit operands (A and B) and outputs a 32-bit result. The operation performed is determined by the ALUOp control signal, which is derived from the instruction's opcode. For the CMP instruction, the ALU compares the two operands and outputs one of three values: 0 (equal), -1 (less than), or 1 (greater than). The ALU also generates a Zero flag, which is set when the result is zero and is used by conditional instructions such as BZ, BGZ, and BLZ. The final output of the ALU is stored in the ALUOut register for use in subsequent stages.

Figure 4: ALU

8. Decode and Multiplexing Unit

In this processor, the format and location of instruction fields vary depending on the instruction type. For example, the source register Rs is located at bits [21:18] in I-type instructions and at [21:18] or [17:14] in R-type instructions depending on operand position. To simplify decoding and avoid using multiple multiplexers in the datapath, a dedicated Decode and Multiplexing Unit is implemented to interpret the instruction directly and extract the correct register addresses and immediate values.

This unit is active during the Instruction Decode (ID) stage. It receives the 32-bit instruction from the Instruction Register (IR) and extracts the following fields:

- addr_read1 – address of the first source register

- addr_read2 – address of the second source register or immediate value

- addr_write – address of the destination register

- I_type_imm – 14-bit immediate for I-type instructions (sign-extended)

- J_type_imm – 14-bit offset for J-type instructions (sign-extended)

- opcode field for ALU control and FSM state

This approach reduces datapath complexity and ensures decoding is handled uniformly for all instruction types.

Instruction	Dest. Reg (addr_write)	Op.1 (addr_read1)	Op.2 (addr_read2)	Comments
R-type	Rd = IR[25:22]	Rs = IR[21:18]	Rt = IR[17:14]	
ALU_I (ADDI, ORI, ANDI)	Rt = IR[25:22]	Rs = IR[21:18]	Imm = IR[13:0]	
LW	Rt = IR[25:22]	Rs = IR[21:18]	Imm = IR[13:0]	
SW	-	Rs = IR[21:18]	Imm = IR[13:0]	Data in: Rt = IR[25:22]
LDW	Rd = IR[25:22]	Rs = IR[21:18]	Imm = IR[13:0]	Second word → Reg[Rd+1]
SDW	-	Rs = IR[21:18]	Imm = IR[13:0]	Rt = IR[25:22], second word Rt+1
BZ, BGZ, BLZ	-	Rs = IR[21:18]	Imm = IR[13:0]	Used for conditional branching
JR	-	Rs = IR[21:18]	-	Jump to register value

Table 3: Decode and Multiplixing Table for R-Type and I-type

For instructions that do not involve the ALU, the data is retrieved during the decode stage, as illustrated in the table below:

Instruction	Data
J	Offset = IR[13:0] → used to calculate the jump target
CLL	RR = PC + 1, target address = PC + Sign-Extended(IR[13:0])

Table 4: Decode and Multiplexing Table for J-Type

As demonstrated in the previous two tables, the positions of addr_read1, addr_read2, and addr_write within the instruction vary depending on the instruction type. Instead of using multiple multiplexers to extract these fields during decoding, we designed a dedicated component called the **Decode and Multiplexing Unit**, which directly interprets the instruction and generates the required control signals and register addresses.

This unit receives the 32-bit instruction as input and extracts the following fields:

- opcode: The 6-bit operation code located in bits [31:26]
- addr_write: The destination register address (Rd or Rt, depending on the instruction)
- addr_read1: The address of the first source register (Rs)
- addr_read2: The second source register address (Rt) or used as a placeholder for immediate instructions

It also extracts the immediate field as needed:

- I_type_imm: A **14-bit immediate** value used in I-type instructions, which is **sign-extended** for arithmetic and memory instructions, and **zero-extended** for logical ones (e.g., ORI, ANDI)
- J_type_imm: A **14-bit offset** used in jump and call instructions, also sign-extended before being added to the PC to calculate the target address

This centralized decoding approach eliminates the need for separate control logic or multiplexers inside the register file, streamlining the datapath and improving decode efficiency.

Figure 5: Decoding and Multiplexing Unit

9. Extender Unit

In this processor, immediate values are encoded using a fixed **14-bit field** in the instruction format. However, since the processor operates on **32-bit words**, the immediate must be expanded to 32 bits before it can be used in arithmetic operations, address calculations, or branching.

To handle this, an **Extender Unit** is implemented. It receives the 14-bit immediate value and outputs a 32-bit version according to the required extension mode. The type of extension is determined by a control signal (ExtSel), which is generated by the control unit based on the instruction's opcode.

- **Sign Extension**:
  Used in instructions such as ADDI, LW, SW, CMP, BZ, BGZ, BLZ, J, and CLL, where the immediate represents either an offset or signed value.
- **Zero Extension**:
  Used in logical operations like ORI and ANDI, where the immediate is treated as an unsigned value.

This separation allows for proper interpretation of the immediate operand while keeping the instruction format consistent and minimizing hardware complexity.

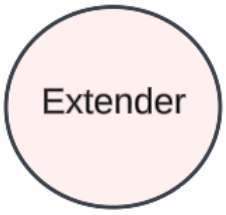


Figure 6: Extender Unit

10. Mux Units

We can observe from the datapath that several components support multiple possible input sources depending on the instruction type. These inputs are selected through **multiplexers (MUXes)**, and the appropriate input is chosen based on the **control signals** generated by the control unit. While some routing is handled internally by the Decode and Multiplexing Unit, there are still several external multiplexers required across the datapath to manage instruction behavior dynamically.

The most prominent example is the **8-to-1 multiplexer** connected to the Program Counter (PC). This MUX determines the source of the next PC value depending on the instruction being executed. The common sources include the default PC+1 (used by most instructions), the result of a branch condition (PC+1+Imm), register-based jumps (e.g., JR), and special return addresses (RR), among others. Each input to this MUX corresponds to a specific instruction behavior, and the control signal PCSrc is used to select the appropriate path.

In addition to the PC MUX, the datapath includes **three 2-to-1 MUXes**. The first one selects the source for writing back to the register file: either the ALU result or the memory output. The second MUX selects the **first ALU operand**, which is either the value from the register file (bus_B) or an immediate value (from the Extender). The third MUX controls the **second ALU operand**, allowing selection between a register value (bus_A) and a constant 1 — the latter is used specifically in the CMP instruction for subtraction and in certain conditional logic.

Moreover, the datapath incorporates a **3-to-1 multiplexer** at the write-back stage. This MUX selects between three possible sources for the value to be written back to the register file: the ALU result, the memory output, or the incremented PC value (PC+1). This is particularly useful for instructions like CLL, where the return address must be stored in register R14, and for load operations (LDW) where different sources may be involved across clock cycles. The control signal WBSel manages this selection.

These MUXes are critical to ensuring correct instruction flow, and they simplify datapath wiring by avoiding direct hard-coded connections. Each MUX is controlled by a dedicated signal from the control unit, allowing for flexible execution of different instruction types while maintaining a clean, modular architecture.

11. Buffers Between Stages

In our multi-cycle processor, each stage is isolated and operates independently across multiple clock cycles. To guarantee that intermediate results remain available when transitioning between stages, we implement a set of **dedicated registers (buffers)** to hold temporary values. These elements are clocked and synchronized with control signals to ensure proper sequencing and data integrity.

In addition to the **Program Counter (PC)**, **Instruction Register (IR)**, and **Return Register (RR)**, the design includes the following key intermediate buffers:

- **Register A**: Holds the value of bus_read1, extracted from the register file during the decode stage. This register serves as the **first operand** for the ALU during the execute stage.
- **Register B**: Holds the value of bus_read2, also extracted during decode. It provides the **second operand** for the ALU.
- **Register Imm**: Stores the **extended immediate value** produced by the Extender during the decode stage. It is used by the ALU when the instruction requires an immediate operand (e.g., ADDI, LW, SW).
- **Register ALURes**: Captures the **output of the ALU** after execution. This value is forwarded to either the memory stage (in load/store operations) or used during write-back to the register file.
- **Register Inst. Address**: Used primarily in double-word memory operations like LDW and SDW. It stores the **base instruction address**, supporting calculation of consecutive memory accesses during multi-cycle execution.
- **Memory Data Register (MDR)**: Temporarily holds **data read from memory** before it is written back to the register file. It ensures that memory output is available during the write-back stage without stalling the memory unit.

These buffers are essential to the multi-cycle architecture, allowing values to persist between cycles and enabling the sequential execution of each instruction stage without conflict or data loss.

a. Constructing the Datapath

Based on our comprehension of the individual components and their functionalities, we designed the data path as illustrated in the below figure.

