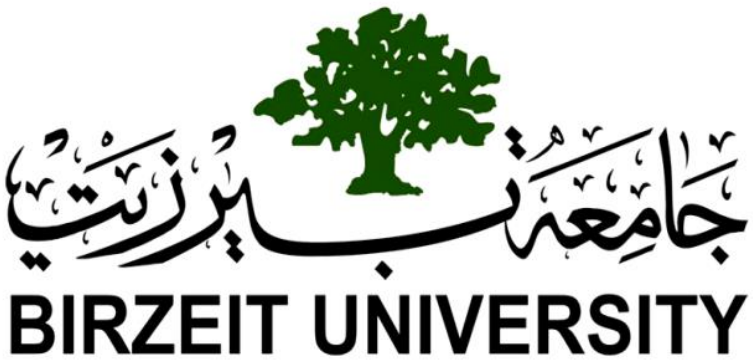

Faculty of Engineering & Technology

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: Malak Ammar. **Partner1_Number:** 1211470.

Partner2_Name: Jana Sawalmah. **Partner1_Number:** 1212467.

Partner3_Name: Sarah Allahalih. **Partner1_Number:** 1211083.

Instructor: Ayman Hroub.

Section: 2.

Date: 26/12/2024.

Abstract

This project presents the design and verification of a multi-cycle RISC processor using Verilog, demonstrating how computer architecture principles guide microprocessor component design. The development process begins with defining the instruction set architecture (ISA) and analyzing its requirements, enabling informed decisions about datapath components and structure. The project adopts a multi-cycle datapath implementation, utilizing a state machine to manage instruction execution across multiple stages. This report details the complete design process, from initial planning to verification, highlighting the step-by-step approach taken to implement and test each instruction.

Table of Contents

Abstract I

Table of figures III

List of tables IV

1. Design and Implementation 1

1.1. Processor Specifications and Overview 1

1.2. Instruction Format & RTL Operations 4

1.2.1. Instruction Formats and Interpretations 4

1.2.2. Instructions' Encoding 6

1.2.3. Instruction RTL Micro Operations 8

1.3. Functional Units and Components 11

1.4. Constructing the Datapath 19

1.5. Designing the Control Unit 22

1.5.1. Designing the Main Control 25

1.5.2. Designing the PC Control 29

1.5.3. Designing the ALU Control 31

2. Simulation and Testing 34

2.1. Processor's Components Simulation 34

2.1.1. ALU 34

2.1.2. Register File 36

2.1.3. Instruction Memory 37

2.1.4. MUX 8x1 38

2.1.5. Extender 38

2.1.6. Adder 39

2.1.7. Decode and Multiplexing 39

2.1.8. Data Memory 40

2.1.9. Zero Comparator – NOR 41

2.2. Processor Simulation 42

Teamwork 56

Conclusion 57

Table of figures

Figure 1: Instruction Memory *12*
Figure 2: Data Memory *12*
Figure 3: Register File *13*
Figure 4: ALU *14*
Figure 5: Decoding and Multiplexing Unit *16*
Figure 6: Extender Unit *16*
Figure 7: NOR Unit *18*
Figure 8: Datapath *19*
Figure 9: Datapath with controls *21*
Figure 10: Finite State Machine *22*
Figure 11: ADD Simulation *34*
Figure 12: SUB Simulation *35*
Figure 13: AND Simulation *35*
Figure 14: SLL Simulation *35*
Figure 15: SRL Simulation *36*
Figure 16: Register File Simulation *36*
Figure 17: Instruction Memory Simulation *37*
Figure 18: MUX 8x1 Simulation *38*
Figure 19: Signed Extension Simulation *38*
Figure 20: Unsigned Extension Simulation *38*
Figure 21: Adder Simulation *39*
Figure 22: Decode and Multiplexing R-Type Simulation *39*
Figure 23: Decode and Multiplexing I-Type Simulation *40*
Figure 24: Data Memory Simulation *40*
Figure 25: NOR Simulation *41*
Figure 26: Instruction Memory *42*
Figure 27: First Instruction Simulation *43*
Figure 28: Second Instruction Simulation *44*
Figure 29: Third Instruction Simulation *45*
Figure 30: Forth Instruction Simulation *47*
Figure 31: Fifth Instruction Simulation *48*
Figure 32: Sixth Instruction Simulation *49*
Figure 33: Seventh Instruction Simulation *51*
Figure 34: Eighth Instruction Simulation *52*
Figure 35: Ninth Instruction Simulation *53*
Figure 36: Tenth Instruction Simulation *54*

List of tables

Table 1: R_Type Instruction Format *4*
Table 2: I_Type Instruction Format *4*
Table 3: J_Type Instruction Format *5*
Table 4: Instruction's Encoding Table *7*
Table 5: Decode and Multiplixing Table for R-Type and I-type *15*
Table 6: Decode and Multiplixing Table for J-Type *15*
Table 7: Finite State Machine Table *24*
Table 8: Main Control Signals *25*
Table 9: Main Control Truth Table *27*
Table 10: Main Control Truth Table *28*
Table 11: PC Control Truth Table *30*
Table 12: ALU Control Truth Table *32*
Table 13: Register File Content *42*
Table 14: Data Memory *43*

1. Design and Implementation

In this project, the goal is to design and verify a pipelined RISC processor using Verilog. The development process starts by thoroughly analyzing the Instruction Set Architecture (ISA), identifying the instruction formats, and defining the processor's specifications. This analysis informs the selection of essential functional units and control mechanisms while guiding the construction of an efficient datapath. This report outlines the methodology employed in designing the processor, provides a comprehensive overview of its components, and explains how instructions are executed step-by-step.

1.1. Processor Specifications and Overview

The designed processor is based on the following specifications:

1. **Instruction and Word Size**: The instruction size and word size are fixed at 16 bits.
2. **Registers**: The processor includes eight general-purpose integer registers, each 16 bits wide.
3. **Program Counter (PC)**: A 16-bit register that tracks the address of the next instruction to execute.
4. **Return Register (RR)**: A dedicated 16-bit register used to store the return address during function calls.
5. **Memory Architecture**: The processor features two separate physical memories: one exclusively for instructions and the other for data storage.
6. **Instruction Types**: The ISA supports three primary instruction formats:
    - **R-Type (Register Type)**
    - **I-Type (Immediate Type)**
    - **J-Type (Jump Type)**
7. **Memory Addressing**: The memory is word-addressable, allowing efficient access to aligned data.

8. **ALU Zero Signal**: The Arithmetic Logic Unit (ALU) generates a zero signal to indicate when the result of the last operation is zero.
9. **Performance Registers**: To monitor execution metrics, the processor incorporates several performance registers that track:
    - Total number of executed instructions
    - Total number of load instructions
    - Total number of store instructions
    - Total number of ALU instructions
    - Total number of control instructions
    - Total number of clock cycles

These specifications significantly shaped the design choices made during the development process, leading to the decision to implement a Multi-Cycle Processor.

In a multi-cycle implementation, each instruction is divided into five primary stages:

1. **Instruction Fetch**
2. **Instruction Decode**
3. **Execution**
4. **Memory Access**
5. **Write Back**

Each stage is executed in a single clock cycle, with the clock cycle duration approximately reduced to one-fifth of that in a single-cycle implementation. Unlike single-cycle designs, where all instructions complete in one cycle, the multi-cycle approach allows each instruction to utilize a varying number of cycles, typically ranging from 2 to 5, depending on the operation. This approach reduces the overall execution time, thereby improving performance.

The **instruction fetch** and **decode** stages are essential for every instruction, while the subsequent steps (execution, memory access, and write back) vary in implementation depending on the instruction type. The specifics of these stages and their variations will be elaborated upon in the following sections.

1.2. Instruction Format & RTL Operations

1.2.1. Instruction Formats and Interpretations

The ISA in our project defines **three instruction types**, each with a distinct format to handle various operations efficiently. These types are detailed below:

1. R-Type (Register Type)

This format is used for register-based operations and is structured as follows:

Opcode^4	Rd^3	Rs^3	Rt^3	Function^3

Table 1: R_Type Instruction Format

- **4-bit Opcode:** The opcode is set to zero for all R-Type instructions.
- **3-bit Rd:** Destination register.
- **3-bit Rs:** First source register.
- **3-bit Rt:** Second source register.
- **3-bit Function:** Specifies the operation to be performed.

2. I-Type (Immediate Type)

This format is designed for instructions involving immediate values or memory access and follows this structure:

Opcode^4	Rs^3	Rt^3	Signed Imm^6

Table 2: I_Type Instruction Format

- **4-bit Opcode:** Specifies the instruction type.
- **3-bit Rs:** First source register.
- **3-bit Rt:** Destination register.
- **6-bit Immediate:**
    - For logical instructions, the immediate value is **zero-extended**.
    - For other instructions, the immediate value is **sign-extended**.

- In **BEQ** and **BNE** (branch instructions), the branch target is calculated as:

$$\boldsymbol{Branch\ Target\ =\ Current\ PC\ +\ (Sign\_Extended\ Immediate)}$$

3. J-Type (Jump Type)

This format is reserved for jump instructions and is structured as follows:

Opcode[4]	9-bit offset	Function[3]

Table 3: J_Type Instruction Format

- **4-bit Opcode:** The opcode is set to **1** for all J-Type instructions.
- **9-bit Offset:** Used to calculate the jump target address.
- **3-bit Function:** Specifies the specific jump operation.
- **Jump Target Address Calculation:**

$$\boldsymbol{Jump\ Target\ =\ PC[15:9]\ ||\ Offset}$$

These three types collectively enable the execution of **15 distinct instructions**, addressing arithmetic, logical, memory, and control operations. The encoding specifics of each instruction type are provided in the corresponding sections of this report.

1.2.2. Instructions' Encoding

The processor implements a carefully designed subset of its Instruction Set Architecture (ISA), encompassing essential operations across three types of instructions: **R-Type**, **I-Type**, and **J-Type**. Each instruction is encoded using specific opcode and function values, ensuring precise operation execution as per its functionality. The table below outlines the instructions supported in this implementation, detailing their type, opcode, function values, and the corresponding Register Transfer Notation (RTN).

Group	No.	Inst.	Format	Meaning	Opcode No.	Opcode Value	Function Value
1	1	AND	R-Type	Reg(Rd) = Reg(Rs) & Reg(Rt)	0	0000	000
	2	ADD	R-Type	Reg(Rd) = Reg(Rs) + Reg(Rt)	0	0000	001
	3	SUB	R-Type	Reg(Rd) = Reg(Rs) - Reg(Rt)	0	0000	010
	4	SLL	R-Type	Reg(Rd) = Reg(Rs) << Reg(Rt)	0	0000	011
	5	SRL	R-Type	Reg(Rd) = Reg(Rs) >> Reg(Rt)	0	0000	100
2	6	ANDI	I-Type	Reg(Rt) = Reg(Rs) & Imm	2	0010	NA
	7	ADDI	I-Type	Reg(Rt) = Reg(Rs) + Imm	3	0011	NA
3	8	LW	I-Type	Reg(Rt) = Mem(Reg(Rs) + Imm)	4	0100	NA
4	9	SW	I-Type	Mem(Reg(Rs) + Imm) = Reg(Rt)	5	0101	NA
5	10	BEQ	I-Type	if (Reg(Rs) == Reg(Rt)) Next PC = Branch Target else PC = PC + 1	6	0110	NA
	11	BNE	I-Type	if (Reg(Rs) != Reg(Rt)) Next PC = Branch Target else PC = PC + 1	7	0111	NA
	12	FOR	I-Type	• Rs stores the loop target address, i.e., the address of the first instruction in the loop block • Rt stores the initial number of the loop iterations, i.e., the initial value of the loop counter • The Rt register is decremented at the end of each iteration. The loop exits when the content of the Rt register becomes zero	8	1000	NA

				• The immediate field is ignored in this instruction			
6	13	JMP	J-Type	Next PC = Jump Target	1	0001	000
7	14	CALL	J-Type	Next PC = Jump Target PC + 1 is saved on the RR	1	0001	001
8	15	RET	J-Type	Next PC = Value of the RR The 9-bit field is ignored in this inst.	1	0001	010

Table 4: Instruction's Encoding Table

1.2.3. Instruction RTL Micro Operations

In this section, we define the Register Transfer Language (RTL) micro-operations for the implemented instructions. The RTL represents the sequence of operations that are executed by the processor during the execution of each instruction. These operations detail how data is moved between registers, the ALU, and memory, and they are fundamental in describing the low-level behavior of the processor. Below, we provide the RTL micro-operations for each instruction type, classified according to the operation type. Using a multi-cycle implementation, each instruction goes through **5 main steps**:

$$IF \rightarrow ID \rightarrow EX \rightarrow M \rightarrow WB$$

The following RTL operations are provided to detail the actions that occur at each stage. According to the specifications, the word size is 2 bytes, memory is byte-addressable, and the program counter (PC) is 16 bits. The instructions have been categorized into groups, as shown in **Table 1**, with each group sharing similar micro-operations. For each group, the required stages are outlined, followed by the corresponding micro-operations in sequence. If a number is not specified next to a micro-operation, it indicates that these operations can be executed simultaneously with the previous micro-operation during the same clock cycle. The micro-operations are as follows:

