BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS4370

COMPUTER ARCHITECTURE

Design and Verification of a Multi-Cycle RISC Processor in Verilog

---

Partners:	Hasan Qarmash	**ID:** 1210611
	Ahmad Saqer	**ID:** 1210085
	Arwa Doha	**ID:** 1190324

Instructors Names: Dr. Aziz Qaroush.

Dr. Ayman Hroub.

Sections: 1, 2

June , 22 2024

Abstract

The aim of this project is to develop and validate a 16-bit multi-cycle Reduced Instruction Set Computing (RISC) processor by using the Verilog hardware description language . The processor employs a multi-cycle execution architecture to maximize resource efficiency and streamline control logic. It supports four different types of instructions: R-type, I-type, J-type, and S-type. The program counter, register file, ALU, instruction memory, data memory, and control unit are among the essential parts of the processor architecture. Each of these parts is essential to the instruction execution pipeline. A foundational understanding of multi-cycle processor design is provided, as well as a practical demonstration of digital logic and computer architecture principles within a Verilog-based simulation environment. Extensive testbenches and code sequences are developed to rigorously verify the correctness and functionality of the processor design.

Table of Contents

Table of Figures

Design Specification

1.1 Processor Specifications

Below is an outline of the processor design, including the key modules and their specifications. This design is a high-level approach and should be refined and verified through detailed implementation and simulation.

- ✓ **Instruction Size**: 16 bits
- ✓ **Word Size**: 16 bits
- ✓ **General-Purpose Registers**: 8 registers (R0-R7, where R0 is hardwired to zero)
- ✓ **Program Counter (PC)**: 16-bit special-purpose register
- ✓ **Instruction Types**: R-type, I-type, J-type, S-type
- ✓ **Memory**: Separate data and instruction memories
- ✓ **Memory Addressing**: Byte addressable, Little endian

1.2 Instruction Formats

As mentioned above, this ISA has four instruction formats, namely, R-type, I-type, J-type, and S-type. These four types have the following common fields:

1.2.1 R-Type (Register Type):

Opcode^4	Rd^3	Rs1^3	Rs2^3	Unused^3

- 3-bit Rd: destination register
- 3-bit Rs1: first source register
- 3-bit Rs2: second source register
- 3-bit unused
- 4-bit opcode

1.2.2 I-Type (Immediate Type)

Opcode^4	m^1	Rd^3	Rs1^3	Immediate^5

- 3-bit Rd: destination register
- 3-bit Rs1: first source register
- 5-bit immediate: unsigned for logic instructions, signed otherwise

- 1-bit mode: used with load and branch instructions
- 4-bit opcode

1.2.3 J-Type (Jump Type)

Opcode^4	Jump Offset^{12}

- 12-bit offset: target address calculation

- jmp L (Unconditional jump to the target L)
  - Format: opcode (4 bits) | offset (12 bits)
  - The target address is calculated by concatenating the most significant 7 bits of the current PC with the 12-bit offset after multiplying the offset by 2.

- call F (Call the function F)
  - Format: opcode (4 bits) | offset (12 bits)
  - The return address is pushed onto R7, and the target address is calculated similarly to jmp.

- ret (Return from a function)

Opcode^4	Unused^{12}

  - Format: opcode (4 bits) | unused (12 bits)
  - The next PC is the value stored in R7.

1.2.4 S-Type (Store)

Opcode^4	Rs^3	Immediate^8

- 3-bit rs: source register.
- 9-bit immediate: address offset0
- 4-bit opcode.

Processor Stages Description

This section of the report will delve into a comprehensive examination of the design and testing procedures for each crucial component within the multiprocessor data path. We will meticulously analyze and evaluate the processes involved in fetching, decoding, executing, accessing memory, and writing back data. These intricate stages will be implemented and thoroughly tested using the Verilog hardware description language in the Active-HDL software environment.