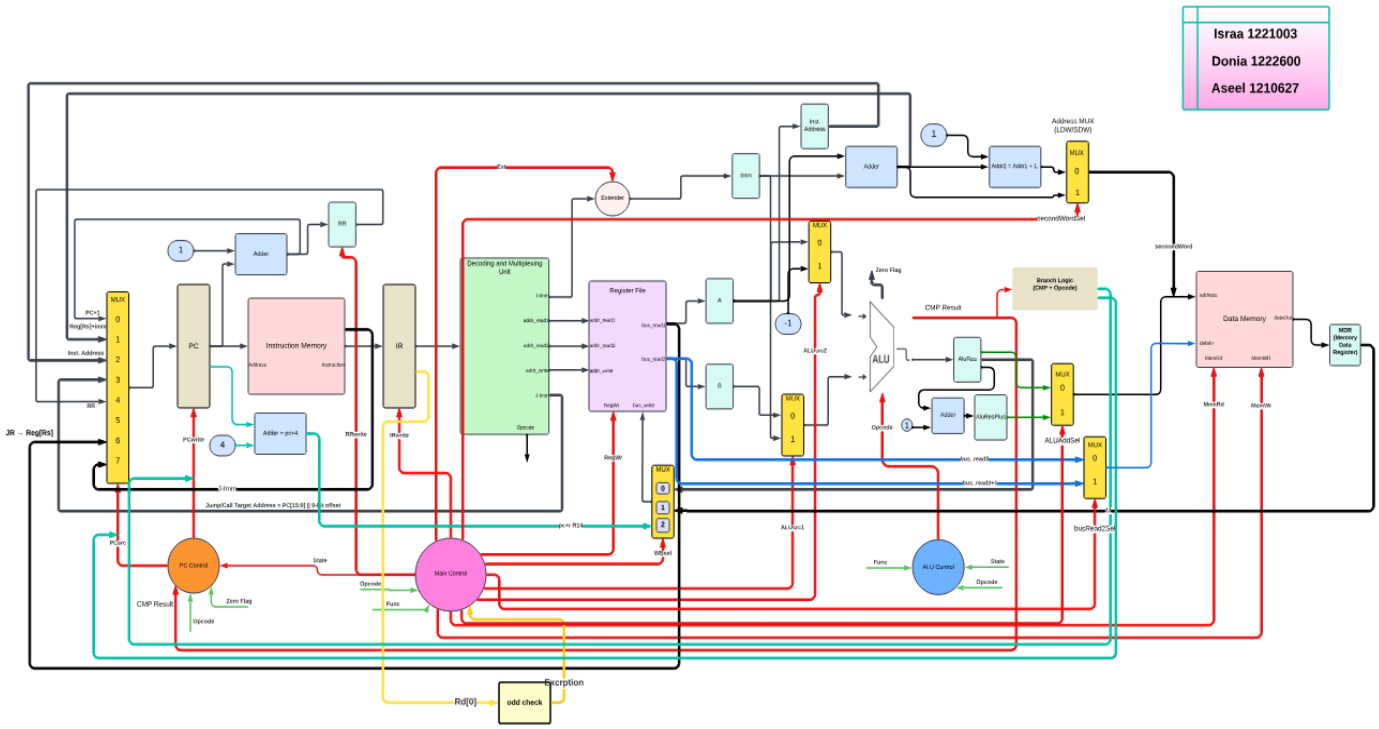


Figure 7: Datapath

Execution begins at the **Program Counter (PC)** during the instruction fetch stage. The PC holds the address of the current instruction. A +1 adder increments the PC to prepare for the next instruction in sequential order. When executing jump or branch instructions, a separate adder computes the **target address** by adding the sign-extended immediate to the current PC value. The selected address is routed via an 8-to-1 multiplexer connected to the PC. The instruction at the computed address is then fetched from **Instruction Memory** and stored in the **Instruction Register (IR)**.

In the decode stage, the IR contents are passed to the **Decode and Multiplexing Unit**, which extracts the opcode, register addresses, and immediate values. This stage also produces addr_read1, addr_read2, and addr_write, in addition to the I-type and J-type immediate values. The outputs of the register file are stored in **Register A** and **Register B**, and the extended immediate is stored in **Register Imm**. The decoded instruction is then routed to the appropriate next stage (Execute, Memory, or Write Back) based on its type and control signals.

In the **Execute stage**, the **Arithmetic Logic Unit (ALU)** performs the required operation. R-type instructions use Register A and Register B as operands. I-type arithmetic instructions use Register A and the extended immediate. For conditional branches (BZ, BGZ, BLZ), the ALU compares values from the register file, generating the Zero Flag signal. The ALU result is stored in the **ALURes** register. For LDW/SDW instructions, an additional adder calculates the second word address (Addr2 = Addr1 + 1).

In the **Memory stage**, only load and store instructions are active. Store instructions use the address from the ALURes register and write data from the Rt register to memory. Load instructions fetch data from memory and temporarily store it in the **Memory Data Register (MDR)**. In LDW and SDW operations, the Address MUX determines if a second consecutive memory access is needed.

During the **Write Back stage**, a 3-to-1 multiplexer selects the final result to be written back to the register file. The value could come from the ALU, from the MDR, or from PC+1 in the case of a CLL instruction. The destination register is determined by the instruction format (Rd for R-type, Rt for I-type, or R14 for CLL).

To implement the multi-cycle architecture, a set of control units orchestrates the operation of the datapath:

1. PC Control Unit

- Manages the **8-to-1 PC multiplexer**, selecting the next PC source depending on instruction type.
- Handles jump, call, branch, and return behaviors by routing the correct address to the PC.
- Enables PC writes using the PCwrite control signal.
- Inputs: Opcode, CMP Result, and Zero Flag.
-

2. ALU Control Unit

- Decides the operation the **ALU** performs at each stage (e.g., ADD, SUB, OR, CMP).
- Takes Opcode and State as inputs.

- Generates function select lines for the ALU.
- Ensures alignment with instruction type and stage.

3. Main Control Unit

- The central controller responsible for coordinating all stages.
- Generates:
    - IRwrite: Enables writing into the Instruction Register.
    - RRwrite: Enables writing into the Return Register.
    - ExtSel: Chooses between sign or zero extension.
    - WBSel: Controls the 3-to-1 MUX for register write-back.
    - ALUsrc1, ALUsrc2: Control the source of each ALU operand.
    - MemRd, MemWr: Select the read/write mode of the Data Memory.
    - RegWr: Enables writing back to the register file.
    - Odd Check line: Triggers the exception path for invalid LDW/SDW destination register numbers.

b. Designing the Control Unit

In our multi-cycle processor, instructions require a variable number of clock cycles based on their operational behavior. This variation depends solely on the opcode field, as our instruction format does not use separate function codes or instruction types. To coordinate this, the control unit is implemented as a Mealy finite state machine (FSM), where the transition between states is driven by the current state, the opcode, and internal condition flags such as the zero flag or CMP result.

Each state in the FSM corresponds to a specific phase in the execution of a certain instruction or instruction group (e.g., OR/ADD, LW, LDW Word2, etc.). For every state, the control unit activates a precise combination of control signals to steer data correctly through the datapath. These signals include register file write enables, memory read/write controls, multiplexer selections, ALU operations, and PC control.The diagram below shows the final FSM used in our project, where each labeled block represents a state with its associated control outputs. Control signals not shown are implicitly set to 0, and ALU_Opcode defaults to a no-op unless overridden. Conditional transitions (e.g., for BZ, BGZ, BLZ) rely on flags computed during the execution stage, ensuring correct branching or sequential flow.

This FSM structure allows for efficient resource sharing across different instructions and guarantees correct sequencing of actions in each stage, reducing hardware complexity while preserving full ISA functionality.

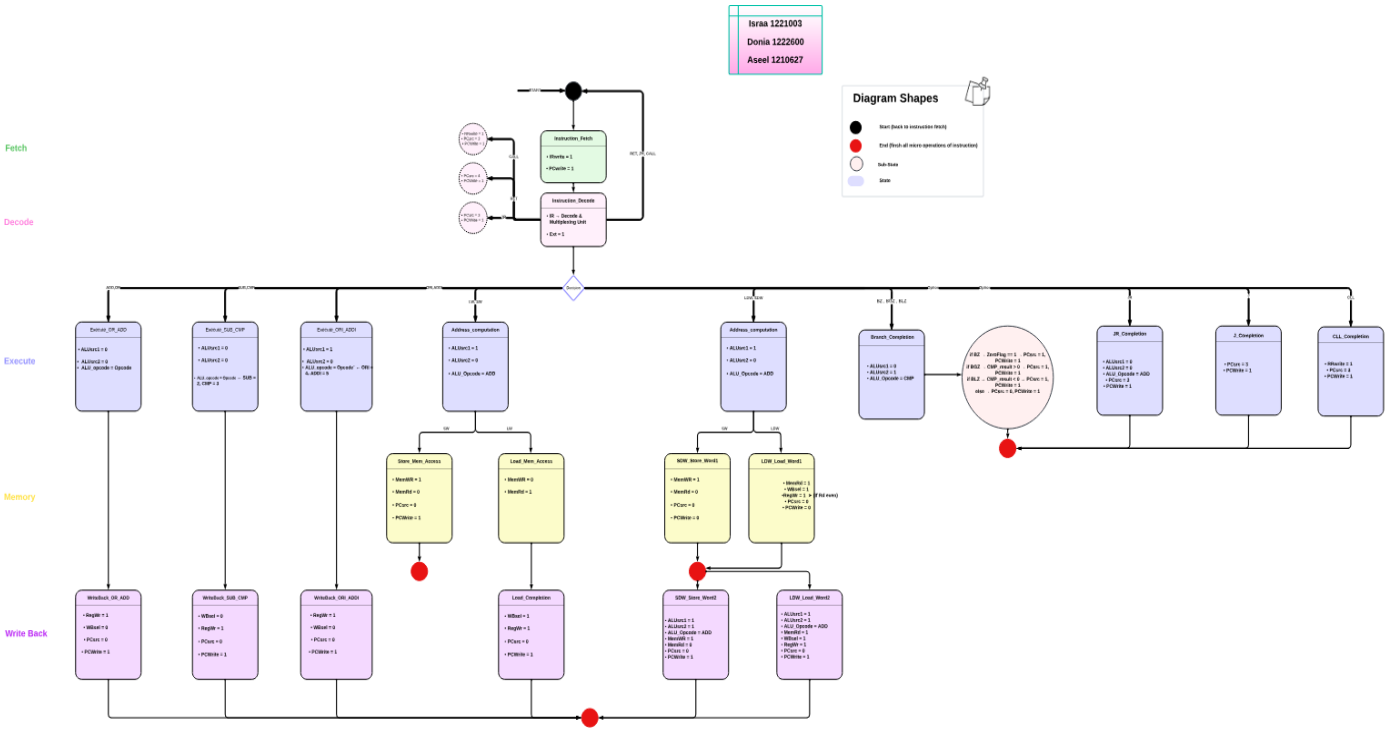


Figure 8 : Finite State Machine

To better understand the FSM state transitions in our processor, consider the **ADDI** instruction as a concrete example. Execution begins in the **Instruction Fetch** stage, where the instruction is loaded from memory and the PC is incremented. Then, the instruction moves to the **Instruction Decode** stage, where the opcode and operand fields are extracted, and the immediate value is extended.