Group 1: R-Type: $IF \rightarrow ID \rightarrow EX \rightarrow WB$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Fetch Operands:** $\text{data1} \leftarrow \text{Reg(Rs)},\ \text{data2} \leftarrow \text{Reg(Rt)}$
3. **Execute:** $ALU\_result \leftarrow ALU\_function(data1, data2)$
4. **Write ALU:** $Reg(\text{Rd}) \leftarrow ALU\_result$
5. **Next PC:** $\text{PC} \leftarrow \text{PC} + 1$

Group 2: I-Type (Arithmetic): $IF \rightarrow ID \rightarrow EX \rightarrow WB$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Fetch Operands:** $\text{data1} \leftarrow \text{Reg(Rs)},\ \text{data2} \leftarrow \text{Extended}_{\text{Imm}}$ *
3. **Execute:** $ALU\_result \leftarrow ALU\_opcode(data1, data2)$
4. **Write ALU:** $Reg(\text{Rt}) \leftarrow ALU\_result$
5. **Next PC:** $\text{PC} \leftarrow \text{PC} + 1$

* *Sign Extended at ADDI , Unsigned Extended at ANDI*

Group 3: I-Type (Load Word): $IF \rightarrow ID \rightarrow EX \rightarrow M \rightarrow WB$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Fetch Base Register:** $\text{base} \leftarrow \text{Reg(Rs)}$
3. **Execute (Calculate address):** $Address \leftarrow base + Sign\_Extended(Imm)$
4. **Read (Memory):** $data \leftarrow Mem[Address]$
5. **Write (Register):** $Reg(Rt) \leftarrow data$
6. **Next PC:** $\text{PC} \leftarrow \text{PC} + 1$

Group 4: I-Type (Store Word): $IF \rightarrow ID \rightarrow EX \rightarrow M$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Fetch Base Register:** $\text{base} \leftarrow \text{Reg(Rs)}, data \leftarrow Reg(Rt)$
3. **Execute (Calculate address):** $Address \leftarrow base + Sign\_Extended(Imm)$
4. **Read (Memory):** $Mem[Address] \leftarrow data$
5. **Next PC:** $\text{PC} \leftarrow \text{PC} + 1$

Group 5: I-Type (Branch): $IF \rightarrow ID \rightarrow EX$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Fetch Registers:** $\text{data1} \leftarrow \text{Reg(Rs)}, data2 \leftarrow Reg(Rt)$
3. **Execute (Calculate address):** $Zero\_flag \leftarrow data1 - data2$

   $if\left(Zero_{flag}\right)$:

   $PC \leftarrow (PC + 1) + Sign\_Extended(Imm)$

$else$:

$PC \leftarrow PC + 1$

Group 6: I-Type (For): $IF \rightarrow ID \rightarrow EX \rightarrow WB$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Fetch Registers:** $Inst_{Address} \leftarrow \text{Reg(Rs)}, Iterations \leftarrow Reg(Rt)$
3. **Execute (Calculate address):** $Zero\_flag \leftarrow Iterations$

   $if(!\,Rt)$:

   $PC \leftarrow Inst_{Address}$

   $Iterations \leftarrow Iterations - 1$

   $else$:

   $PC \leftarrow PC + 1$
4. **Write (Iterations Register):** $Reg(Rt) \leftarrow Iterations$

Group 7: J-Type (Jump): $IF \rightarrow ID$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Target PC Address:** $Target\ Address \leftarrow \text{PC}[15:9]\ ||\ 9 - offset$

   $PC \leftarrow Target\ Address$

Group 8: J-Type (Call): $IF \rightarrow ID$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Target PC Address:** $Target\ Address \leftarrow \text{PC}[15:9]\ ||\ 9 - offset$
3. **Save Next Instruction Address:** $RR \leftarrow PC + 1$

   $PC \leftarrow Target\ Address$

Group 9: J-Type (RET): $IF \rightarrow ID$

1. **Fetch Instruction:** $IR \leftarrow Mem[PC]$
2. **Target PC Address:** $PC \leftarrow RR$

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. PC

The Program Counter (PC) is a specialized register essential for program execution. It stores the address of the current instruction that needs to be executed. According to the processor's specifications, the PC is 16 bits in size, allowing it to access up to 2^{16} locations. Since the memory is word-addressable, the PC can access up to 2^{16} words, which equals 128 kilobytes (128KB). To ensure proper operation within the data path, the PC is clocked and includes an enable signal (PCwrite), which allows for writing to the PC at specific times.

2. IR

The Instruction Register (IR) is another specialized register responsible for holding the current instruction. As the instruction size is 16 bits, the IR must also be 16 bits wide. The IR plays a key role in tracking the current instruction and is essential during the decode stage, where various operands and fields are extracted directly from it. Like other registers, the IR is clocked and includes an enable signal (IRwrite), which controls when data can be written to it.

3. RR

The 16-bit Return Register (RR) is a specialized register used to store the return address during function calls. This register is crucial for managing control flow in the processor, as it holds the address to which the processor should return after completing a function call. The RR is particularly important in handling function calls and returns, ensuring that the processor can correctly resume execution from the point where the function was invoked. It is designed to be 16 bits wide, aligning with the processor's word size and supporting the appropriate range of addresses for function returns. Like other registers, the RR is clocked and includes an enable signal (RRwrite), which controls when data can be written to it.

4. Instruction Memory

Both the Program Counter (PC) and the Instruction Register (IR) are integral components involved in fetching, storing, and executing instructions. As per the processor specifications, the data memory and instruction memory are separate, each serving distinct functions within the data path. The instruction memory is specifically responsible for storing the program's instructions. It operates as a read-only memory for instruction fetches, meaning no writes are made to it during execution. The instruction memory accepts a 16-bit address input and outputs a 16-bit instruction. Given that the memory is word-addressable and the instruction length is 16 bits, each instruction occupies a single memory cell. This organization ensures efficient instruction retrieval and execution during the processing cycle.

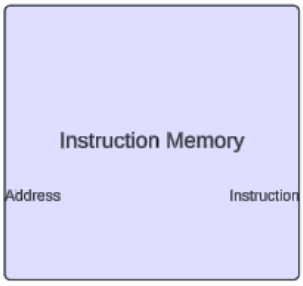

Figure 1: Instruction Memory

5. Data Memory

In the Memory Access stage, reading data from memory is essential. The data memory in this processor is designed to store and retrieve data during load and store instructions. It is word-addressable and provides four inputs: a 16-bit address, a 16-bit data input, a MemRd signal for read operations, and a MemWrite signal for write operations. It has one output: a 16-bit data output. During load instructions, the MemRd signal is activated, and the data at the specified address is output via data_out. For store instructions, the MemWrite signal is enabled, allowing the input data to be stored at the specified address. A clock signal is required to synchronize the write operations.

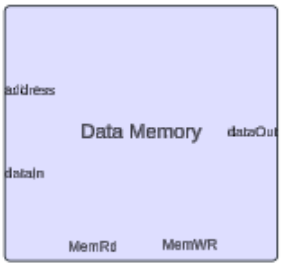

Figure 2: Data Memory

6. Register File

In the micro-operations, alongside memory and special-purpose registers, the instruction also utilizes general-purpose registers, as outlined in the Processor Specifications section. Our design includes eight 16-bit wide registers, labeled R0 to R7. Notably, R0 is hardwired to zero, and any attempt to modify its value is disregarded. The register file, as shown in the accompanying figure, is equipped with multiple ports, including a clock input to synchronize the data path. It features two read ports and one write port, along with three address inputs (addr_read1, addr_read2, addr_write), each 3-bits wide to accommodate the eight registers. The corresponding outputs (bus_read1, bus_read2) provide the data of the registers at the specified addresses. It is important to note that reading from the register file is a combinational process and does not require clocking. For writing, the 16-bit bus (bus_write) carries the data to be written to the register at the address specified by addr_write. The control signal (RegWr) enables or disables the writing process. Writing is clocked and must be synchronized to occur only on the edge of the clock when (RegWr) is asserted.

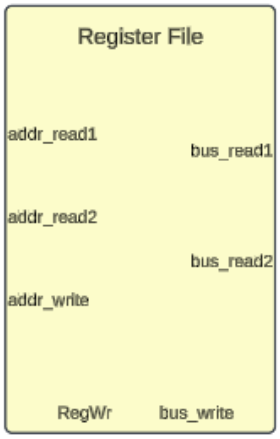

Figure 3: Register File

7. ALU

The ALU is designed to perform five primary operations: logical shift left, logical shift right, logical AND, addition, and subtraction, as specified by the instruction set and micro-operations. It also includes a no_op operation for cases where its output is not required or when a delay is needed. The ALU has two 16-bit inputs, A and B, for the operands, and a 16-bit output, result. The specific operation is determined by the ALU_opcode input. Additionally, the ALU generates a zero-flag signal, which is set when the result of the last operation is zero. This signal plays a crucial role in control and decision-making processes within the processor.

Figure 4: ALU

8. Decode and Multiplexing Unit

As observed in the instruction formats and decoding process, the interpretation of specific bits varies depending on the instruction type and its purpose. For instance, in R-type and I-type instructions, the source register (Rs) occupies different bit positions: in R-type, it is located in bits [8:6], while in I-type, it occupies bits [11:9]. This creates multiple cases for decoding the instruction format. To streamline this, we developed the Decode and Multiplexing Unit, which simplifies the decoding process by directly interpreting the instruction. This approach reduces the need for additional muxes, saving time and minimizing complexity.

The Decode and Multiplexing Unit is utilized during the decode stage, where the opcode is identified, operands are fetched, and the destination register is determined based on the provided function. The table below demonstrates how the opcode is used to identify the source and destination operand addresses for ALU-based instructions.

Instruction	Dest. Reg. (addr_write)	Op.1 (addr_read1)	Op.2 (addr_read2)	Comments
R - type	Rd = IR [11:9]	Rs = IR [8:6]	Rt = IR [5:3]	
ALU_I	Rt = IR [8:6]	Rs = IR [11:9]	Imm [5:0]	
LW	Rt = IR [8:6]	Rs = IR [11:9]	Imm [5:0]	
SW	-	Rs = IR [11:9]	Imm [5:0]	Data in : Rt = IR [8:6]

BEQ, BNE	-	Rs = IR [11:9]	Rt = IR [8:6]	Imm = IR[5:0]
FOR	Rt = IR [8:6]	Rs = IR [11:9]	Rt = IR [8:6]	

Table 5: 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 [11:3] for calculating jump address
Call	Dst register = RR offset = IR [11:3] for calculating function address
Ret	Read Reg = RR from Reg file Addr_read1 = RR

Table 6: Decode and Multiplexing Table for J-Type

As demonstrated in the previous two tables, there are different options for addr_read1, addr_read2, and addr_write in the register file. Instead of using three multiplexers, we introduced a new component called the decode_and_multiplexing_unit to decode the instruction. This unit takes the instruction as input and generates the opcode, func, addr_read1, addr_read2, and addr_write for the register file. Specifically:

- **Address_read1**: The address of the first register to read from the register file.

- **Address_read2**: The address of the second register to read from the register file.

- **Address_write**: The address of the destination register to write to the register file.

Additionally, the unit produces I_type_imm and J_type_imm. Since the size of the immediate varies depending on the instruction type, it is classified into two categories:

- **I_type_imm**: The immediate in I-type instructions (6 bits).

- **J_type_imm**: The immediate (offset) used in Jump and Call instructions (9 bits).

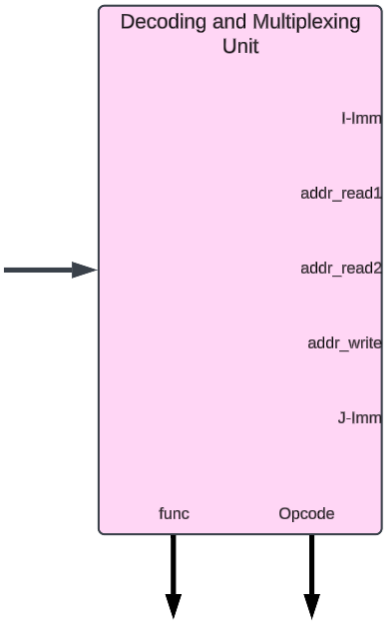

Figure 5: Decoding and Multiplexing Unit

9. Extender Unit

As discussed in the instruction format and the decoding process, the immediate values in the instructions have varying sizes. To address this, extenders are used to expand each immediate from 6 bits to 16 bits. The extension can either be signed or unsigned, with the type of extension determined by a control signal.