2.1 Instruction Fetch Stage (IF)

The instruction fetch stage represents the crucial first step in the data path's operation. This stage is responsible for retrieving the next instruction from the instruction memory block and preparing it for further processing in subsequent stages. Crucially, the program counter (PC) is also updated during the fetch stage to ready the system for the next instruction to be retrieved. The instruction memory module serves as the central repository for the 21 instructions that comprise the 4 instruction types (R-type, I-type, J-type, and S-type) of the RISC processor. This module is tasked with providing the appropriate next instruction to the fetching stage, enabling the seamless execution of the processor's operations.

Firstly, The instructionMemory module features the following input and output ports:

Inputs:-

1. **clock**: This is a clock signal input (input wire clock). It is used to synchronize the reading of the instruction memory. The instruction register is updated on the positive edge of this clock signal.
2. **AddressBus**: This is a 16-bit address input (input wire [15:0] AddressBus). It specifies the address in the instruction memory from which the instruction should be read.

Output:-

1. **InstructionReg**: This is a 16-bit output register (output reg [15:0] InstructionReg). It holds the instruction fetched from the memory corresponding to the address provided on the AddressBus.

```
module instructionMemory(clock, AddressBus, InstructionReg);

    // clock
    input wire clock;

    // address bus
    input wire [15:0] AddressBus;


    // instruction register
    output reg [0:15] InstructionReg;


    // instruction memory
    reg [15:0] instruction_memory [0:127];
```

Figure 1: Instruction Memory module inputs and outputs

Initialization:

- When the simulation starts, the initial block runs, clearing the memory and loading the predefined instructions.

Fetching Instructions:

- During each clock cycle (on the positive edge of the clock signal), the always block reads the instruction from the memory location specified by the AddressBus.
- The fetched instruction is stored in InstructionReg, making it available for further processing by the processor.

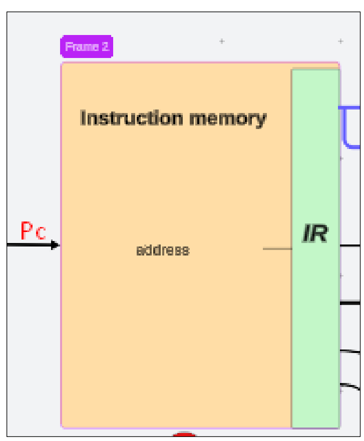

Figure 2: Instruction Memory Block Diagram

This module serves as a basic component in a processor's instruction fetch stage, providing instructions for execution based on the current program counter value.

2.2 Instruction Decode Stage (ID)

Following the instruction fetch stage, the instruction decode (ID) stage represents the next critical component of the data path. This stage plays a pivotal role in preparing the necessary information and operands for the subsequent execution stage. Key responsibilities of the ID stage include extracting the relevant fields from the fetched instruction, extending the immediate values using a dedicated extender module, and obtaining the required register values from the register file. By performing these tasks, the ID stage ensures that the execution stage has access to all the necessary data and instructions to carry out the desired operations.

2.2.1 Register File

The Register File module serves as a crucial component within multiprocessor systems, facilitating the storage and retrieval of data through a register-based approach. This module enables both read and write operations on the registers, with the specific actions determined by the input addresses and control signals. The key control signals include the WriteEnable signal, which is regulated by the system's control unit, as well as the input clock signal. By managing these inputs, the Register File module provides the necessary data access and manipulation capabilities to support the overall functionality of the multiprocessor system.

To design a register file module that interfaces well with the given instructionMemory module and the described ISA, we need to consider the requirements for reading and writing registers as specified by the instructions. Here’s a breakdown of the inputs and outputs needed for the register file module.

Register File Inputs and Outputs

Inputs:

- **clock**: Clock signal to synchronize read and write operations.
- **reset**: Reset signal to initialize registers.
- **readReg1**: Address of the first register to read (3 bits).
- **readReg2**: Address of the second register to read (3 bits).

- **writeReg**: Address of the register to write (3 bits).
- **writeData**: Data to be written to the register (16 bits).
- **regWrite**: Control signal to enable writing to a register (1 bit).