From there, **ADDI** transitions into the **Execute_ORI_ADDI** state. In this state, the first operand is fetched from register Rs, and the immediate is used as the second ALU input. The control signal ALUsrc1 is asserted (set to 1), and the ALU is configured for the ADD operation using the Opcode value. Finally, in the **WriteBack_ORI_ADDI** state, the result is written back to the destination register Rt, and the PC is incremented again to fetch the next instruction. This sequence spans **four clock cycles**: Fetch → Decode → Execute → Write Back.

Now consider the **BZ** (Branch if Zero) instruction. It also begins with **Instruction Fetch** and **Instruction Decode**, shared with all other instructions. Then it proceeds to the **Branch_Completion** stage, where the ALU compares the values in registers Rs and Rt using a SUB operation (with ALUsrc2 = 1 to select Rt - Rs). The result updates the internal comparison flags, such as the Zero flag. The final state is a **Branch Decision** block, where based on the condition (ZeroFlag == 1), the PC is either updated to the branch target or proceeds sequentially. The **BZ** instruction completes in **four clock cycles**: Fetch → Decode → Execute → Branch Decision. It does **not** write to any register, but it does conditionally modify the PC.

To manage this behavior efficiently, control signals are divided into three dedicated units:

- **Main Control**: Implements the finite state machine. Based on the current state and instruction opcode, it determines the next state and activates high-level control signals such as memory access, write enables, and MUX selections.
- **ALU Control**: Receives the current state and opcode as input and generates ALU-specific control signals (ALU_opcode, ALUsrc1, ALUsrc2) to define the operation type and data sources.
- **PC Control**: Also driven by the current state and internal flags like ZeroFlag or CMP_result, it controls how the PC is updated (e.g., PC+1, PC + Imm, RR, or target from jump/call).

This modular separation of responsibilities simplifies the design and debugging process, enhances clarity, and ensures full compatibility with the processor's supported instruction set.

State	State Number	Abbreviation
Instruction Fetch	0	IF
Instruction Decode	1	ID
Execute OR / ADD	2	EX_OR_ADD
WriteBack OR / ADD	3	WB_OR_ADD
Execute SUB / CMP	4	EX_SUB_CMP
WriteBack SUB / CMP	5	WB_SUB_CMP
Execute ORI / ADDI	6	EX_ORI_ADDI
WriteBack ORI / ADDI	7	WB_ORI_ADDI
Address Computation	8	AC
Load Mem Access	9	L_Mem
Load Completion	10	LC
Store Mem Access	11	S_Mem
LDW Load Word 1	12	LDW1
LDW Load Word 2	13	LDW2
SDW Store Word 1	14	SDW1
SDW Store Word 2	15	SDW2
Branch Completion	16	BC
Branch Decision	17	BD
JR Completion	18	JR
J Completion	19	J
CLL Completion	20	CLL

Table 5: Finite State Machine Table

i. Designing the Main Control

To better understand the functionality of the **Main Control Unit**, we analyze its output control signals and their effects on the data path. Below is a table summarizing the one-bit control signals and the behavior of the data path depending on their values (0 or 1).

Signal	Effect when 0	Effect when 1	Effect when 2
PCWrite	No effect	PC gets updated with selected address	No effect
RRwrite	RR holds previous value	PC address is written into RR	No effect
IRwrite	IR holds previous value	Instruction memory output is written to IR	No effect
Ext	Immediate is zero-extended	Immediate is sign-extended	No effect
RegWr	No write to register file	Register file is updated with result	No effect
MemWr	No write to memory	Data is written to memory	No effect
MemRd	No read from memory	Memory output placed on data_out	No effect
ALUsrc1	ALU input A = RegA	ALU input A = 1	No effect
ALUsrc2	ALU input B = RegB	ALU input B = Immediate	No effect
WBsel	WriteBack source = ALU result	WriteBack source = Memory data	No effect
PCsrc	PC input = PC + 1	Other sources selected via PCmux	No effect

ALUAddSel	Use ALURes	Use ALURes + 1 (for LDW/SDW)	No effect
secondWordSel	Access first word	Access second word	No effect
opcode	No operation / default	Defines ALU operation	No effect
BusRead2sel	Use bus_read2	Use bus_read2 + 1	No effect

Table 6: Main Control Signals

The Main Control Unit is implemented as a finite state machine (FSM). It takes the clock signal and opcode as inputs. Based on these inputs, it determines the values of the main control signals, including the 9 primary signals (PcWrite, PcSrc[0-2], etc.).

Below is the truth table for the first five output signals of the main control unit. Each combination of inputs (opcode) determines specific outputs.

Instruction	Opcode	IRwrite	RRwrite	Ext	RegWr	MemWr	MemRd	ALUsrc1	ALUsrc2	WBsel	ALUAddSel	secondWordSel	BusRead2sel
AND	0000	0	0	X	1	0	0	0	0	0	0	X	0
ADD	0001	0	0	X	1	0	0	0	0	0	0	X	0
SUB	0010	0	0	X	1	0	0	0	0	0	0	X	0
CMP	0011	0	0	X	1	0	0	0	1	0	0	X	0
ORI	0100	0	0	1	1	0	0	1	0	0	0	X	0
ADDI	0101	0	0	1	1	0	0	1	0	0	0	X	0
LW	0110	0	0	1	1	0	1	1	0	1	0	X	0
SW	0111	0	0	1	0	1	0	1	0	X	0	X	0
LDW	1000	0	0	1	1	0	1	1	0	1	0	1	1
SDW	1001	0	0	1	0	1	0	1	0	X	0	1	1
BZ	1010	0	0	0	0	0	0	0	0	X	X	X	0
BGZ	1011	0	0	0	0	0	0	0	0	X	X	X	0
BLZ	1100	0	0	0	0	0	0	0	0	X	X	X	0
JR	1101	0	0	X	0	0	0	0	0	X	X	X	0
J	1110	0	0	X	0	0	0	0	0	X	X	X	0
CLL	1111	0	1	X	0	0	0	0	0	X	X	X	0

Table 7: Main Control Truth Table

The logical expressions that are derived for each signal:

IRwrite = IF

RRwrite = ID_CALL

Ext = ID_ADDI + ID_LW + ID_LDW + ID_SW + ID_SDW + ID_BZ + ID_BGZ + ID_BLZ

RegWr = RC + IC + FCC + LC + LDW_Load_Word2

MemWr = S_Mem + SDW_Store_Word2

MemRd = L_Mem + LDW_Load_Word1 + LDW_Load_Word2

ALUsrc1 = I_ALU + AC + LDW_Load_Word2 + SDW_Store_Word2 + FCC

ALUsrc2 = FCC + SDW_Store_Word2

WBsel = LC + FCC + IC

ALUAddSel = LDW + SDW

secondWordSel = LDW + SDW

BusRead2Sel = SDW

ii. Designing the PC Control

The PC control unit is responsible for managing the flow of program execution by determining the next instruction address to be fetched. It receives the following input:

- Opcode: The operation code of the current instruction.

Based on this input, the PC control unit generates two control signals:

- PcWrite (1 bit): Controls whether a new value is written to the PC register.
    - Enabled (1): A new address is written to the PC. This typically occurs during the execution of most instructions, except in the second cycle of LDW and SDW instructions or in invalid cases.
    - Disabled (0): The PC value remains unchanged, such as in the second cycle of LDW and SDW instructions or for invalid opcodes.