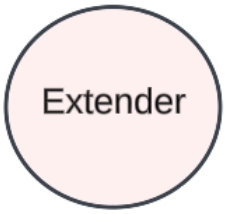

Figure 6: Extender Unit

10. Mux Units

We can observe that certain components have various values written based on the instruction if we have a better knowledge of the micro-operations of the instructions. These components are chosen via a mux, with the control units' control signals serving as the selection lines. A portion of this has been resolved in the multiplexing and decoding unit, although some still require extra muxes. These can be summed up as follows as having **three of 2×1 mux** and **one of 8×1 mux**:

1. 8×1 PC mux

➔ bit 0 represent the original PC which $PC = PC + 1$

➔ bit 1 represent the conditional branches PC which $PC = PC + 1 + Imm$

➔ bit 2 represent the conditional branches PC which $PC = Inst.Address$

➔ bit 3 represent the jump instruction and get target address then put it in

$PC = target\ address = [\ PC[15:9]\ ||\ 9_{bit} offset\ ]$

➔ bit 4 represent the RR which is return PC which PC=PC+1 back to the instruction that followed the instruction that we call label far of it

2. 2×1 muxes:

2.1 writing back to destination Reg

➔ bit 0 represent the value from ALU

➔ bit 1 represent the value from memory that comes from store instruction

2.2 first source to ALU

➔ bit 0 represent the value that comes from the bus_B

➔ bit 1 represent the value that it is immediate not a register source

2.3 second source to ALU

➔ bit 0 represent the value that comes from the bus_A

➔ bit 1 represent the value of 1 in order to subtract used in **for** instruction

11. NOR Unit

the **NOR gate** comparator was specifically utilized to address the issue of pre-decrementing the rt register in the FOR instruction. The goal was to determine whether the rt value had reached zero after iterations, ensuring the system executes the loop the correct number of times.

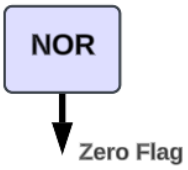

Figure 7: NOR Unit

12. Buffers Between Stages

In our multi-cycle implementation, each processing stage operates independently. To ensure data availability for subsequent clock cycles, we employ storage elements (registers) synchronized with the clock signal. Beyond the Instruction Register (IR), Program Counter (PC), and Return Register (RR), we utilize five additional buffers:

- **Register A:** Stores the output of bus_read1 from the decode stage, serving as the first operand for the ALU in the execution stage.

- **Register B:** Stores the output of bus_read2 from the decode stage, serving as the second operand for the ALU in the execution stage.

- **Register Immediate:** Stores the extended immediate value from the decode stage, used as the ALU's second operand when necessary.

- **Register ALU_result:** Stores the output of the ALU from the execution stage, utilized in the memory or write-back stages.

- **Register Inst. Address:** Stores the address of the first instruction within a loop block.

1.4. 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.

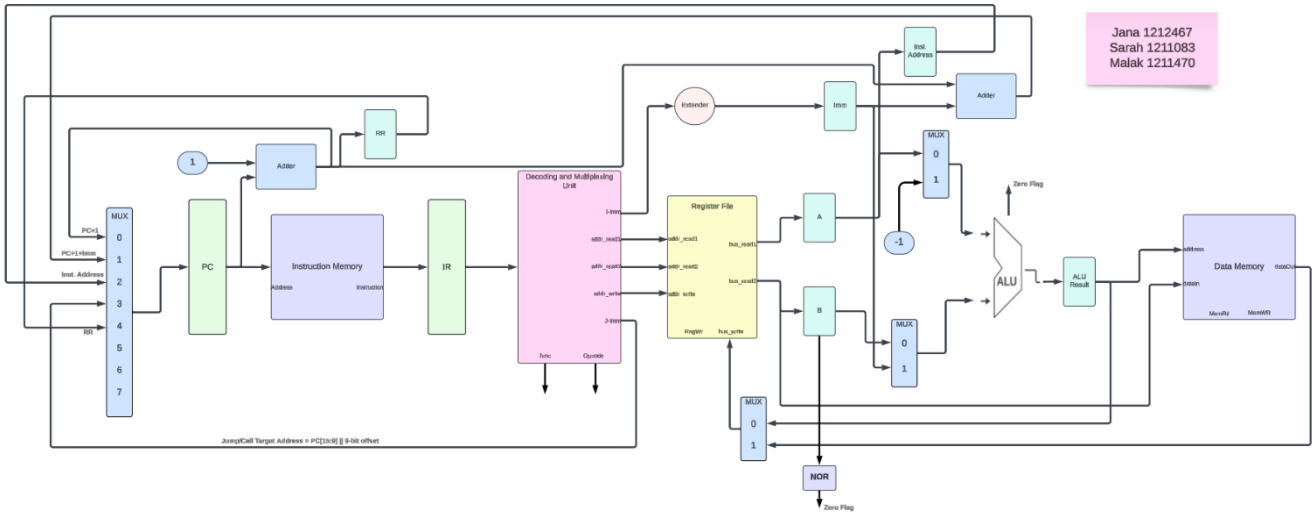

Figure 8: Datapath

Execution commences at the Program Counter (PC) during the instruction fetch stage. The PC stores the address of the next instruction. A (+1) adder increments the PC to determine the address of the subsequent instruction in the sequential flow. A separate adder calculates the branch target address when a branch instruction is encountered. In the decode stage, the fetched instruction is loaded into the Instruction Register (IR) and then processed by the decode unit (combinational logic). This unit extracts the necessary fields from the instruction. Finally, the instruction proceeds to the appropriate stage: Execute, Memory, or Write Back, depending on its opcode.

In the Execute stage, the ALU performs operations based on the instruction type. For R-type, the inputs are Register A and Register B. For I-type arithmetic, the inputs are Register A and the immediate value. For I-type branch instructions (BEQ, BNE, FOR), the ALU performs a subtraction operation: Rs - Rt for BEQ/BNE and Rt - 1 for FOR. ALU computes the result and sets the zero flag accordingly. Memory access is restricted to Load and Store instructions. The address for memory access is derived from the ALU result. Store instructions write data from the Rt register to memory, while Load instructions only read data from memory. The Write Back stage writes the result back to registers: Rt for Load, I-type arithmetic, and FOR instructions; Rd for R-type instructions.

To implement the multi-cycle design effectively, each multiplexer in the data path requires selection lines determined by the control units. As such, three dedicated control units—**PC Control**, **Main Control**, and **ALU Control**—are integrated into the data path. These units generate the necessary control signals, enabling the execution of instructions in a structured manner. The connections and signal flow are depicted in **Figure 8**, where the execution and data flow at each stage depend entirely on the values of these signals.

1. **PC Control Unit**
    - The **PC Control Unit** is responsible for managing the signals that govern the Program Counter (PC) multiplexer (8-to-1 mux).
    - It determines the mode of PC operation: Original, FOR loop, Return Register (RR), conditional branch, Call or jump.
    - Additionally, it controls the **PC Register**, enabling write operations to update the current instruction address.
    - Inputs to the PC Control Unit include the **Opcode** and the **Zero Flag**, which dictate its output signals.
2. **ALU Control Unit**
    - The **ALU Control Unit** manages the signals sent to the Arithmetic Logic Unit (ALU), dictating the operation the ALU performs (e.g., ADD, SUB, SHIFT).
    - It takes two key inputs: the **Opcode** and the **State**, ensuring the ALU executes the correct operation based on the current stage of execution.
3. **Main Control Unit**
    - The **Main Control Unit** oversees the signals sent to various units, enabling precise execution of instructions. Key responsibilities include:
    1. **Instruction Register (IR)**: Controls the enable signal for the IR, ensuring it updates with a new instruction at the appropriate time.
    2. **Return Register (RR)**: Controls the enable signal for the RR, ensuring it updates with a new address at the appropriate time.

3. **Extender**: Sets the extension mode for immediate values, determining whether the extension is **signed** or **unsigned**.
4. **Register File Multiplexer (4-to-1)**: Manages the selection line to determine which data is written back to the register file.
5. **ALU Source Inputs**: Controls the selection lines for the ALU input multiplexers (2-to-1), determining the source of each ALU operand.
6. **Data Memory**: Controls the operation mode of the memory unit, specifying whether it performs a **write**, **read**, or remains idle, depending on the current stage.

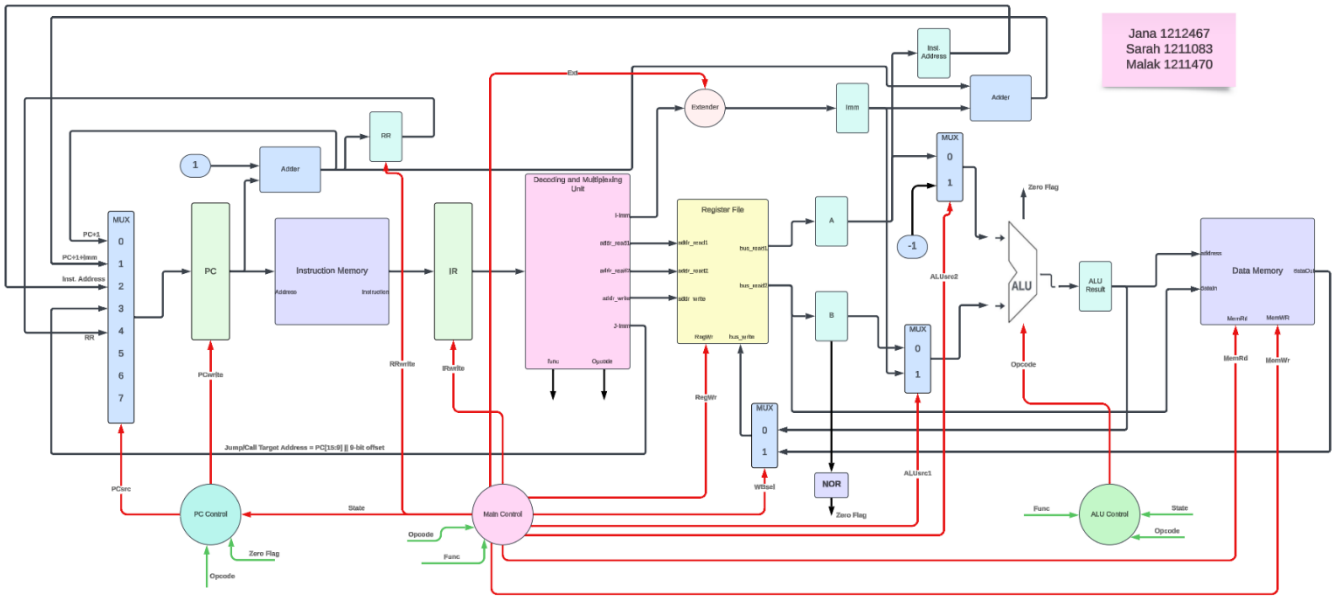

Figure 9: Datapath with controls

1.5. Designing the Control Unit

In a multi-cycle implementation, each instruction requires a varying number of clock cycles depending on the stages it must traverse. This variation means that the subsequent stage of an instruction is determined by its current state, function, and opcode. To manage this, the multi-cycle control unit is designed as a finite state machine (Mealy state machine). Each state is uniquely defined by the control signals generated by the control unit at that moment, which dictate the operations performed during that stage.

The state diagram for the designed state machine is illustrated in the below figure, where each state specifies the control signals activated during that stage. **Any control signals not explicitly shown are assumed to default to 0, with the ALU_Opcode defaulting to no_op**. Additionally, transition logic between states is determined by the current state, the instruction's opcode, function, and specific condition flags (e.g., Zero flag for branch instructions).