Outputs:

- **readData1**: Data read from the first register (16 bits).
- **readData2**: Data read from the second register (16 bits).

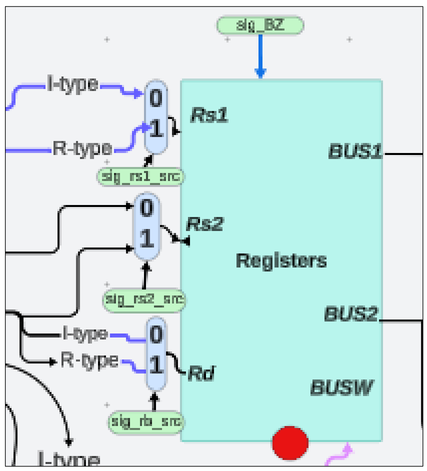

Figure 3: Register File Block Diagram

2.3 Execution Stage

The execution stage is responsible for executing the instruction based on the decoded operation and operands. It utilizes the ALU module to perform the required arithmetic and logical operations, producing the desired result. The ALU module acts as the engine for the execution stage, carrying out a range of operations as directed by the input data and control signals.

To design an ALU (Arithmetic Logic Unit) module that fits well with the provided instructionMemory and registerFile modules, we need to consider the operations required by the instruction set architecture (ISA) provided. The ALU should handle operations like addition, subtraction, AND, and comparisons for branch instructions.

ALU Module Design :

The ALU will have the following inputs and outputs:

Inputs

- **A**: First operand (16 bits).
- **B**: Second operand (16 bits).
- **ALUControl**: Control signal to specify the operation (4 bits).

Outputs

- **Result**: The result of the ALU operation (16 bits).
- **Zero**: A flag that is high (1) when the result is zero (1 bit).
- **Negative**: A flag that is high (1) when the result is negative (1 bit).
- **Carry**: A flag that indicates a carry out (1 bit).
- **Overflow**: A flag that indicates an overflow (1 bit).

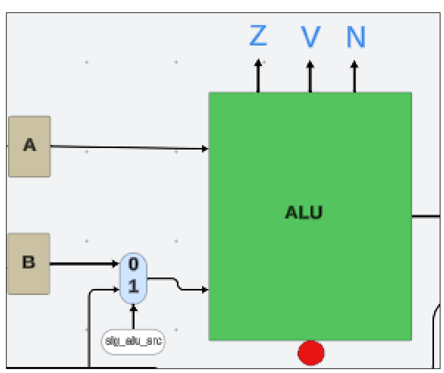

Figure 4: ALU Block Diagram.

2.4 Memory Access

The Data Memory block, the fourth stage of the multiprocessor data path, is responsible for storing and retrieving data during program execution.

To implement the Memory Access stage of our multi-cycle processor, we need to design modules for both data memory and instruction memory. These modules will interact with the control unit and the rest of the processor pipeline to read and write data based on the instruction being executed. Given the instruction set and the memory specifications, the Memory Access stage handles loading data from memory and storing data to memory.

2.4.1 Data Memory Module Design

The data memory module will have the following inputs and outputs:

2.4.1.1 Inputs

1. **clock**: Clock signal for synchronization.
2. **memRead**: Control signal to enable reading from memory.
3. **memWrite**: Control signal to enable writing to memory.
4. **address**: Memory address from which to read or write data (16 bits).
5. **writeData**: Data to be written to memory (16 bits).