- PcSrc (3 bits): Selects the source address to be written to the PC:
    - 000: Next PC = PC + 1 (sequential execution for most instructions like OR, ADD, etc.)
    - 001: Next PC = Branch Target (used for conditional branches like BZ, BGZ, BLZ, and register-based jumps like JR)
    - 011: Next PC = Jump or Call Target Address (for unconditional jumps like J and subroutine calls like CALL)
    - 100: Next PC = Return Address (for RET instruction)

Opcode	PcWrite	PcSrc
Jump	1	011
CALL	1	011
RET	1	100
BGZ	1	001
BLZ	1	001
JR	1	001
BZ	1	001
OR	1	000
ADD	1	000
SUB	1	000
CMP	1	000
ORI	1	000
ADDI	1	000
LW	1	000
LDW	1 (cycle 1), 0 (cycle 2)	000
SW	1	000
SDW	1 (cycle 1), 0 (cycle 2)	000
Else	0	000

Table 8: PC Control Truth Table

Logical expressions that were derived are as follows:

PcWrite:

- PcWrite = (Opcode = OR) + (Opcode = ADD) + (Opcode = SUB) + (Opcode = CMP) + (Opcode = ORI) + (Opcode = ADDI) + (Opcode = LW) + (Opcode = SW) + (Opcode = LDW [cycle 1]) + (Opcode = SDW [cycle 1]) + (Opcode = BGZ) + (Opcode = BLZ) + (Opcode = JR) + (Opcode = J) + (Opcode = CALL) + (Opcode = BZ)

PcSrc[0]:

- PcSrc[0] = (Opcode = BGZ) + (Opcode = BLZ) + (Opcode = JR) + (Opcode = J) + (Opcode = CALL) + (Opcode = BZ)

PcSrc[1]:

- PcSrc[1] = (Opcode = J) + (Opcode = CALL)

PcSrc[2]:

- PcSrc[2] = 0

iii. Designing the ALU Control

Instruction	ALUOp
OR	01
ADD	00
SUB	10
CMP	10
ORI	01
ADDI	00
LW	00
SW	00
LDW	00

SDW	00
BZ	xx
BGZ	xx
BLZ	xx
JR	xx
J	xx
CALL	xx

Table 9: ALU Control Truth Table

Logical expressions that were derived are as follows:

ALUOp[0] = (Opcode = OR) + (Opcode = ORI)

ALUOp[1] = (Opcode = SUB) + (Opcode = CMP)

12. Simulation and Testing

The simulation and testing phase of the project were conducted using Active-HDL, a versatile platform for simulating and debugging digital designs. The process started with simulating individual processor components, including the arithmetic logic unit (ALU), control unit, and separate instruction and data memory modules. By generating waveforms for these components, we verified their functionality and ensured that each unit operated as expected. This step was essential for detecting and resolving potential issues at the component level, enabling efficient debugging.

After validating the components, we proceeded to simulate the entire 5-stage pipelined processor. Using a dedicated testbench module, we developed a comprehensive test program that included all supported instruction types (OR, ADD, SUB, CMP, ORI, ADDI, LW, SW, LDW, SDW, BZ, BGZ, BLZ, JR, J, CALL) to evaluate the processor's performance. This program was designed to test the processor's ability to execute instructions across multiple cycles, particularly accounting for the two-cycle execution of LDW and SDW instructions, in line with the intended pipelined design. By analyzing the processor's behavior through waveforms, we confirmed its correctness and verified that the execution cycle counts matched our design expectations.

a. Processor's Components Simulation

2.1.1. ALU

- The ALU component handles both logical and arithmetic operations, supporting five functions: Addition (ADD), Subtraction (SUB), AND (via OR and ORI), Comparison (CMP), and address calculation (for LW, SW, LDW, and SDW).

```
[2025-06-10 16:40:56 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
VCD info: dumpfile alu_tb.vcd opened for output.
Testing OR operation
OR: a=f0f0f0f0, b=0f0f0f0f, result=ffffffff (Expected: ffffffff)
Testing ADD operation
ADD: a=00000001, b=00000002, result=00000003, overflow=0 (Expected: 00000003, 0)
ADD Overflow: a=7fffffff, b=7fffffff, result=fffffffe, overflow=1 (Expected: fffffffe, 1)
Testing SUB operation
SUB: a=00000005, b=00000003, result=00000002, overflow=0 (Expected: 00000002, 0)
SUB Overflow: a=80000000, b=00000001, result=7fffffff, overflow=1 (Expected: 7fffffff, 1)
Testing CMP operation
CMP Equal: a=00000005, b=00000005, result=00000000, zero=1, positive=0, negative=0 (Expected: 0, 1, 0, 0)
CMP Greater: a=00000005, b=00000003, result=00000001, zero=0, positive=1, negative=0 (Expected: 1, 0, 1, 0)
CMP Less: a=00000003, b=00000005, result=ffffffff, zero=0, positive=0, negative=1 (Expected: -1, 0, 0, 1)
Testing ORI with immediate
ORI: a=f0f0f0f0, b=00000fff, result=f0f0ffff (Expected: f0f0ffff)
Testing ADDI with negative immediate
ADDI: a=00000005, b=fffffffe, result=00000003 (Expected: 00000003)
Testing ADD with zero
ADD Zero: a=00000000, b=00000000, result=00000000, overflow=0 (Expected: 00000000, 0)
Testing CMP with large numbers
CMP Large: a=7fffffff, b=80000000, result=00000001, zero=0, positive=1, negative=0 (Expected: 1, 0, 1, 0)
testbench.sv:83: $finish called at 120 (1s)
```

Figure 9 : ALU Test

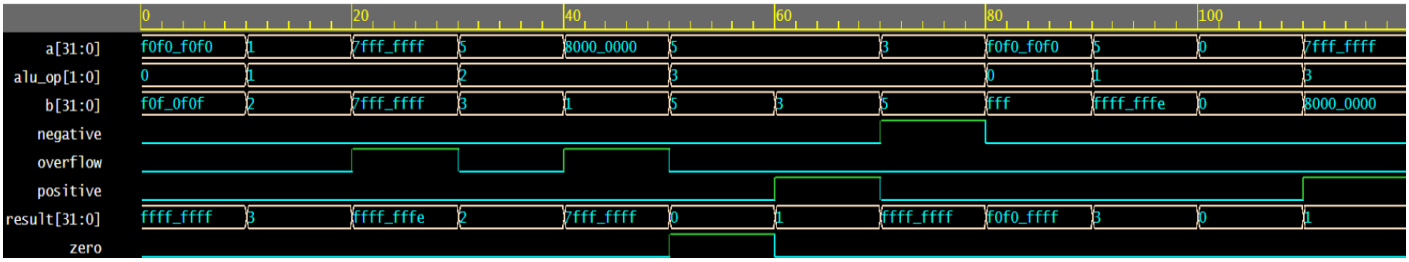


Figure 10: ALU WaveForm

The testbench verifies the functionality of the Arithmetic Logic Unit (ALU) across a variety of operations. It includes standard logical and arithmetic operations such as OR, ADD, and SUB, and confirms correctness by checking both the result and the overflow flag in edge cases. The CMP instruction was tested thoroughly for equality, greater-than, and less-than comparisons, confirming correct behavior of the zero, positive, and negative flags. Immediate instructions like ORI and ADDI were also verified, including scenarios involving negative and zero immediate values. Additionally, extreme-value tests (e.g., large number comparisons) were included to validate stability under edge conditions.All tests passed successfully, indicating that the ALU behaves as expected across all supported operations.

2.1.2. Register File

```
[2025-06-10 17:59:06 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
VCD info: dumpfile register_file_tb.vcd opened for output.
Testing Write and Read
Write R1=12345678, Read R1=12345678 (Expected: 12345678)
Testing LDW Exception
LDW Odd Rd: exception=1 (Expected: 1)
Testing SDW Exception
SDW Odd Rd: exception=1 (Expected: 1)
Testing CLL
CLL: R14=000000ff (Expected: 000000ff)
testbench.sv:76: $finish called at 80 (1s)
Done
```

Figure 11: Register File Test

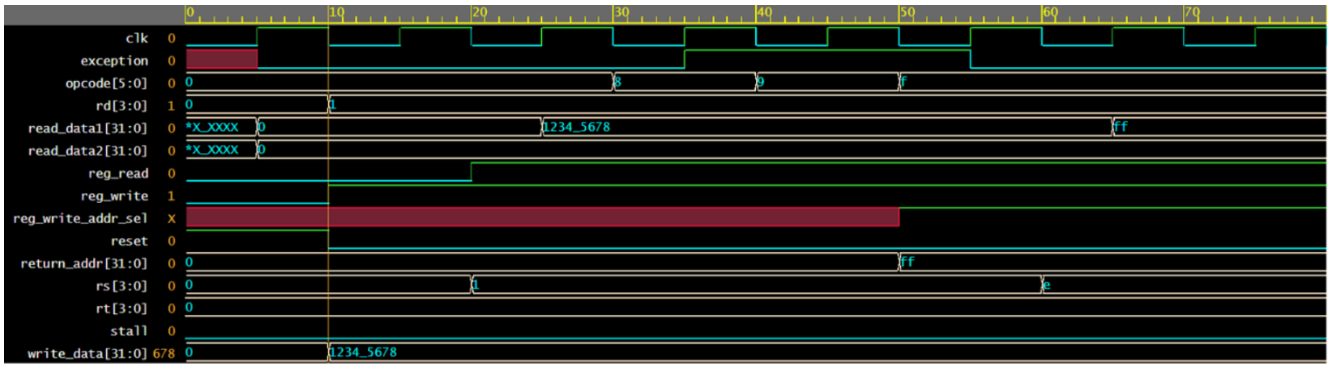


Figure 12: Register File WaveForm

This testbench validates key functionalities of the register file, including writing and reading data, exception handling, and special instruction behavior. A basic write-read test confirms that data written to a register is accurately retrieved. Exception handling is verified by triggering errors for LDW and SDW instructions when the destination/source register is odd, which is an illegal condition in the design. The CLL instruction is also tested, confirming that the return address is correctly stored in register R14.

All outputs match expected results, confirming the register file operates correctly and enforces instruction-specific constraints.