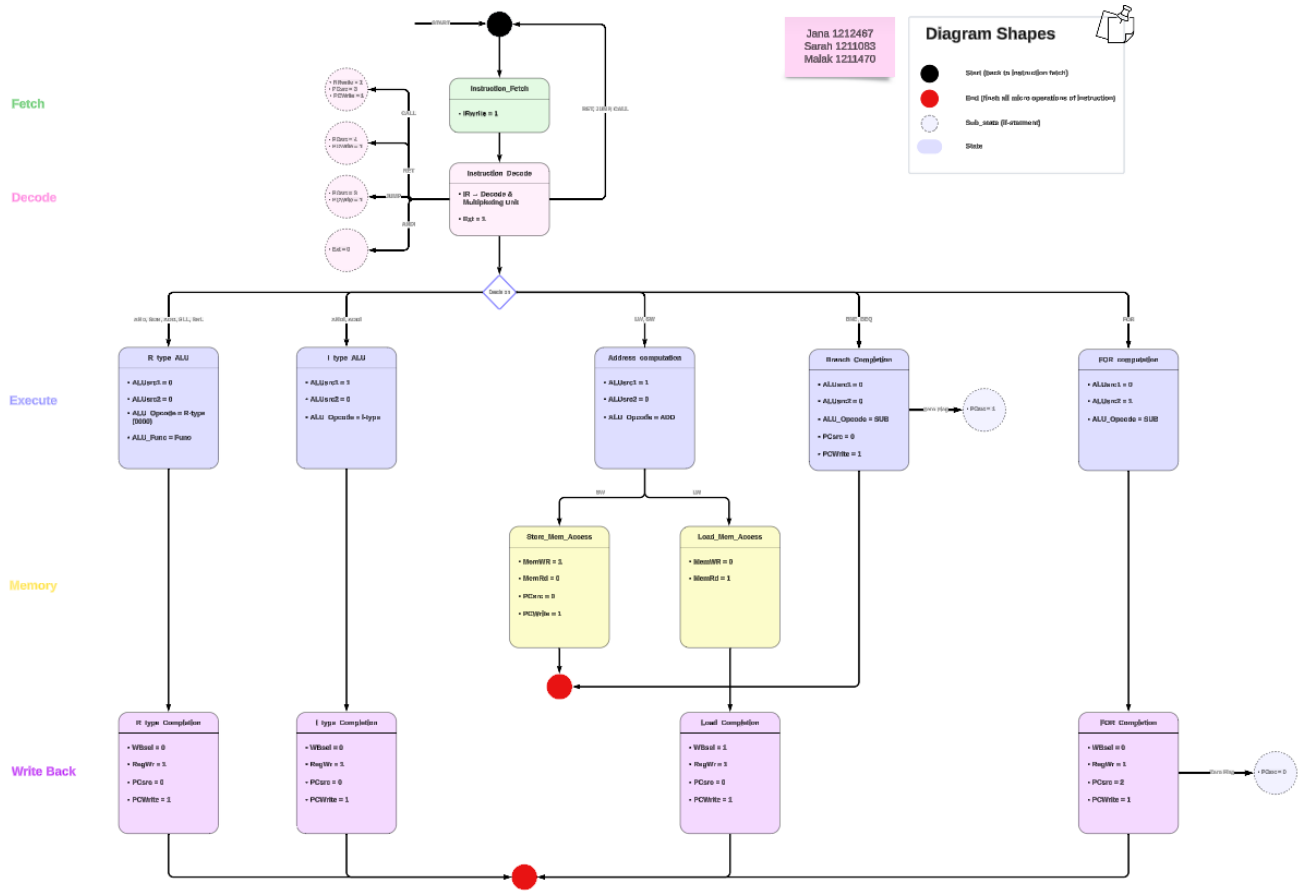

Figure 10: Finite State Machine

To better comprehend the state diagram, consider the **ADDI** instruction as an example. This instruction begins at the **Fetch** state, where the instruction is retrieved from memory, and then transitions to the **Decode** state. These two initial states are shared by all instructions in the pipeline. After decoding, the **ADDI** instruction proceeds to the **I_type_ALU** state. In this state, the ALUsrc control signal is set to 1, and the ALU_Opcode is configured for the I-type operation, which in this case is an **ADD**. Following execution, the instruction transitions to the **I_type_Completion** state (WriteBack stage), where the ALU result is written to a register, and the PC is updated to point to the next instruction. This sequence spans **four states or clock cycles**, meaning that the instruction requires four cycles to execute.

On the other hand, a **BEQ** (Branch Equal) instruction follows a different sequence. It begins in the **Fetch** state and moves to the **Decode** state, like all instructions. After decoding, it transitions to the **Branch_Completion** state. Here, the ALU is used to compare the values from the source registers (Rs and Rt), and the Zero flag is evaluated. If the flag indicates equality, the branch target address is calculated as the sum of the current PC and the sign-extended immediate value. This branch operation requires **three clock cycles** to complete, as the instruction does not include a WriteBack stage.

To implement these operations efficiently within the data path, the control signals are divided into three categories: **PC Control**, **ALU Control**, and **Main Control**. The **Main Control** unit, designed as a finite state machine, determines the next state based on the current state and opcode. This next state serves as input to the **ALU Control** and **PC Control** units, which generate the corresponding signals required for their respective stages.

The following sections further analyze the derivation of these signals and their roles within the multi-cycle implementation. This modular approach ensures that the design is scalable, simplifies the debugging process, and guarantees seamless integration into the system.

State	State Number	Abbreviation
Instruction_Fetch	0	IF
Instruction_Decode	1	ID
R_type_ALU	2	R_ALU
R_type_Completion	3	RC
I_type_ALU	4	I_ALU
I_type_Completion	5	IC
Address_computation	6	AC
Load_Mem_Access	7	L_Mem
Store_Mem_Access	8	S_Mem
Load_Completion	9	LC
Branch_Completion	10	BC
FOR_computation	11	FC
FOR_Completion	12	FCC

Table 7: Finite State Machine Table

1.5.1. 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
IRwrite	IR value does not change the content	The output of instruction memory is written on IR	No effect
RRwrite	RR value does not change the content	The PC address is written on RR	No effect
Ext	I_type_imm is unsigned extended.	I_type_imm is signed extended.	No effect
WBsel	Data_Written_To_Regfile ALU data	Data_Written_To_Regfile Memory data	No effect
ALU_src 1	ALUsrc = Reg A	ALUsrc = 1	No effect
ALU_src 2	ALUsrc = Reg B	ALUsrc = Immediate	No effect
MemRd	No effect	Data is put on data_out	No effect
MemWr	No effect	Data in is written into memory	No effect
RegWr	No effect	Write on register file	No effect

Table 8: Main Control Signals

The **Main Control Unit** is implemented as a **finite state machine (FSM)**. It takes the **clock signal**, **opcode**, and **function** as inputs. Based on these inputs and the current state, it determines the values of the 9 main control signals as well as the next state.

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

Inputs			Outputs					
Current state	**Opcode**	**Func**	**Next State**	**IR write**	**RR write**	**Ext**	**ALU_ src1**	**ALU_ src2**
IF	**X**	**X**	ID	1	0	X	X	X
ID	**0000**	**AND 000 ADD 001 SUB 010 SLL 011 SRL 100**	R _ALU	0	0	X	X	X
ID	**ADDI**	**X**	I_ ALU	0	0	1	X	X
ID	**ANDI**	**X**	I_ALU	0	0	0	X	X
ID	**LW**	**X**	AC	0	0	1	X	X
ID	**SW**	**X**	AC	0	0	1	X	X
ID	**BEQ BNE**	**X**	BC	0	0	1	X	X
ID	**FOR**	**X**	FC	0	0	X	X	X
ID	**0001**	**JUMP 000**	IF	0	0	X	X	X
ID	**0001**	**CALL 001**	IF	0	1	X	X	X
ID	**0001**	**RET 010**	IF	0	0	X	X	X
R_ALU	**0000**	**AND 000 ADD 001 SUB 010 SLL 011 SRL 100**	RC	0	0	X	0	0
RC	**0000**	**AND 000 ADD 001 SUB 010 SLL 011 SRL 100**	IF	0	0	X	X	X
I_ALU	**I_type**	**X**	IC	0	0	1	1	0
IC	**I_type**	**X**	IF	0	0	X	X	X
BC	**branch**	**X**	IF	0	0	X	0	0
AC	**LW**	**X**	L_Me m	0	0	X	1	0
AC	**SW**	**X**	S_Me m	0	0	X	1	0

L_Mem	**X**	**X**	LC	0	0	X	X	X
LC	**LW**	**X**	IF	0	0	X	X	X
S_Mem	**SW**	**X**	IF	0	0	X	X	X
FC	**FOR**	**X**	FCC	0	0	X	0	1
FCC	**FOR**	**X**	IF	0	0	X	X	X

Table 9: Main Control Truth Table

Inputs			**Outputs**					
Current state	**Opcode**	**Func**	**Next State**	**IR write**	**MemRd**	**MemWr**	**WBsel**	**RegWr**
IF	**X**	**X**	ID	1	0	0	X	0
ID	**0000**	**AND 000 ADD 001 SUB 010 SLL 011 SRL 100**	R _ALU	0	0	0	X	0
ID	**ADDI**	**X**	I_ ALU	0	0	0	X	0
ID	**ANDI**	**X**	I_ALU	0	0	0	X	0
ID	**LW**	**X**	AC	0	0	0	X	0
ID	**SW**	**X**	AC	0	0	0	X	0
ID	**BEQ BNE**	**X**	BC	0	0	0	X	0
ID	**FOR**	**X**	FC	0	0	0	X	0
ID	**0001**	**JUMP 000**	IF	0	0	0	X	0
ID	**0001**	**CALL 001**	IF	0	0	0	X	0
ID	**0001**	**RET 010**	IF	0	0	0	X	0
R_ALU	**0000**	**AND 000 ADD 001 SUB 010 SLL 011 SRL 100**	RC	0	0	0	X	0
RC	**0000**	**AND 000 ADD 001**	IF	0	0	0	0	1

		SUB 010 SLL 011 SRL 100						
I_ALU	**I_type**	**X**	IC	0	0	0	X	0
IC	**I_type**	**X**	IF	0	0	0	0	1
BC	**branch**	**X**	IF	0	0	0	X	0
AC	**LW**	**X**	L_Mem	0	0	0	X	0
AC	**SW**	**X**	S_Mem	0	0	0	X	0
L_Mem	**X**	**X**	LC	0	1	0	X	0
LC	**LW**	**X**	IF	0	0	0	1	1
S_Mem	**SW**	**X**	IF	0	0	1	X	0
FC	**FOR**	**X**	FCC	0	0	0	X	0
FCC	**FOR**	**X**	IF	0	0	0	0	1

Table 10: Main Control Truth Table

The logical expressions that are derived for each signal:

IRwrite = IF

RRwrite = ID.CALL

Ext = ADDI .BEQ.BNE.LW.SW

ALU_src1 = I_ALU + AC

ALU_src2 = FC

RegWr = RC + IC + FCC + LC

WB_sel [0] = RC + IC + FCC

WB_sel [1] = LC

MemRd = L_Mem

MemWr = S_Mem

1.5.2. 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 **inputs**:

- **Opcode**: The operation code of the current instruction.

- **ALU Zero Flag**: A flag indicating whether the result of the last ALU operation was zero.

- **State**: The current stage of instruction execution (e.g., fetch, decode, execute, writeback).

Based on these inputs, 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 at the final stage of an instruction's execution.
- **Disabled (0)**: The PC value remains unchanged.

- **PCsrc (3 bits)**: Selects the source address to be written to the PC:

  0: Next PC = PC + 1 (sequential execution)

  1: Next PC = PC + 1 + Immediate (relative addressing)

  2: Next PC = Instruction Address (used for FOR instructions)

  3: Next PC = Jump or Call Target Address (for unconditional jumps and subroutine calls)

  4: Next PC = Return Register (RR) (for subroutine returns)

Inputs				**Outputs**	
State	**Opcode**	**Func**	**Zero**	**PCwrite**	**PCsrc**
ID	**Jump**	000	**X**	1	011
ID	**CALL**	**001**	**X**	1	011
ID	**RET**	010	**X**	1	100
BC	**BEQ**	**X**	**1**	1	001

BC	**BEQ**	**X**	**0**	1	000
BC	**BNE**	**X**	**1**	1	000
BC	**BNE**	**X**	**0**	1	001
RC	**X**	**X**	**X**	1	000
IC	**X**	**X**	**X**	1	000
LC	**LW**	**X**	**X**	1	000
S_Mem	**SW**	**X**	**X**	1	000
FCC	**FOR**	**X**	**1**	1	000
FCC	**FOR**	**X**	**0**	1	010

Table 11: PC Control Truth Table

Logical expressions that were derived are as follows:

PCwrite = (ID.(JMP+RET_CALL)) + BC + RC + IC + LC + S_Mem + FCC

PCsrc:

- **PCsrc[0]** = (State = ID AND (Opcode = Jump OR Opcode = CALL)) OR (State = BC AND Opcode = BEQ AND Zero = 1) OR (State = BC AND Opcode = BNE AND Zero = 0) OR (State = FCC AND Opcode = FOR AND Zero = 0)
- **PCsrc[1]** = (State = ID AND (Opcode = Jump OR Opcode = CALL)) OR (State = ID AND Opcode = RET) OR (State = BC AND Opcode = BEQ AND Zero = 1) OR (State = BC AND Opcode = BNE AND Zero = 0) OR (State = FCC AND Opcode = FOR AND Zero = 0)
- **PCsrc[2]** = (State = BC AND ((Opcode = BEQ AND Zero = 1) OR (Opcode = BNE AND Zero = 0))) OR (State = RC) OR (State = IC) OR (State = LC) OR (State = S_Mem)

1.5.3. Designing the ALU Control

The table below summarizes the operations required for each instruction during the execute stage. There are five unique ALU operations, which are represented using 3 bits as follows:

1. **AND_op**: 000
2. **ADD_op**: 001
3. **SUB_op**: 010
4. **Shift_Left_Op**: 011
5. **Shift_Right_Op**: 100
6. **NO_operation**: 101

These ALU operations cover the main functions that our ALU will perform, including logical, arithmetic, and shift operations.