2.4.1.2 Outputs

1. **readData**: Data read from memory (16 bits)

The following figure shows the block diagram of the data memory stage with its inputs and outputs:

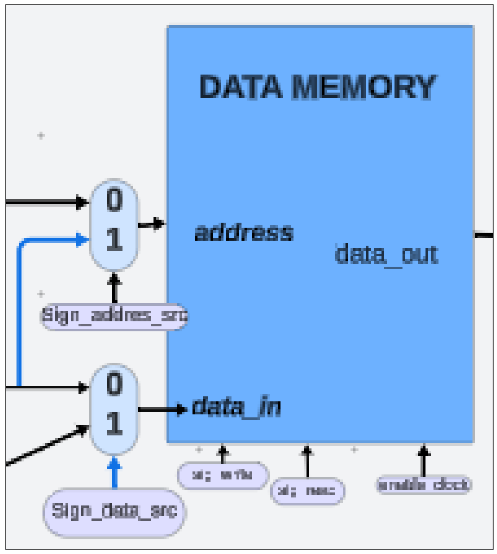

Figure 5: The block diagram of the data memory stage.

2.5 Write Back

The Write Back stage completes the execution of an instruction by updating the register file with the result of ALU operations, memory reads, or special values like return addresses. It ensures that the processor state is updated according to the instruction being executed.

In the write back stage, the processor writes the result of the ALU operation or the data read from memory back to the register file. Here's how it works:

1. **Control**:
   - This stage is activated based on control signals indicating that the instruction being executed requires writing back to the register file.
2. **Data Selection**:
   - The data to be written back depends on the instruction type and the control signals.
   - For most instructions, the result of the ALU operation or the data read from memory is selected for writing back.
3. **Write Address**:
   - The destination register address is determined based on the instruction type and control signals.
   - For R-Type and I-Type instructions, the destination register address is obtained from the instruction.
   - For special instructions like `CALL`, the return address is written to a specific register (e.g., `R7`).
4. **Write Back Operation**:
   - The selected data (ALU result or memory read data) is written back to the register file.
   - In the case of `CALL` or other special instructions, the return address is written to the designated register.

In our Verilog code, the write back stage is implemented in the following:

```
assign BusW = (sig_write_back_data_select == 0 && OpCode == CALL) ?(PC + 16'd2) :
    (sig_write_back_data_select == 0 ) ? ALU_Output : DataMemoryOutputBus;
```

Figure 6: Write back code in riscProcessor

- ✓ the BusW signal holds the data to be written back to the register file.
- ✓ The value of **BusW** depends on the control signals and the instruction being executed.
- ✓ For example, if **sig_write_back_data_select** indicates that the data comes from the ALU, **ALU_Output** will be written back.
- ✓ If it's a **CALL** instruction, the return address (PC + 2) is written back.
- ✓ DataMemoryOutputBus holds the data read from memory, which is written back if applicable.

Control Unit

The control unit in our processor design is responsible for generating the control signals required for the proper operation of each stage of the instruction cycle (fetch, decode, execution, memory access, and write back). It uses the opcode and flags (zero, negative, overflow) to generate these signals.

3.1 Control Unit Description:

Inputs:

- **clock**: The clock signal to synchronize the control unit.
- **OpCode**: The opcode from the instruction register, which determines the type of instruction being executed.
- **flag_zero**: Zero flag from the ALU indicating if the result of the ALU operation is zero.
- **flag_negative**: Negative flag from the ALU indicating if the result of the ALU operation is negative.
- **flag_overflow**: Overflow flag from the ALU indicating if there was an overflow in the ALU operation.
- **mood**: A signal indicating if the immediate value should be treated as signed or unsigned.
- **sig_BZ**: A control signal related to branching instructions.

Outputs:

- **sig_alu_op**: ALU operation control signal.
- **sig_pc_src**: Program counter source control signal.
- **sig_rb_src, sig_rs1_src, sig_rs2_src, sig_rsORpc_src, sig_BZ**: Register and branching control signals.
- **sig_alu_src**: ALU source control signal.
- **sig_rf_enable_write**: Register file write enable signal.
- **sig_enable_data_memory_write**: Data memory write enable signal.
- **sig_enable_data_memory_read**: Data memory read enable signal.
- **sig_write_back_data_select**: Write-back data select signal.
- **sig_address_src**: Address source control signal for memory operations.

- **sig_data_src**: Data source control signal for memory operations.
- **en_instruction_fetch, en_instruction_decode, en_execute**: Stage enable signals.