2.1.3. Instruction Memory

```
[2025-06-10 22:41:50 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Testing Instruction Memory
Addr=00000000, Instr=0048c000 (OR R1, R2, R3)
Addr=00000004, Instr=05048000 (ADD R4, R1, R2)
Addr=00000008, Instr=09504000 (SUB R5, R4, R1)
Addr=0000000c, Instr=0d950000 (CMP R6, R5, R4)
Addr=00000010, Instr=04760004 (ORI R7, R6, 4)
Addr=00000014, Instr=05870002 (ADDI R8, R7, 2)
Addr=00000018, Instr=06980004 (LW R9, 4(R8))
Addr=0000001c, Instr=079a0004 (SW R9, 4(R10))
Addr=00000020, Instr=08a90004 (LDW R10, 4(R9))
Addr=00000024, Instr=09ab0004 (SDW R10, 4(R11))
Addr=00000028, Instr=0a0a0002 (BZ R10, 2)
Addr=0000002c, Instr=0b0a0002 (BGZ R10, 2)
Addr=00000030, Instr=0c0a0002 (BLZ R10, 2)
Addr=00000034, Instr=0d0b0000 (JR R11)
Addr=00000038, Instr=0e000002 (J 2)
Addr=0000003c, Instr=0f000002 (CLL 2)
testbench.sv:50: $finish called at 160 (1s)
Done
```

Figure 13: Instruction Memory Test

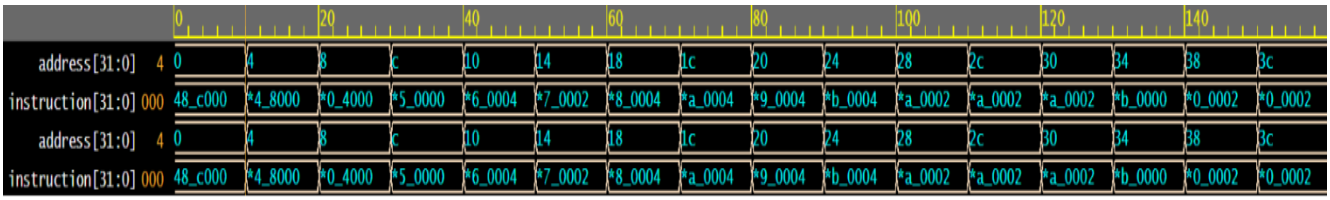


Figure 14: Instruction Memory Wave Form

To verify the functionality of the Instruction Memory, we initialized it with a complete sequence of instructions covering all required opcodes in the project. The simulation output shows each instruction being fetched correctly from the expected address and decoded accurately. This confirms that the Instruction Memory is functioning properly and capable of delivering the correct instruction stream to the processor for execution.

2.1.4. MUX 2x1

```
[2025-06-10 17:12:40 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
VCD info: dumpfile tb_mux2x1.vcd opened for output.
Test 1: sel=0, in0=12345678, in1=abcdef01, out=12345678
Test 2: sel=1, in0=12345678, in1=abcdef01, out=abcdef01
Test 3: sel=0, in0=00000000, in1=ffffffff, out=00000000
Test 4: sel=1, in0=00000000, in1=ffffffff, out=ffffffff
testbench.sv:44: $finish called at 50 (1s)
Done
```

Figure 15: MUX 2x1 Test

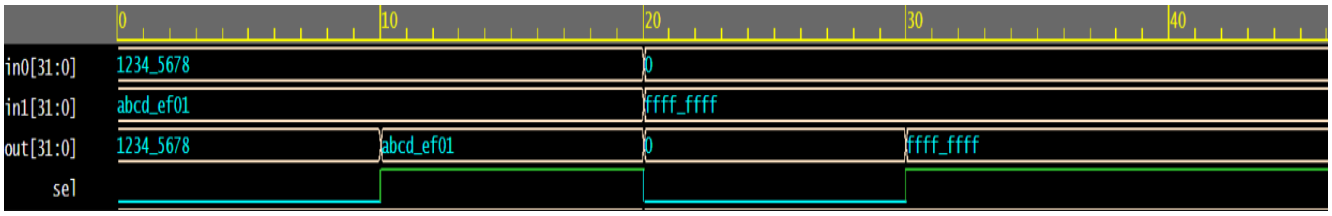


Figure 16: MUX 2x1 Wave Form

This testbench verifies the behavior of a 2-to-1 multiplexer by evaluating multiple combinations of inputs and select signals. When sel=0, the output correctly follows in0, and when sel=1, the output switches to in1. Various test cases were used, including edge values such as all-zeros and all-ones, to confirm stable operation across different input scenarios. The multiplexer correctly routed the selected input to the output in all cases, validating its functional correctness.

2.1.5. Extender

```
Time=0 opcode=000000 imm_in=0000 imm_out=00000000
Time=20 opcode=000100 imm_in=1234 imm_out=00001234
Time=30 opcode=000100 imm_in=3fff imm_out=00003fff
Time=40 opcode=000101 imm_in=1234 imm_out=00001234
Time=50 opcode=000101 imm_in=3234 imm_out=ffff234
Time=60 opcode=000110 imm_in=1fff imm_out=00001fff
Time=70 opcode=000110 imm_in=2000 imm_out=ffffe000
Time=80 opcode=001010 imm_in=0100 imm_out=00000100
Time=90 opcode=001010 imm_in=3f00 imm_out=ffffff00
Time=100 opcode=001110 imm_in=2000 imm_out=ffffe000
Testbench completed successfully!
testbench.sv:85: $finish called at 130 (1s)
Done
```

Figure 17: MUX Extender Test

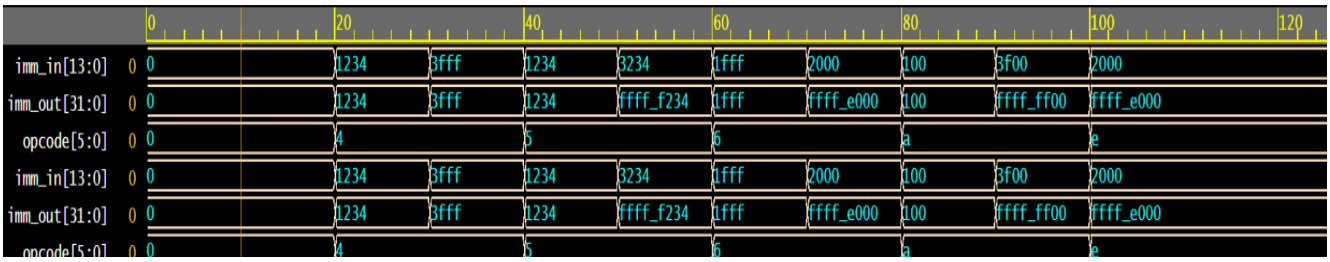


Figure 18: MUX Extender Wave Form

How it appeared: Due to a flaw in the display of imm_in in Edaplayground, where negative values (such as F234, E000, FF00) are shown incorrectly (e.g., 3234, 2000, 3f00), likely caused by an issue in Edaplayground's conversion of negative values. imm_out is working: Yes, imm_out operates correctly based on the actual values assigned in the code, confirming that the flaw is not in the internal logic of the extender module.

This testbench validates the behavior of the immediate extender unit under different opcodes. The component successfully performed both zero-extension and sign-extension based on the opcode input. For unsigned operations like ORI and ANDI, the immediate was zero-extended,

while for signed instructions such as ADDI and LW, the immediate was correctly sign-extended. The test cases included edge values (e.g., 0x1FFF, 0x2000, 0x3F00) to confirm that both positive and negative immediates are handled properly. All results matched the expected outputs, confirming the extender's correctness.

2.1.6. PC

```
[2025-06-10 17:19:36 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
VCD info: dumpfile program_counter_tb.vcd opened for output.
Testing PC Increment
PC=00000001 (Expected: 00000001)
Testing Branch
Branch: PC=00000009 (Expected: 00000009)
Testing Jump
Jump: PC=00000011 (Expected: 00000011)
CLL Return Addr: 00000012 (Expected: 00000012)
Testing JR
JR: PC=00000014 (Expected: 00000014)
testbench.sv:68: $finish called at 50 (1s)
Done
```

Figure 19: PC Test

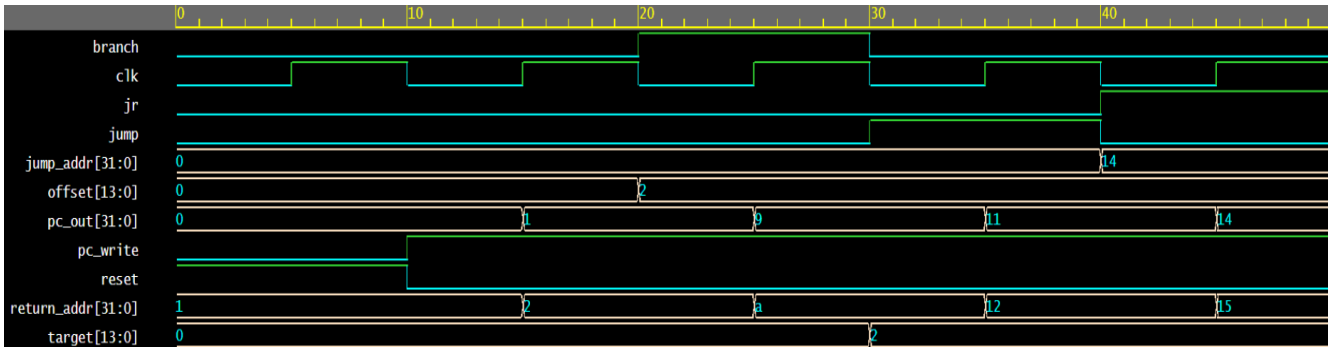


Figure 20: PC Wave Form

The testbench verifies the correct functionality of the program counter across multiple control paths. It checks standard increment behavior, conditional branches, direct jumps, call return addresses, and jump register (JR) instructions. Each tested case correctly updates the PC value according to the expected logic. For example, the PC increments by 1, jumps to the specified address on branch and jump instructions, stores the return address for CLL, and correctly follows the register content for JR. All outcomes matched expected results, confirming reliable PC behavior.