No.	Inst.	Opcode Value	ALU Operation	ALU Opcode (decimal)	ALU Opcode Value	State
1	AND	0000	AND_Op	0	000	R_type_ALU
2	ADD	0000	ADD_Op	1	001	R_type_ALU
3	SUB	0000	SUB_Op	2	010	R_type_ALU
4	SLL	0000	Shift_Left_Op	3	011	R_type_ALU
5	SRL	0000	Shift_right_Op	4	100	R_type_ALU
6	ADDI	0011	ADD_Op	1	001	I_type_ALU
7	ANDI	0010	AND_Op	0	000	I_type_ALU
8	LW	0100	ADD_Op	1	001	Address_Computation
9	SW	0101	ADD_Op	1	001	Address_Computation
10	BEQ	0110	SUB_Op	2	010	Branch_Completion
11	BNE	0111	SUB_Op	2	010	Branch_Completion
12	JMP	0001	-	-	-	-
13	CALL	0001	-	-	-	-

14	**RET**	**0001**	**-**	**-**	**-**	**-**
15	**FOR**	**1000**	**SUB_Op**	**2**	**010**	**FC**

Table 12: ALU Control Truth Table

We aim to derive a Boolean expression for the **ALU_opcode**, considering both the opcode and the current execution state. This can be expressed behaviorally as follows:

```
If (State != R_type_ALU && State != I_type_ALU && State != Address_Computation
&& State != Branch_Completion && State != FC)
   ALU_Opcode = No_Op
else if (State == Address_Computation)
   ALU_Opcode = ADD_Op
else if (State == Branch_Completion)
   ALU_Opcode = SUB_Op
else if (State == FC)
   ALU_Opcode = ADD_Op
else: // R_type_ALU or I_type_ALU
   if (Instr == ADD || Instr == ADDI || Instr == LW || Instr == SW)
      ALU_Opcode = ADD_Op
   else if (Instr == SUB || Instr == BEQ || Instr == BNE|| Instr == FOR)
      ALU_Opcode = SUB_Op
   else if (Instr == AND || Instr == ANDI)
      ALU_Opcode = AND_Op
   else if (Instr == SLL)
      ALU_Opcode = Shift_Left_Op
```

```
else if (Instr == SRL)
    ALU_Opcode = Shift_Right_Op
```

The ALU opcode can be derived through Boolean expressions for each of its bits, taking into account the execution state and opcode of the current instruction, as shown below:

- **ALU_Opcode[0]** = (State == Address_Computation) + (State == FC) + (Instr == ADD) + (Instr == ADDI) + (Instr == LW) + (Instr == SW) + (Instr == SLL) + (Instr == SRL)
- **ALU_Opcode[1]** = (State == Branch_Completion) + (Instr == SUB) + (Instr == BEQ) + (Instr == BNE) + (Instr == FOR)
- **ALU_Opcode[2]** = (Instr == AND) + (Instr == ANDI)

It is crucial to understand that the **ALU_opcode is assigned only during the execution state.** Therefore, for instance, when the ANDI instruction is used, the **opcode will be No_op in the instruction fetch, decode, and write-back stages.**

2. Simulation and Testing

The simulation and testing phase of the project were carried out using Active-HDL, a versatile platform for simulating and debugging digital designs. The process began with simulating individual processor components, such as the arithmetic logic unit (ALU), control unit, and memory modules. By generating waveforms for these components, we verified their functionality and ensured that each unit performed as expected. This step was crucial for identifying and resolving potential issues at the component level, facilitating efficient debugging.

After validating the components, we proceeded to simulate the entire processor. Using a dedicated test processor module, we created a comprehensive program encompassing all instruction types to evaluate the processor's performance. This program was designed to assess the processor's ability to execute instructions in a multi-cycle manner, adhering to the intended design. By observing the processor's behavior and analyzing the waveforms, we confirmed its correctness and verified that the execution cycle count aligned with our expectations.

2.1. 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, Logical Shift Left (SLL), and Logical Shift Right (SRL).

- **ADD**

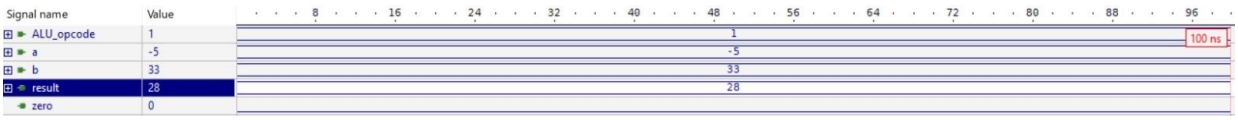

Figure 11: ADD Simulation

In this example, an addition operation is performed. The ALU opcode for addition is **(001)**. The initial value stored in register A is **-5**, and register B holds **33**. After performing the operation, the result of **-5 + 33 = 28** is correctly displayed in the result register. The zero flag remains **0** since the result is not zero.

- **SUB**

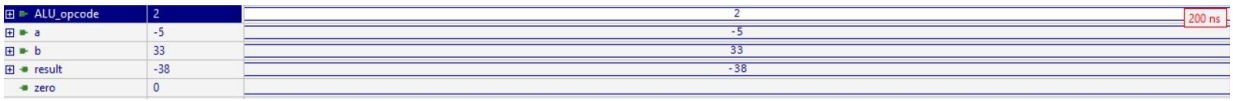

Figure 12: SUB Simulation

In this example, subtraction operation is performed. The ALU opcode for subtraction is **(010)**. The initial value stored in register A is **-5**, and register B holds **33**. After performing the operation, the result of **-5 - 33 = -38** is correctly displayed in the result register. The zero flag remains **0** since the result is not zero.

- **AND**

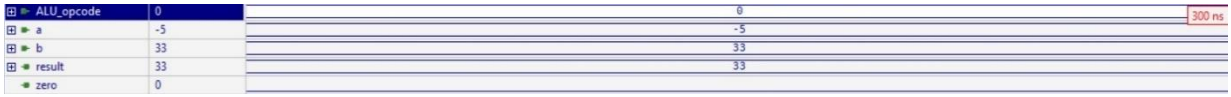

Figure 13: AND Simulation

In this example, an AND operation is performed. The ALU opcode for AND is **(000)**. The initial value stored in register A is **-5** , and register B holds **33**.

– Represent the numbers in 16-bit binary:

- **-5** (using two's complement): 1111 1111 1111 1011
- **33**: 0000 0000 0010 0001

After performing the operation, the result of **-5 AND 33 = 33 (**0000 0000 0010 0001) is correctly displayed in the result register. The zero flag remains **0** since the result is not zero.

- **SLL**

Figure 14: SLL Simulation

In this example, shift lift logical operation is performed. The ALU opcode for SLL is **(011)**. The initial value stored in register A is **-5**, and register B holds **2**. Represent the numbers in 16-bit binary:

- **-5** (using two's complement): 1111 1111 1111 1011
- **2**: 0000 0000 0000 0010

After performing the logical shift left operation, the result of **-5 << 2 = -20 (1111 1111 1110 1100)** is correctly displayed in the result register. The zero flag remains **0** since the result is not zero.

- **SRL**

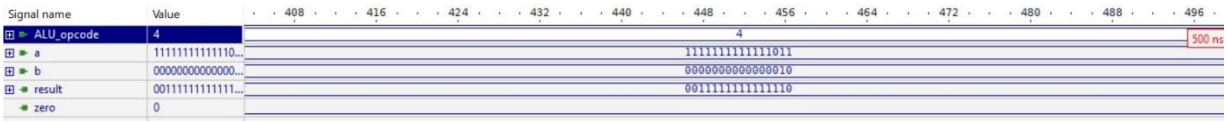

Figure 15: SRL Simulation

In this example, shift right logical operation is performed. The ALU opcode for SRL is **(110)**. The initial value stored in register A is **-5**, and register B holds **2**.

Represent the numbers in 16-bit binary:

- **-5** (using two's complement): 1111 1111 1111 1011
- **2**: 0000 0000 0000 0010

After performing the logical shift right operation, the result of **-5 >> 33 = -2** (1111 1111 1111 1110) is correctly displayed in the result register. The zero flag remains **0** since the result is not zero.

2.1.2. Register File

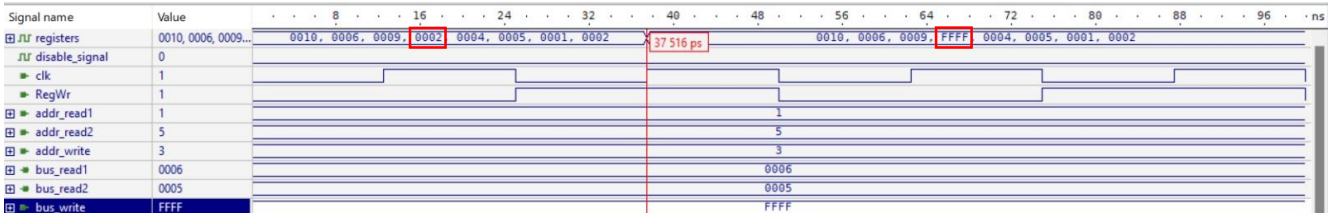

Figure 16: Register File Simulation

In the register file, the reading of the addresses (addr_read1, addr_read2, addr_write) is not dependent on the clock signal. These values are fetched immediately when required.

First Positive Edge of the Clock:

When the first positive edge of the clock arrives, the values stored at the address inputs (addr_read1 and addr_read2) are used to fetch the corresponding data from the registers.

The fetched values are placed on the respective data buses:

- **RegWr** = 0 (register write is disabled)
- **Bus_read1** = 0006 (data from register at addr_read1)
- **Bus_read2** = 0005 (data from register at addr_read2)

Second Positive Edge of the Clock:

When the second positive edge of the clock arrives, the register values are again fetched based on the addresses. This time, since RegWr = 1, the write-back operation is enabled.

The value from bus_write is written to the register specified by addr_write (which is register 3):

- **RegWr** = 1 (register write is enabled)
- **Register[3]** = FFFF (the value from bus_write is written to register 3)
- **Bus_read1** = 0006 (data from register at addr_read1)
- **Bus_read2** = 0005 (data from register at addr_read2)

2.1.3. Instruction Memory

```
module instruction_memory (addr,instruction);

    parameter instruction_size = 16, address_width = 16;
    input [(address_width-1):0] addr;
    output reg [(instruction_size -1):0] instruction ;

    reg [15:0] memory [0:65535]; // 2^16

    initial begin


        {memory[0]} = 16'b0000001010100000; // R-Type: Opcode=0000 (R-Type), Rd=001, Rs=010, Rt=100, Func=000  AND          02A0
        {memory[1]} = 16'b0011001101000011; // I-Type: Opcode=0011 (ADDI-Type),  Rs=001, Rt= 101, Imm=000011 (signed)       3343
```

Figure 17: Instruction Memory Simulation

Memory instructions store instructions at specific addresses. In this example, when we request the instruction at address 0000, the instruction stored at memory address zero, which is 4h02A0, will be retrieved and correctly displayed in the instruction register.

2.1.4. MUX 8x1

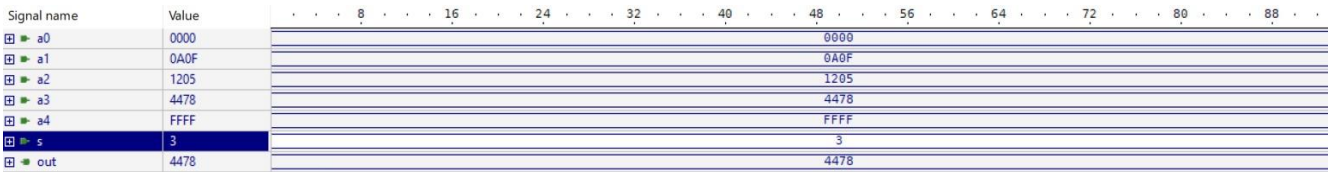

Figure 18: MUX 8x1 Simulation

This is an 8-to-1 multiplexer, but we only use selection lines from 0 to 3. When the selection is set to 3, as shown in the simulation, the output will be a3, which is correctly displayed in the output buffer.

2.1.5. Extender

It is used for both signed and unsigned extensions.

- **Signed extension**

Figure 19: Signed Extension Simulation

Here, we performed signed extension of the number 0x30 by setting sign = 1.

Representation:

- 30 (hex) = 11 0000 (binary, 6 bits)
- 30 (hex) after sign extension to 16 bits = 1111 1111 1111 0000

- **Unsigned extension**

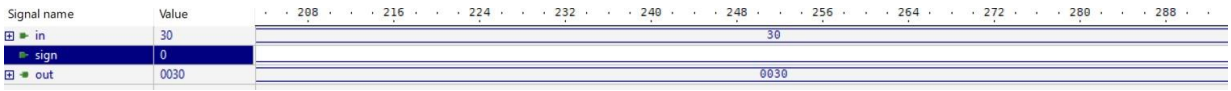

Figure 20: Unsigned Extension Simulation

Here, we performed signed extension of the number 0x30 by setting sign = 1.

Representation:

- 30 (hex) = 11 0000 (binary, 6 bits)
- 30 (hex) after sign extension to 16 bits = 0000 0000 0011 0000

2.1.6. Adder

This adder is used to calculate the next PC address by incrementing the current address by one.

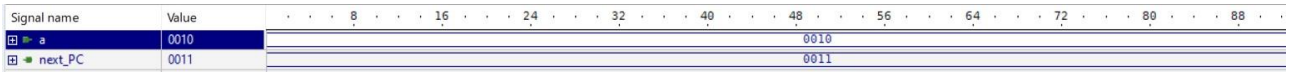

Figure 21: Adder Simulation

Here, "a" represents the current PC address with a value of (0x0010). We incremented it by 1, as shown in "next_PC," where the new value of the PC is (0x0011).

2.1.7. Decode and Multiplexing

The Decode and Multiplexing Unit simplifies the decoding process by directly interpreting instruction formats, reducing the need for extra muxes. This unit is used in the decode stage to identify the opcode, fetch operands, and determine the destination register. It efficiently handles the varying bit positions of source registers (Rs) in R-type and I-type instructions, streamlining instruction decoding and minimizing complexity.

- **R_Type**

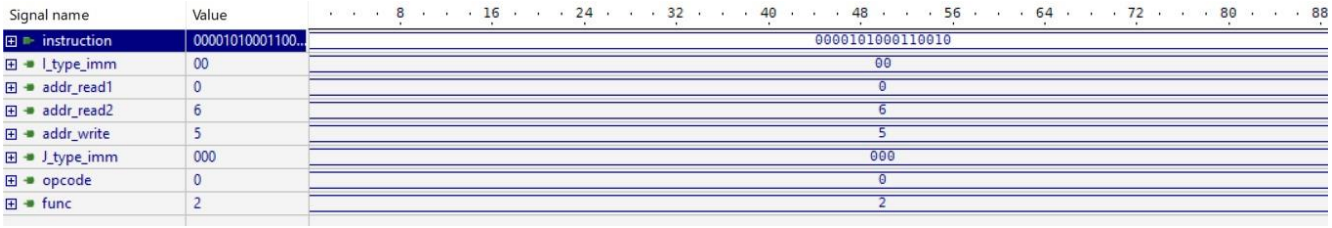

Figure 22: Decode and Multiplexing R-Type Simulation

The instruction is : 0000 101 000 110 010

As shown above, the opcode is (0000), indicating an R-Type instruction. The function is (010), which corresponds to a subtraction operation.

- Addr_read1, representing Rs, is 0 (000).
- Addr_read2, representing Rt, is 6 (110).
- Addr_write, representing Rd, is 5 (101).

R-Type instructions do not have an immediate value, so it is set to zero.

- **I_Type**

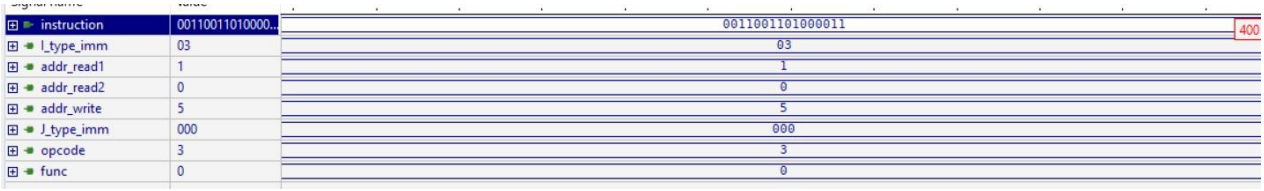

Figure 23: Decode and Multiplexing I-Type Simulation

The instruction is : 0011 001 101 000011

As shown above, the opcode is (0011), indicating that it is an I-Type instruction, specifically an ADDI operation.

- Addr_read1, representing Rs, is 1 (001).
- Addr_read2, representing Rt, is 5 (101).
- I_Type_imm, representing immediate, is 3 (00 0011).

I-Type instructions do not include a register destination (Rd) value, function, or J_Type_immediate, so these are set to zero.

2.1.8. Data Memory

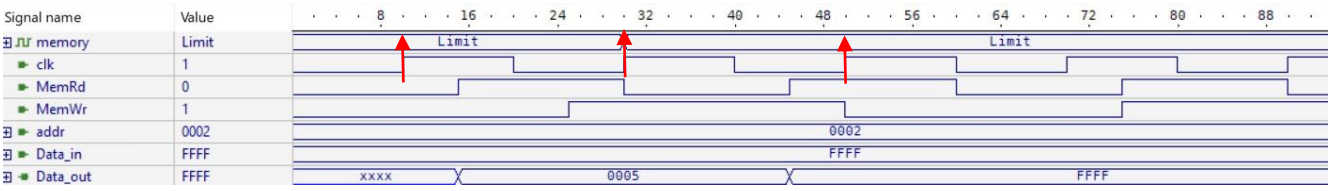

Figure 24: Data Memory Simulation

Memory Read and Write Operations:

Both MemRd (Memory Read) and MemWr (Memory Write) operate on the positive edge of the clock.

- **First Positive Edge:**

– MemRd is 0, so no data can be read from Data_in and passed to Data_out.

– MemWr is also 0, so no data can be written to memory[2], and as a result, Data_out remains undefined (represented as x).

- **Second Positive Edge:**

– MemRd is 1, so the data stored at memory[2] (which is 0x0005) is passed to Data_out.

– Additionally, since MemWr is 1, the value from Data_in (0xFFFF) is written to memory[2].

- **Third Positive Edge:**

– MemRd remains 1, so the updated value stored at memory[2] (now 0xFFFF) is passed to Data_out.

2.1.9. Zero Comparator – NOR

The zero-comparator module is a logical component that performs a NOR operation. It takes the value stored in register B (Rt), applies a bitwise OR operation on all its bits, and then negates the result. If the result is zero, the zero flag is set to 1, indicating that the value is zero. If the result is non-zero, the zero flag is set to 0, indicating that the value is not zero.

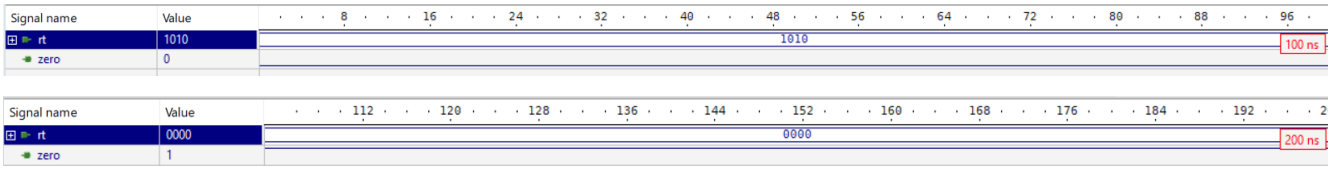

Figure 25: NOR Simulation

When rt is 1010, the zero flag will be 0 because rt is not zero. However, when rt is 0000, the zero flag will be 1, indicating that the value is zero.

2.2. Processor Simulation

The set of instructions stored in the memory.

```
 //AND
{memory[0]} = 16'b0000001010100000; // R-Type: Opcode=0000 (R-Type), Rd=001, Rs=010, Rt=100, Func=000  AND

//ADDI
{memory[1]} = 16'b0011001101000011; // I-Type: Opcode=0011 (ADDI-Type),  Rs=001, Rt= 101, Imm=000011 (signed)

//Load
{memory[2]} = 16'b0100011010000000; // LW: load word, Rs=011, Rt=010, Imm=000000 state(signed)

//Store
{memory[3]} = 16'b0101011010000000; // SW: Store word, Rs=011, Rt=010, Imm=000000 state(signed)

//BEQ
{memory[4]} = 16'b0110001100101010; // BEQ: Branch if equal, Rs=001, Rt=100, Imm=101010 (signed)

//BNE
{memory[5]} = 16'b0111110101001100; // BNQ: Branch if not equal, Rs=110, Rt=101, Imm=001100 (signed)   //jump to mem 17

//JUMP
{memory[17]} = 16'b0001000000010010; // JUMP: Opcode=0001 (jump-Type), Imm=010010 (signed)  , function=000

//JUMP
{memory[17]} = 16'b0001000000010010; // JUMP: Opcode=0001 (jump-Type), Imm=010010 (signed)  , function=000

//CALL
{memory[18]} = 16'b0001000000010101; // CALL: Opcode=0001 (call-Type), Imm=000000 (signed)  , function=001    jump to 21 then give RR <- 19

//RET
{memory[21]} = 16'b0001000000000010; // RET: Opcode=0001 (return-Type), Imm=000000 ignored(signed)  , function=010     PC <- 19

//FOR
{memory[19]} = 16'b1000001111000000; // FOR: Opcode=0000 (for-Type), Rs=001, Rt=111, Imm=000000 (signed)
```

Figure 26: Instruction Memory

The data stored within the register file.

Register	Initial value
R1	16’h6
R2	16’h9
R3	16’h2
R4	16’h4
R5	16’h5
R6	16’h2
R7	16’h2

Table 13: Register File Content

The information stored in memory.

Memory address	Values stored
0	16’h0003
1	16’h0004

2	16’h0005
3	16’h0006
4	16’h0007
5	16’h0008
6	16’h0009

Table 14: Data Memory

The first instruction in our program:

```
//AND
{memory[0]} = 16'b0000001010100000; // R-Type: Opcode=0000 (R-Type), Rd=001, Rs=010, Rt=100, Func=000  AND
```

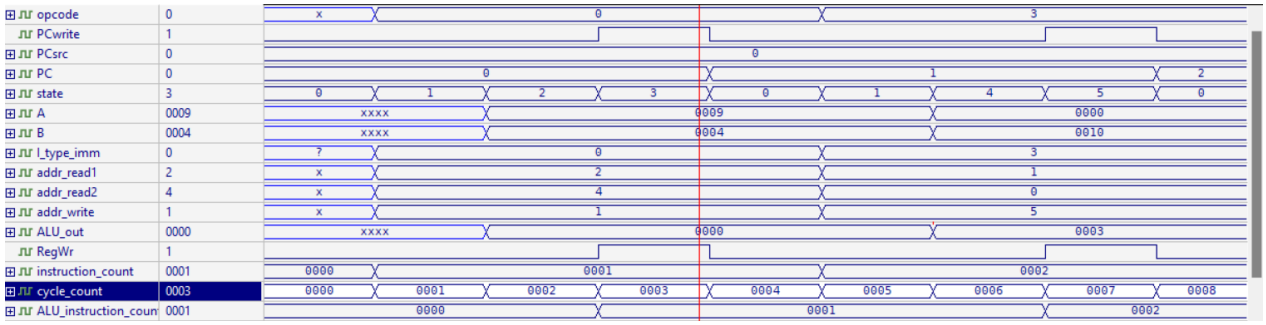

Figure 27: First Instruction Simulation

The instruction specifies a logical AND operation between the values in registers located at addresses 2 and 4, with the result of the AND operation being stored in the register at address 1. As it is an R-type instruction, the opcode is 0, and the instruction will include a function code corresponding to the AND operation.

In our state machine, the AND instruction proceeds through four stages:

- **State 0:** Instruction fetch
- **State 1:** Instruction decode
- **State 2:** R-type ALU operation
- **State 3:** R-type ALU completion

In this case, the operation performed is Reg[1] = Reg[2] & Reg[4]. With values Reg[2] = 9 and Reg[4] = 4, the result of the AND operation is 0. This result is then stored in Reg[1], making Reg[1] = 0.

By the end of this instruction, the processor has completed 4 cycles. Therefore, the instruction_count is incremented to 1, the ALU_instruction count is also incremented to 1, and the cycle_count is 4.

Summary of counts:

- instruction_count = 1
- ALU_instruction = 1
- cycle_count = 4

The Second instruction in our program:

Note that Reg[1] is updated to 0.

```
//ADDI
{memory[1]} = 16'b0011001101000011; // I-Type: Opcode=0011 (ADDI-Type),  Rs=001, Rt= 101, Imm=000011 (signed)
```

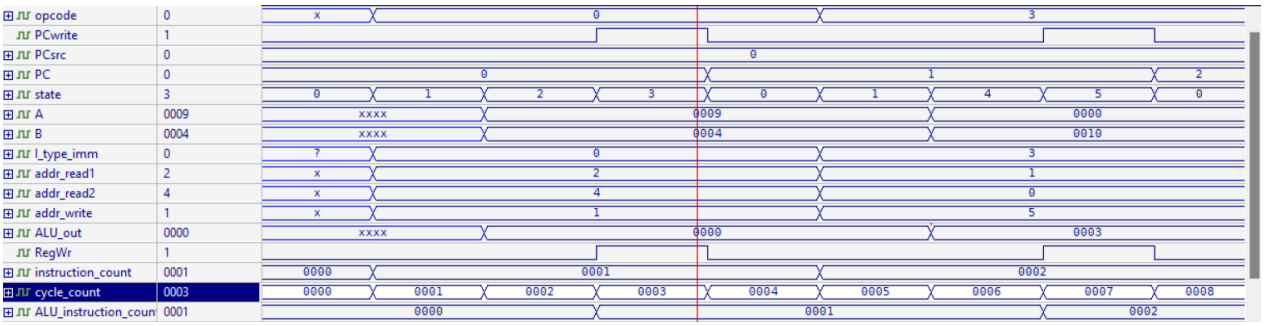

Figure 28: Second Instruction Simulation

The instruction represents an arithmetic operation, specifically an ADDI (Add Immediate), which adds the value in the register at address 1 to an immediate value of 3 and stores the result in the register at address 5. Since it is an I-type instruction, the opcode is 3, and it does not require a function field.

According to the state machine, the ADDI instruction proceeds through the following 4 steps:

- State 0: Instruction fetch
- State 1: Instruction decode
- State 4: I-type ALU operation
- State 5: I-type ALU completion

The operation Reg[5] = Reg[1] + Imm results in 0 + 3 = 3, so ALU_out will be 3, and the result is stored in the register at address 5.

By the end of this instruction, there will have been 4 cycles, increasing the instruction count by 1 and the ALU instruction count by 1.