2.1.8. Data Memory

```
[2025-06-10 17:06:10 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Testing LW
LW: addr=00000010, read_data=44444444 (Expected: 44444444)
Testing SW
SW: addr=00000020, read_data=deadbeef (Expected: deadbeef)
Testing LDW
LDW first word: addr=00000010, read_data=44444444 (Expected: 44444444)
LDW second word: addr=00000010, read_data=55555555 (Expected: 55555555)
Testing SDW
SDW first word: addr=0000000c, read_data=cafebabe (Expected: cafebabe)
SDW second word: addr=0000000c, read_data=decaf000 (Expected: decaf000)
testbench.sv:86: $finish called at 90 (1s)
Done
```

Figure 21: Data Memory Test

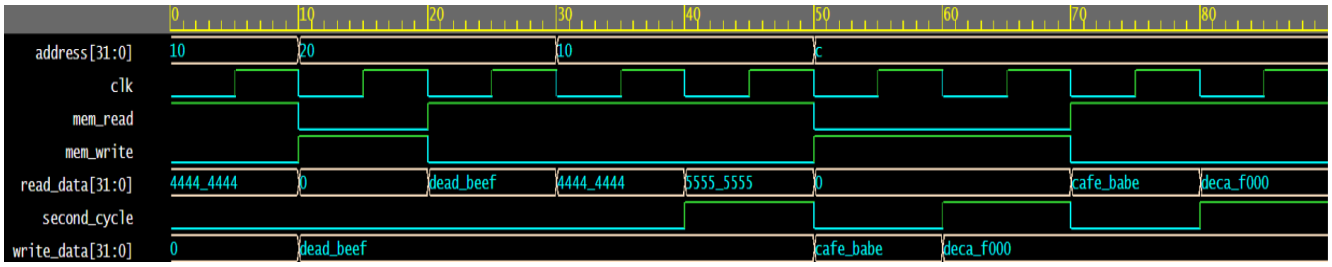


Figure 22: Data Memory Wave Form

This test validates standard and word-based memory access operations. It confirms correct LW and SW behavior for single-word reads/writes, returning the expected values. It also tests double-word instructions LDW and SDW, verifying that two consecutive memory locations are correctly accessed and updated. All outputs matched the expected results, ensuring the memory module accurately handles aligned access, dual-word operations, and data consistency across operations.

2.1.9. Control Unit

To verify the correctness of the control unit, we created a comprehensive testbench that covers all instructions required in the project. Each instruction was assigned a unique opcode, and we monitored the control signals generated by the control unit in response. As shown in the waveform and simulation logs, the signals such as pc_write, reg_write, mem_read, mem_write, and others were correctly asserted for each instruction at the appropriate states. The simulation confirms that the control unit transitions correctly across states and handles instruction execution according to the defined FSM and datapath behavior.