- Instruction_count = 2
- ALU_instruction = 2
- cycle_count = 8

Instruction 3 in our program:

Note that Reg[5] is updated to 3.

```
//Load
{memory[2]} = 16'b0100011010000000; // LW: load word, Rs=011, Rt=010, Imm=000000 state(signed)
```

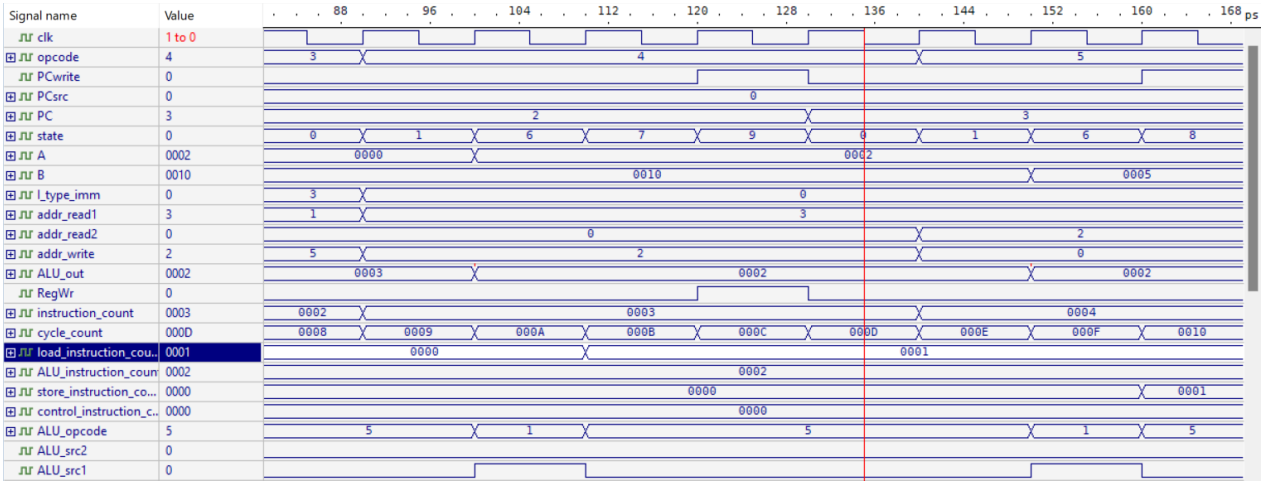

Figure 29: Third Instruction Simulation

The instruction indicates a load operation, where data is loaded from the address obtained by adding the value in Reg[3] and the immediate value (which is 0). The result of the addition is used as the address to access memory, and the retrieved data is stored in Reg[2]. This is an I-type instruction with an opcode of 4 and no function code.

According to our state machine, the load instruction goes through the following five steps:

- State 0 → Instruction fetch
- State 1 → Instruction decode
- State 6 → Address computation
- State 7 → Memory access for load
- State 9 → Load completion

The base address is calculated as Reg[3] + Imm → 0 + 2 = 2. Therefore, the ALU computes the address as 2, and the data at memory[base address] is loaded into Reg[2].

By the end of the instruction, the total number of cycles is 5. The instruction_count is incremented by one, and the load_instruction count increases by one.

- Instruction_count = 3
- ALU_instruction = 2
- Cycle_count = 13
- Load_instruction = 1

Instruction 4 in our program:

Note that Reg[2] has been updated to 5.

```
//Store
{memory[3]} = 16'b0101011010000000; // SW: Store word, Rs=011, Rt=010, Imm=000000 state(signed)
```

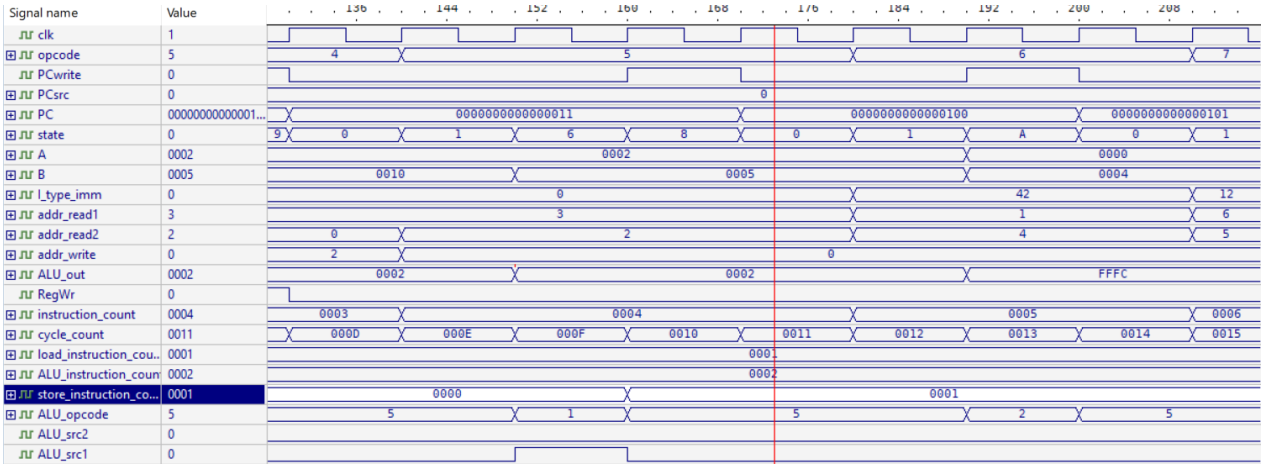

Figure 30: Forth Instruction Simulation

The instruction specifies a store operation, where data from the register at address 2 will be written to memory. The memory address is calculated by adding the value in Reg[3] and the immediate value (which is 0). This calculated address is then used to store the value from Reg[2] into memory. This is an I-type instruction, so the opcode is 5, with no specific function code.

According to our state machine, the store instruction progresses through four steps as follows:

- State 0: Instruction fetch
- State 1: Instruction decode
- State 6: Address computation
- State 8: Memory access for store

The base address is calculated as Reg[3] + Imm → 0 + 2 = 2. The ALU computes the address as 2, and the value from Reg[2] is stored at the memory location 2.

By the end of this instruction, the total number of cycles will be 4. The instruction count increases by one, and the store instruction count also increases by one.

- Instruction_count = 4
- ALU_instruction_count = 2
- Cycle_count = 17

- Load_instruction_count = 1
- Store_instruction_count = 1

Instruction 5 in our program:

```
//BEQ
{memory[4]} = 16'b0110001100101010; // BEQ: Branch if equal, Rs=001, Rt=100, Imm=101010 (signed)
```

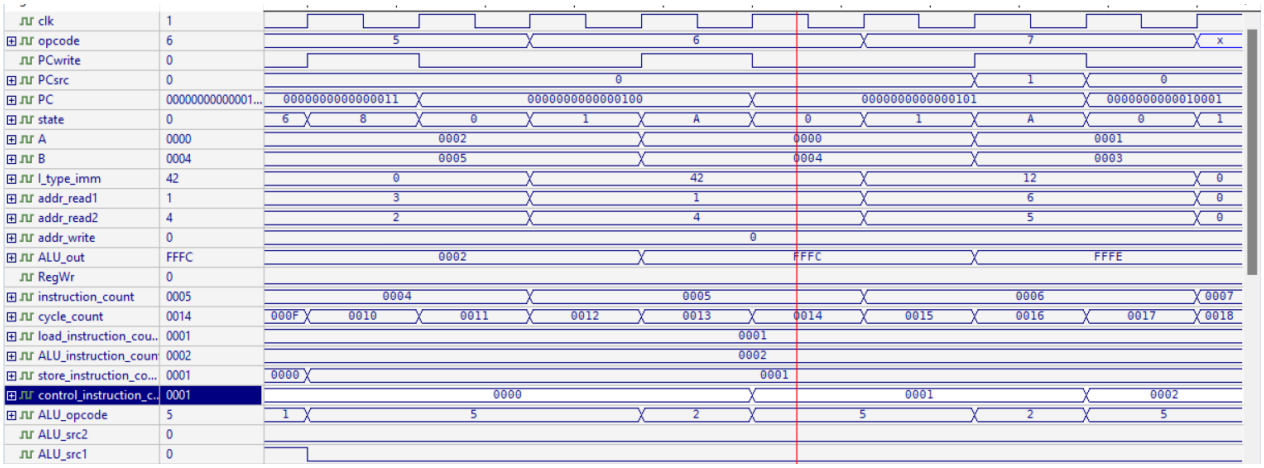

Figure 31: Fifth Instruction Simulation

The instruction specifies a conditional branch operation, BEQ, which compares the values in Reg[1] and Reg[4] by subtracting them. If the result is zero, the zero flag is set, and the program counter (PC) will jump to the address calculated by adding the immediate value to the current PC. In this case, since the values in Reg[1] and Reg[4] are not equal, the zero flag is not set, and the PC will not jump, instead incrementing by 1. This is an I-type instruction, with an opcode of 6 and no function code.

According to our state machine, the BEQ instruction progresses through three steps:

- State 0: Instruction fetch
- State 1: Instruction decode
- State 10: Branch completion

The ALU computes Reg[1] - Reg[4] → 0 - 4, which is not zero, so the zero flag remains off, and the PC is incremented by 1, not jumping to the address based on the branch offset.

By the end of this instruction, the total number of cycles will be 3. The instruction count will increase by one, and the control instruction count will also increase by one.

- Instruction_count = 5
- ALU_instruction_count = 2
- Cycle_count = 20
- Load_instruction_count = 1
- Store_instruction_count = 1
- Control_instruction_count = 1

Instruction 6 in our program:

```
//BNE
{memory[5]} = 16'b0111110101001100; // BNQ: Branch if not equal, Rs=110, Rt=101, Imm=001100 (signed)   //jump to mem 17
```

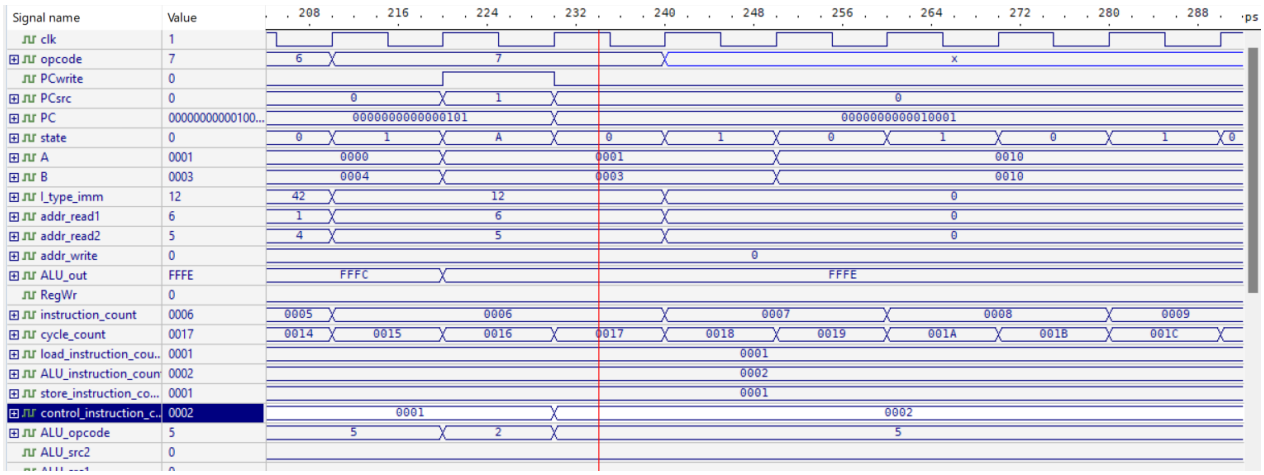

Figure 32: Sixth Instruction Simulation

The instruction specifies a conditional branch operation, BNE, which compares the values in Reg[6] and Reg[5] by subtracting them. If the result is non-zero, the zero flag will not be set, and the program counter (PC) will jump to the address calculated by adding the immediate

value (Imm) to the current PC. In this case, since the values in Reg[6] and Reg[5] are not equal, the zero flag will not be set, and the PC will jump. The jump address is calculated as PC + Imm, which equals 5 (current PC) + 12 (Imm) = 17. This is an I-type instruction with an opcode of 7 and no function code.

According to our state machine, the BNE instruction progresses through three steps:

- State 0: Instruction fetch
- State 1: Instruction decode
- State 10: Branch completion

The ALU computes Reg[6] - Reg[5] → 1 - 2, which is not zero, so the zero flag remains off, and the PC is updated by adding the immediate value to the current PC (PC = 5 + 12 = 17). Therefore, the next instruction will be located at address 17.

By the end of this instruction, the total number of cycles will be 3. The instruction count will increase by one, and the control instruction count will also increase by one.