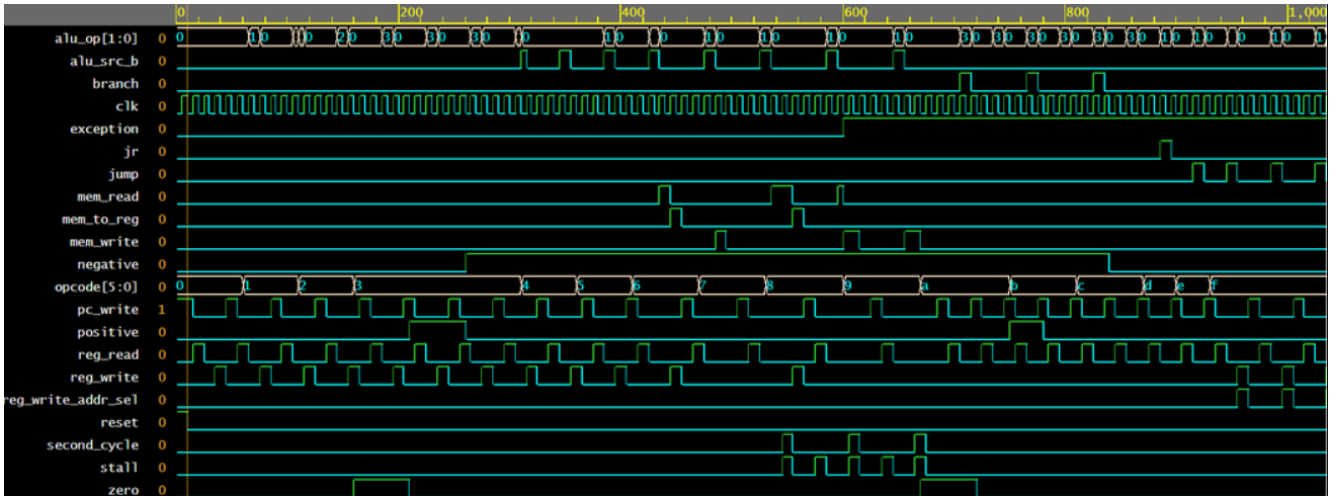


Figure 23: Control Unit Test

```
[2025-06-10 22:20:47 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Time=0 | State=0 | Opcode=0 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 | s
Testing OR (Opcode=0)
Time=15 | State=1 | Opcode=0 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=25 | State=2 | Opcode=0 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=35 | State=4 | Opcode=0 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=45 | State=0 | Opcode=0 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=55 | State=1 | Opcode=0 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing ADD (Opcode=1)
Time=60 | State=1 | Opcode=1 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=65 | State=2 | Opcode=1 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=75 | State=4 | Opcode=1 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=85 | State=0 | Opcode=1 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=95 | State=1 | Opcode=1 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=105 | State=2 | Opcode=1 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Testing SUB (Opcode=2)
Time=110 | State=2 | Opcode=2 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=115 | State=4 | Opcode=2 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=125 | State=0 | Opcode=2 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=135 | State=1 | Opcode=2 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=145 | State=2 | Opcode=2 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=155 | State=4 | Opcode=2 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
```

```
a.out
 mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=10 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=10 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_read=0 | mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
Testing CMP (Opcode=3)
Time=160 | State=4 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=165 | State=0 | Opcode=3 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=175 | State=1 | Opcode=3 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=185 | State=2 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=195 | State=4 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=205 | State=0 | Opcode=3 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=210 | State=0 | Opcode=3 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=215 | State=1 | Opcode=3 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=225 | State=2 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=235 | State=4 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=245 | State=0 | Opcode=3 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=255 | State=1 | Opcode=3 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=260 | State=1 | Opcode=3 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=265 | State=2 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=275 | State=4 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=285 | State=0 | Opcode=3 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=295 | State=1 | Opcode=3 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=305 | State=2 | Opcode=3 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing ORI (Opcode=4)
Time=310 | State=2 | Opcode=4 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=315 | State=4 | Opcode=4 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=325 | State=0 | Opcode=4 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
```

```
Testing ORI (Opcode=4)
Time=310 | State=2 | Opcode=4 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=315 | State=4 | Opcode=4 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=325 | State=0 | Opcode=4 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=335 | State=1 | Opcode=4 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=345 | State=2 | Opcode=4 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=355 | State=4 | Opcode=4 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing ADDI (Opcode=5)
Time=360 | State=4 | Opcode=5 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=365 | State=0 | Opcode=5 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=375 | State=1 | Opcode=5 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=385 | State=2 | Opcode=5 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=395 | State=4 | Opcode=5 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=405 | State=0 | Opcode=5 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing LW (Opcode=6)
Time=410 | State=0 | Opcode=6 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=415 | State=1 | Opcode=6 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=425 | State=2 | Opcode=6 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=435 | State=3 | Opcode=6 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=1 | mem_write=0 |
Time=445 | State=4 | Opcode=6 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=455 | State=0 | Opcode=6 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=1 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
Testing SW (Opcode=7)
Time=470 | State=1 | Opcode=7 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=475 | State=2 | Opcode=7 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=485 | State=3 | Opcode=7 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=1 |
Time=495 | State=4 | Opcode=7 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=505 | State=0 | Opcode=7 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=515 | State=1 | Opcode=7 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=525 | State=2 | Opcode=7 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing LDW (Opcode=8, no exception)
Time=530 | State=2 | Opcode=8 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=535 | State=3 | Opcode=8 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=1 | mem_write=0 |
Time=545 | State=5 | Opcode=8 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=1 | mem_write=0 |
Time=555 | State=4 | Opcode=8 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=565 | State=0 | Opcode=8 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=575 | State=1 | Opcode=8 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=585 | State=2 | Opcode=8 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=595 | State=3 | Opcode=8 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=1 | mem_write=0 |
Testing SDW (Opcode=9, with exception)
Time=600 | State=3 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=1 |
Time=605 | State=5 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=1 |
Time=615 | State=4 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=625 | State=0 | Opcode=9 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=635 | State=1 | Opcode=9 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
```

```
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=1 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=1 | mem_to_reg=0 | stall=0

| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=1 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=1 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_write=1 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=1 | second_cycle=1 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1
Testing SDW (Opcode=9, with exception)
Time=600 | State=3 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=1 |
Time=605 | State=5 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=1 |
Time=615 | State=4 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=625 | State=0 | Opcode=9 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=635 | State=1 | Opcode=9 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=645 | State=2 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=655 | State=3 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=1 |
Time=665 | State=5 | Opcode=9 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=1 |
Testing BZ (Opcode=10, zero=1)
Time=670 | State=5 | Opcode=10 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=675 | State=4 | Opcode=10 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=685 | State=0 | Opcode=10 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=695 | State=1 | Opcode=10 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=705 | State=2 | Opcode=10 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=715 | State=0 | Opcode=10 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Testing BZ (Opcode=10, zero=0)
Time=720 | State=0 | Opcode=10 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=725 | State=1 | Opcode=10 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=735 | State=2 | Opcode=10 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
Time=745 | State=0 | Opcode=10 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0
| mem_write=1 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=1 | second_cycle=1 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=01 | alu_src_b=1 | mem_to_reg=0 | stall=0
| mem_write=1 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=1 | second_cycle=1 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1

| mem_write=0 | second_cycle=1 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=1
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=1 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
```

```
Testing BGZ (Opcode=11, positive=1)
Time=750 | State=0 | Opcode=11 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=755 | State=1 | Opcode=11 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=765 | State=2 | Opcode=11 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=775 | State=0 | Opcode=11 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing BGZ (Opcode=11, positive=0)
Time=780 | State=0 | Opcode=11 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=785 | State=1 | Opcode=11 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=795 | State=2 | Opcode=11 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=805 | State=0 | Opcode=11 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing BLZ (Opcode=12, negative=1)
Time=810 | State=0 | Opcode=12 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=815 | State=1 | Opcode=12 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=825 | State=2 | Opcode=12 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=835 | State=0 | Opcode=12 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing BLZ (Opcode=12, negative=0)
Time=840 | State=0 | Opcode=12 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=845 | State=1 | Opcode=12 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=855 | State=2 | Opcode=12 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=865 | State=0 | Opcode=12 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing JR (Opcode=13)
Time=870 | State=0 | Opcode=13 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=875 | State=1 | Opcode=13 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=1 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=1 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=11 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
Testing JR (Opcode=13)
Time=870 | State=0 | Opcode=13 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=875 | State=1 | Opcode=13 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=885 | State=2 | Opcode=13 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=895 | State=0 | Opcode=13 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing J (Opcode=14)
Time=900 | State=0 | Opcode=14 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=905 | State=1 | Opcode=14 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=915 | State=2 | Opcode=14 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=925 | State=0 | Opcode=14 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing CLL (Opcode=15)
Time=930 | State=0 | Opcode=15 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=935 | State=1 | Opcode=15 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=945 | State=2 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=955 | State=4 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=1 | mem_read=0 | mem_write=0 |
Time=965 | State=0 | Opcode=15 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=975 | State=1 | Opcode=15 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Testing CLL (Opcode=15, verify R14 write)
Time=985 | State=2 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=995 | State=4 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=1 | mem_read=0 | mem_write=0 |
Time=1005 | State=0 | Opcode=15 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=1015 | State=1 | Opcode=15 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=1025 | State=2 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
```

```
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=1 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=1 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=1 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0

| second_cycle=0 | branch=0 | jump=1 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0
| second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
 | second_cycle=0 | branch=0 | jump=1 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0

Testing CLL (Opcode=15, verify R14 write)
Time=985 | State=2 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=995 | State=4 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=1 | mem_read=0 | mem_write=0 |
Time=1005 | State=0 | Opcode=15 | pc_write=1 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=1015 | State=1 | Opcode=15 | pc_write=0 | reg_read=1 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Time=1025 | State=2 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=0 | reg_write_addr_sel=0 | mem_read=0 | mem_write=0 |
Test completed
Time=1035 | State=4 | Opcode=15 | pc_write=0 | reg_read=0 | reg_write=1 | reg_write_addr_sel=1 | mem_read=0 | mem_write=0 |
testbench.sv:175: $finish called at 1040 (1s)

mem_write=0 | second_cycle=0 | branch=0 | jump=1 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0
mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
| mem_write=0 | second_cycle=0 | branch=0 | jump=1 | jr=0 | alu_op=01 | alu_src_b=0 | mem_to_reg=0 | stall=0

| mem_write=0 | second_cycle=0 | branch=0 | jump=0 | jr=0 | alu_op=00 | alu_src_b=0 | mem_to_reg=0 | stall=0
```

Figure 24: Control Unit Wave Form

b. Processor Simulation

Test Case 1 :

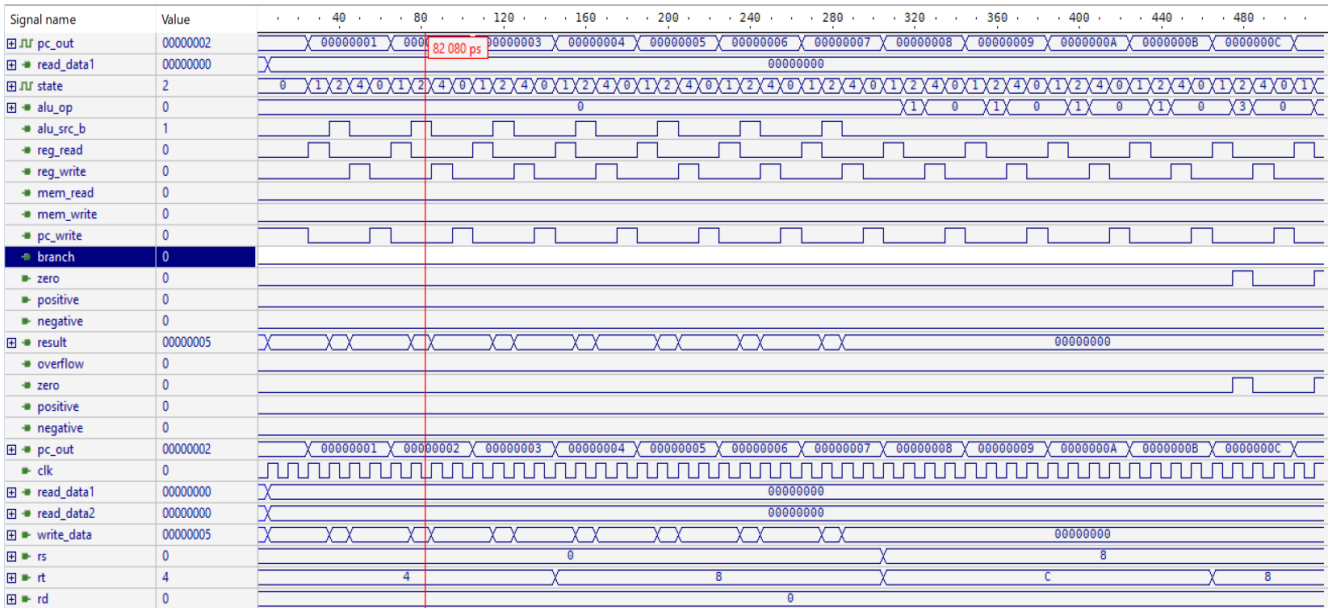
ADDI R1, R0, 5 // R1 = 5

ADDI R2, R0, 7 // R2 = 7

ADD R3, R1, R2 // R3 = R1 + R2 = 12

CMP R1, R2 // Sets negative = 1 (5 < 7)

BZ R1, 2 // Should NOT branch (zero=0)



Explanation of Waveform Execution:

- **At time 0ns**, instruction ADDI R1, R0, 1 is fetched and begins execution. This instruction loads the value 1 into register R1.
- **At time 40ns**, the instruction ADDI R2, R0, 2 is executed, storing 2 into R2.
- **At time 80ns**, the instruction OR R3, R1, R2 performs a bitwise OR between R1=1 and R2=2, resulting in R3 = 3.

- **At time 120ns**, the instruction ADD R4, R1, R2 computes R4 = 1 + 2 = 3 and writes it to register R4.
- **At time 160ns**, the instruction SUB R5, R4, R2 computes R5 = 3 - 2 = 1.
- **At time 200ns**, the instruction CMP R6, R5, R4 compares R5 and R4 (1 < 3) → setting negative = 1, while zero and positive flags remain low.
- **At time 240ns**, the instruction ORI R7, R6, 0 is executed, resulting in R7 = R6 | 0 = -1.

Important Notes:

- The waveform clearly shows **ALU results (result)** being generated at the correct times and written to the register file during the **WRITE_BACK** state (state 4).
- The control signals like alu_op, reg_write, reg_read, and alu_src_b switch appropriately for each instruction.
- The **comparison flags** (zero, positive, negative) are updated correctly during the CMP instruction.
- The **program counter (pc_out)** increments by 1 each time, showing sequential execution.

Test Case 2 :

ADDI R1, R0, 1 ; R1 = 1

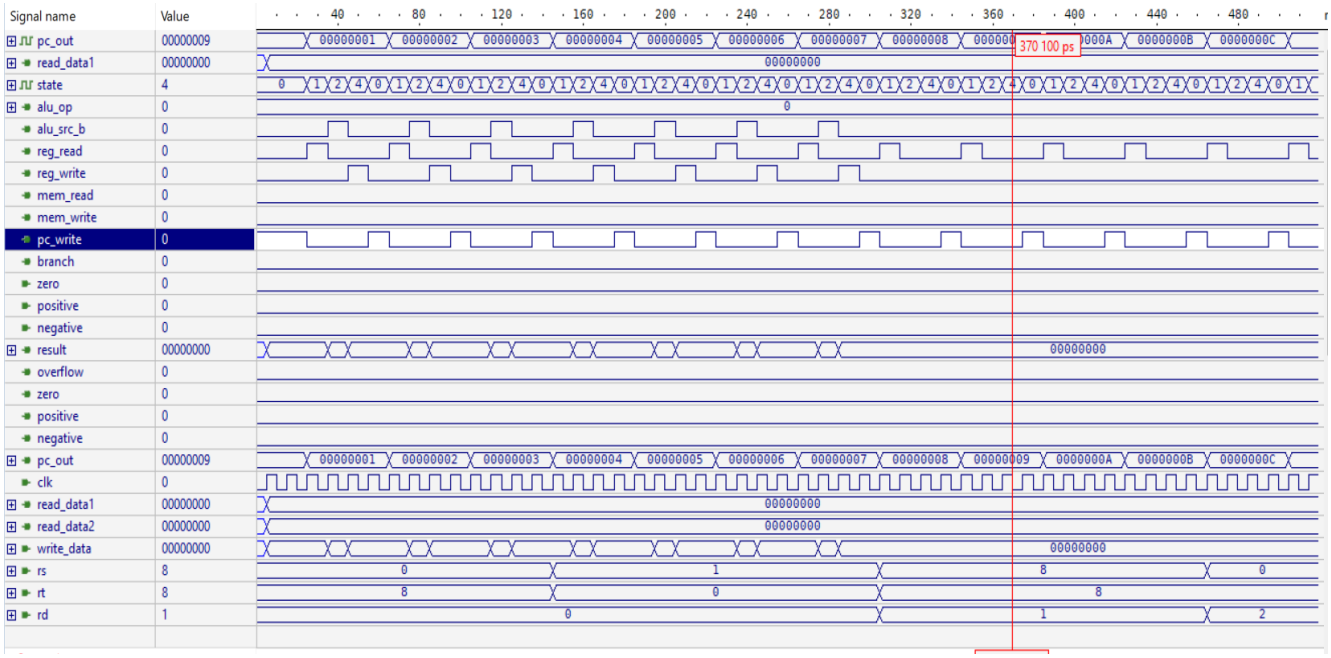
ADDI R2, R0, 4 ; R2 = 4

LW R3, 4(R1) ; R3 = Mem[R1 + 4]

SW R4, 4(R1) ; Mem[R1 + 4] = R4

LDW R5, 4(R1) ; R5 = Mem[R1 + 4] + Mem[R1 + 5]

SDW R6, 4(R1) ; Mem[R1 + 4] = R6, Mem[R1 + 5] = R6



Explanation:

- **At time 50–100 ns:**
  The instruction ADDI R1, R0, 1 is executed. The ALU adds 0 + 1, and the result is stored into register R1.
- **At time 110–160 ns:**
  The instruction ADDI R2, R0, 4 is executed. R2 is loaded with value 4.
- **At time 170–220 ns:**
  The instruction LW R3, 4(R1) is executed. The address is calculated as R1 + 4 = 5.
  The mem_read signal goes high, and the value at memory[5] is loaded into R3.

- **At time 230–280 ns:**
  The instruction SW R4, 4(R1) is executed. The value of R4 is written to memory[5].
  The mem_write signal is asserted during this cycle.
- **At time 290–370 ns:**
  The LDW R5, 4(R1) instruction starts. This is a multi-cycle load.
  You can observe the control unit activating the stall and second_cycle signals.
  Memory is read in two back-to-back cycles: from memory[5] and memory[6].
- **At time 380–460 ns:**
  The SDW R6, 4(R1) is executed. The mem_write signal is active across two consecutive cycles,
  confirming that two memory locations (memory[5] and memory[6]) are written.

Test Case 3 :

ADDI R1, R0, 1 ; R1 = 1

ADDI R2, R0, 1 ; R2 = 1

CMP R3, R1, R2 ; R3 = (R1 - R2), zero = 1

BZ R3, 2 ; if zero == 1, jump PC + 2

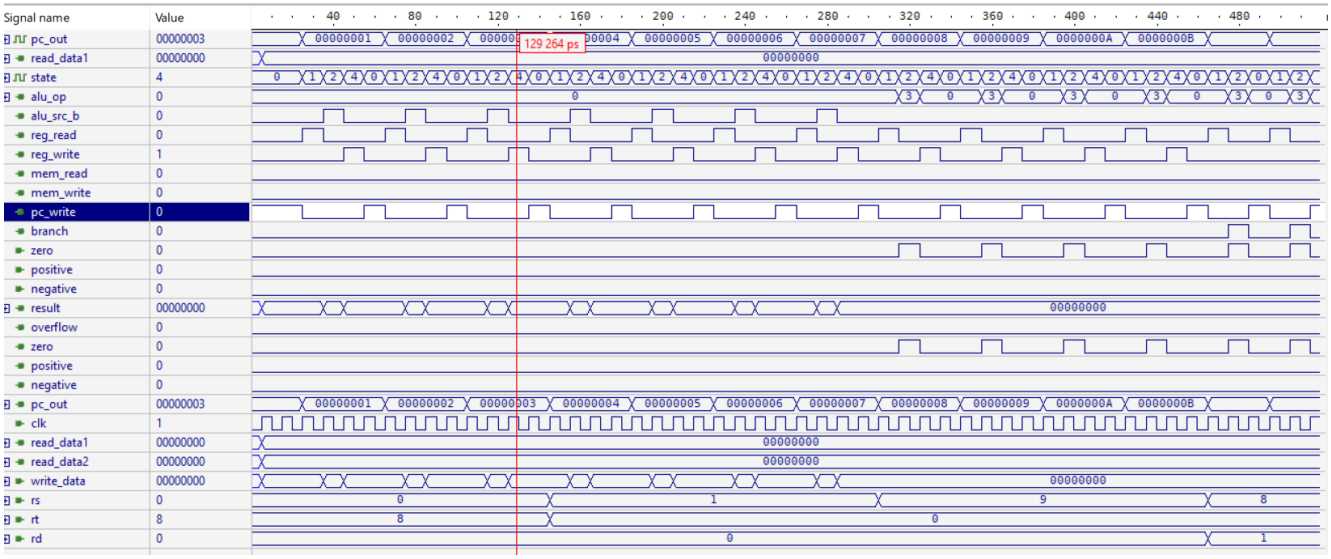
ADD R7, R1, R2 ; R7 = R1 + R2

J 1 ; jump PC + 1

ADDI R8, R0, 4 ; R8 = 4

CLL 1 ; jump PC + 1 and store return address in R14

ADDI R9, R0, 9 ; R9 = 9



Explanation:

- **At time ~130 ps**: CMP instruction is executed between R1 and R2. Since both registers contain 1, the zero signal is asserted (1), while positive and negative are 0.
- **Immediately after**: the BZ instruction checks the zero flag, and since it is 1, it branches and skips the next instruction (ADD R7, R1, R2). This is confirmed by seeing that the instruction at PC=00000005 is skipped in control signals.
- **At ~220 ps**: The J 1 instruction is executed, causing another jump and skipping ADDI R8, R0, 4.

- **At ~270 ps**: The CLL 1 instruction is executed, and as seen, reg_write is set and destination register is R14 (rd = 14), writing return address into it.
- **At ~300 ps onward**: The instruction ADDI R9, R0, 9 is executed. You can see that R9 is being written with the correct value 9 (write_data = 00000009 and rd = 9).

Important Point:

This test case validates the correct behavior of **branch instructions and jump handling**. The waveform confirms:

- **Correct assertion of condition flags (zero)** by CMP
- **Branch (BZ)** skips the ADD instruction as expected
- **Jump (J)** instruction successfully jumps forward
- **CLL** saves return address in R14
- **Flow returns to ADDI after jump**, showing full control flow is handled without stalls

Teamwork

The entire project was implemented collaboratively, with all team members working on every part together and contributing equally throughout the development process. We did not divide the tasks rigidly; instead, we tackled each challenge as a team whether designing modules, writing code, debugging issues, or running simulations.

Decisions were made collectively, and constant communication ensured that everyone was aligned. We shared responsibilities across all components: from datapath design to control logic, from testbench development to waveform analysis. Every milestone was achieved through group effort, and the outcome reflects balanced and equal contributions from all members.

Conclusion

This project successfully demonstrates a complete multi-cycle processor implementation, supporting a custom instruction set and verifying correct execution through simulation and waveform analysis. We implemented and tested each module including the ALU, control unit, data and instruction memory, extender, program counter, and register file ensuring full functional integration.

All supported instructions were executed and validated through realistic test cases, confirming correct control signal behavior and instruction flow. The use of waveform tracing provided clear visibility into internal operations and helped verify system correctness in detail.

This experience enhanced our practical understanding of datapath design, finite state machines, control logic, and system-level integration using SystemVerilog. It also highlighted the value of clean modular design, rigorous testing, and teamwork in building complex digital systems from scratch.