- Instruction_count = 6
- ALU_instruction_count = 2
- Cycle_count = 23
- Load_instruction_count = 1
- Store_instruction_count = 1
- Control_instruction_count = 2

Instruction 7 in our program:

```
//JUMP
{memory[17]} = 16'b0001000000010010; // JUMP: Opcode=0001 (jump-Type), Imm=000010010 (signed)  , function=000
```

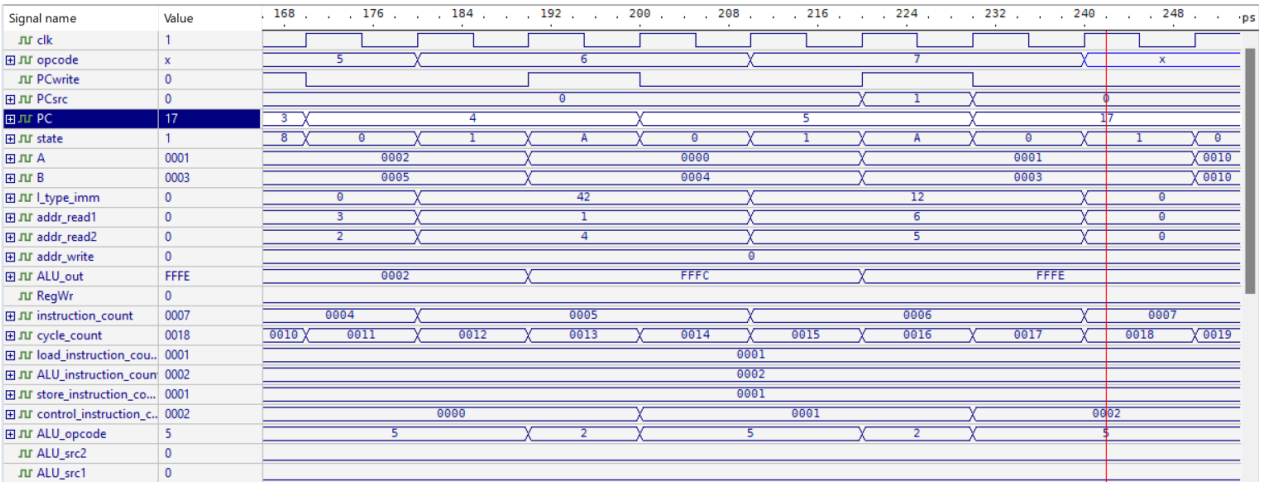

Figure 33: Seventh Instruction Simulation

The instruction specifies an unconditional jump operation, which directs the program to jump to a target instruction. The jump uses the immediate value (Imm) of 18, which will be concatenated to the upper part of the current PC (bits [15:9]), resulting in a new PC value of 18.

According to our state machine, the jump instruction progresses through two steps:

- State 0: Instruction fetch
- State 1: Instruction decode

In this case, the PC is updated to the new target address, which is formed by concatenating the top part of the current PC ([15:9] bits) with the target instruction (Imm = 18). As a result, the new PC value becomes 18.

By the end of this instruction, the total number of cycles will be 2. The instruction count will increase by one, and the control instruction count will also increase by one.

- Instruction_count = 7
- ALU_instruction_count = 2
- Cycle_count = 25
- Load_instruction_count = 1

- Store_instruction_count = 1
- Control_instruction_count = 3

Instruction 8 in our program:

```
//CALL
{memory[18]} = 16'b0001000000010101; // CALL: Opcode=0001 (call-Type), Imm=000000 (signed)  , function=001    jump to 21 then give RR <- 19
```

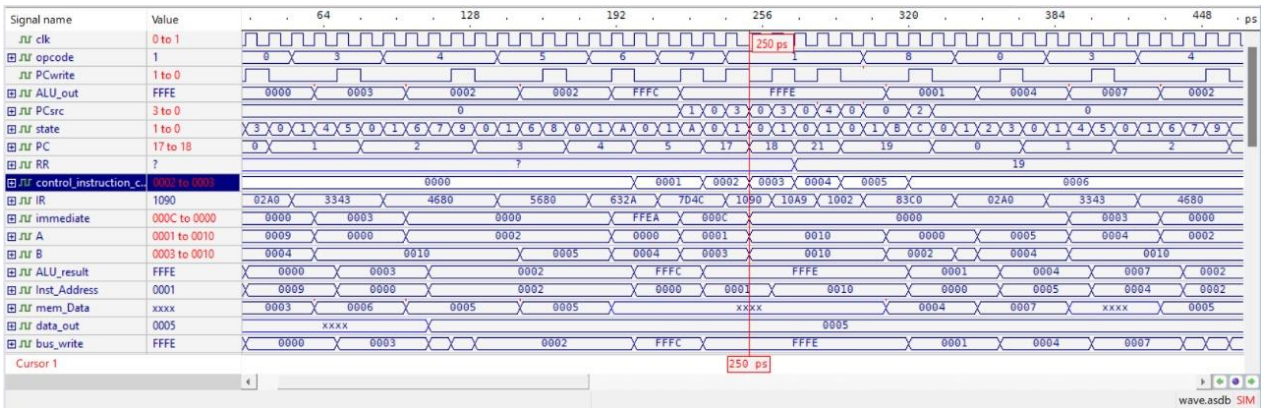

Figure 34: Eighth Instruction Simulation

The instruction specifies an unconditional call operation, which primarily saves the current PC + 1 into the return register (RR) and then jumps to a target address similar to a jump instruction. In this case, the current PC is 18, and the target address is calculated by concatenating the immediate value (Imm) of 21 to the upper bits of the PC ([15:9]), resulting in a new PC value of 21. Meanwhile, the value saved in the RR is the current PC + 1, which is 18 + 1 = 19.

According to our state machine, the call instruction progresses through two steps:

- State 0: Instruction fetch
- State 1: Instruction decode

The PC is updated to the target address, formed by concatenating the upper bits of the current PC with the immediate value. Additionally, the current PC value + 1 (19) is saved in the RR.

By the end of this instruction, the total number of cycles will be 2. The instruction count will increase by one, and the control instruction count will also increase by one.

- Instruction_count = 8
- ALU_instruction_count = 2
- Cycle_count = 27
- Load_instruction_count = 1
- Store_instruction_count = 1
- Control_instruction_count = 4

Instruction 9 in our program:

```
//RET
{memory[21]} = 16'b0001000000000010; // RET: Opcode=0001 (return-Type), Imm=000000 ignored(signed)  , function=010     PC <- 19
```

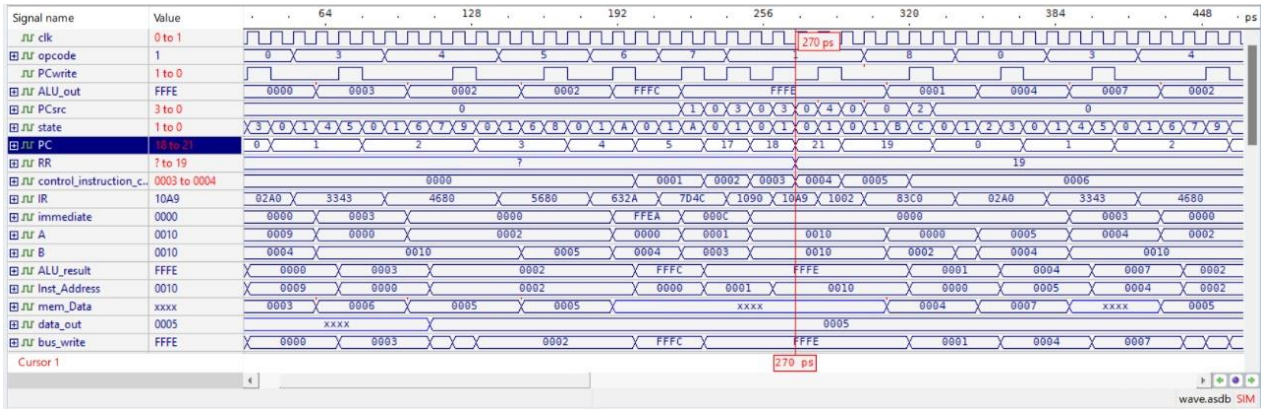

Figure 35: Ninth Instruction Simulation

The instruction specifies a return operation (RET), which primarily retrieves the value from the return register (RR) and updates the program counter (PC) to that value. In this case, the current PC is 21, and it will be updated to the value stored in RR, which is the PC + 1 from the previous instruction, equaling 19.

According to our state machine, the return instruction progresses through two steps:

- State 0: Instruction fetch
- State 1: Instruction decode

The PC is updated to the value stored in the RR, which was 19.

By the end of this instruction, the total number of cycles will be 2. The instruction count will increase by one, and the control instruction count will also increase by one.

- Instruction_count = 9
- ALU_instruction_count = 2
- Cycle_count = 29
- Load_instruction_count = 1
- Store_instruction_count = 1
- Control_instruction_count = 5

Instruction 10 in our program:

```
//FOR
{memory[19]} = 16'b1000001111000000; // FOR: Opcode=0000 (for-Type), Rs=001, Rt=111, Imm=000000 (signed)
```

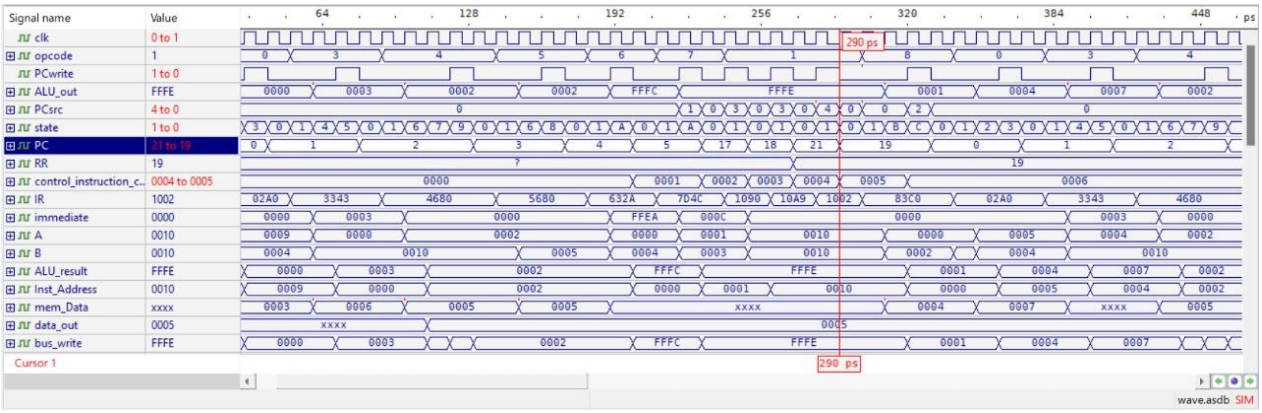

Figure 36: Tenth Instruction Simulation

The instruction specifies an unconditional "FOR" loop operation, where the base address is stored in Reg[1] and the loop iterates based on the value in Reg[3]. The loop will decrement the value in Reg[3] by 1 on each iteration, and the process will continue until the zero flag is set. Once the loop finishes, the next program counter (PC) will be updated to the value of PC + Imm, where the immediate value is provided in the instruction.

According to our state machine, the "FOR" loop instruction will go through the following four steps:

- State 0: Instruction fetch
- State 1: Instruction decode
- State 11: FOR computation
- State 12: FOR completion

By the end of this instruction, the total number of cycles will be 4. The instruction count will increase by one, and the control instruction count will increase by one as well.

- Instruction_count = 7
- ALU_instruction_count = 2
- Cycle_count = 25
- Load_instruction_count = 1
- Store_instruction_count = 1
- Control_instruction_count = 6

Teamwork

The success of this project was made possible by the collaborative efforts of the team, where each member brought their unique strengths and expertise to the table. The project was divided into well-defined tasks, with each team member taking ownership of specific sections. We worked together to ensure that our individual contributions aligned with the overall objectives, leveraging our collective skills to address challenges as they arose.

Clear communication and regular meetings allowed us to stay on track and ensure that everyone was informed about the project's progress. We utilized various tools for collaboration, such as version control systems for code sharing, project management platforms for task tracking, and collaborative documentation tools. This enabled us to efficiently exchange ideas, review progress, and resolve issues in a timely manner.

During the course of the project, we encountered several technical challenges, but through collective brainstorming and problem-solving, we were able to overcome them. Whether debugging code, optimizing designs, or refining processes, the teamwork demonstrated throughout the project was crucial in achieving our milestones.

In conclusion, the project's success is a testament to the strength of the team's collaboration, and it showcased our ability to effectively combine individual efforts to meet common goals. We are proud of the synergy created by the group, and the final results reflect the collective dedication and hard work invested by each team member.

Conclusion

In conclusion, this project has successfully demonstrated the integration of various components and techniques to achieve the desired outcomes. Through careful design and implementation, we have effectively utilized state machines, ALU operations, and various instruction types to execute the given tasks efficiently. The project has not only allowed us to apply theoretical concepts but also provided valuable hands-on experience in working with hardware description languages, state-based control, and instruction set architecture.

By utilizing a structured and collaborative approach, the team was able to overcome challenges and deliver a functional system that meets the objectives outlined in the initial design. Each instruction's execution cycle was thoroughly tested and verified, ensuring that the system operates as expected. Additionally, the careful attention to performance optimization and resource utilization has resulted in an efficient solution.

Overall, the project has been an enriching learning experience, enhancing our understanding of computer architecture, state machines, and instruction-level operations. The success of this project underscores the importance of teamwork, clear communication, and systematic problem-solving in achieving project goals.