Enable Signals in the Control Unit:

- **en_instruction_fetch**: This signal enables the instruction fetch stage. When asserted, it allows the program counter (PC) to update and fetch the next instruction from the instruction memory.
- **en_instruction_decode**: This signal enables the instruction decode stage. When asserted, it allows the instruction register to decode the fetched instruction and generate the necessary control signals for the subsequent stages.
- **en_execute**: This signal enables the execute stage. When asserted, it allows the ALU to perform the required arithmetic or logical operation based on the decoded instruction and the generated control signals.

3.2 The Data path Control Signal:

Table 1: The Data path Control Signal

Control signal name	Explanation
sig_alu_op	Specifies the operation to be performed by the ALU (add, subtract, etc.).
sig_pc_src	Selects the source for the next value of the Program Counter (PC).
sig_rb_src	Selects the source for the RB register input (used in register file).
sig_rsORpc_src	Selects between using RS or PC for a certain operation (used in register file).
sig_rs1_src	Selects the source for the RS1 register input.
sig_rs2_src	Selects the source for the RS2 register input.
sig_alu_src	Selects the second operand source for the ALU (e.g., register or immediate).
sig_rf_enable_write	Enables writing to the register file.
sig_enable_data_memory_write	Enables writing to the data memory.
sig_enable_data_memory_read	Enables reading from the data memory.
sig_write_back_data_select	Selects the source for the data to be written back to the register file.
sig_address_src	Selects the source for the address used in data memory operations.
sig_data_src	Selects the source for the data to be written to memory.
en_instruction_fetch	Enables the instruction fetch stage.
en_instruction_decode	Enables the instruction decode stage.
en_execute	Enables the execute stage.

3.3 Control Truth Table:

Table 2: Control Truth Table

	M	Z &N Flag sig_BZ	sig_alu_op	sig_pc_src	sig_rb_src	sig_rs1_src	sig_rs2_src	sig_rf_enable_write	sig_write_back_data_select
ADD	X	X	sig_add	2'b00	0	0	0	1	1
SUB	X	X	sig_sub	2'b00	0	0	0	1	1
AND	X	X	sig_And	2'b00	0	0	0	1	1
ADDI	X	X	sig_add	2'b00	1	1	1	1	1
ANDI	X	X	sig_and	2'b00	1	1	1	1	1
LW	X	X	sig_add	2'b00	1	1	1	1	1
LBu	0	X	sig_add	2'b00	1	1	1	1	1
LBs	0	X	sig_add	2'b00	1	1	1	1	1
SW	X	X	sig_add	2'b00	1	1	1	0	x
JMP	X	X	X	2'b11	1	1	1	0	x
CALL	X	X	X	2'b00	1	1	1	0	x

RET	X	X	sig_add	2'b00	1	1	1	0	x
Sv	X	X	X	2'b01	1	1	1	0	x
BGT	0	X	sig_BGT	2'b10	1	1	1	0	x
BGTZ	1	X	sig_BGTZ	2'b10	1	1	1	0	x
BLT	0	X	sig_BLT	2'b10	1	1	1	0	x
BLTZ	1	X	sig_BLTZ	2'b10	1	1	1	0	x
BEQ	0	X	sig_BEQ	2'b10	1	1	1	0	x
BEQZ	1	X	sig_BEQZ	2'b10	1	1	1	0	x
BNE	0	X	sig_BNE	2'b10	1	1	1	0	x
BNEZ	1	X	sig_BNEZ	2'b10	1	1	1	0	x

3.4 Logic equation of the control signals:

- **MemRD (Memory Read):**
  - MemRD = $\overline{(\text{ LW + LBu +LBs})}$
- **MemWR (Memory Write):**
  - MemWR = SW
- **ExtOp (ALU Operation):**
  - ExtOp = $\overline{(\text{ANDI + AND})}$
- **ALUSrc (ALU Source):**
  - ALUSrc = ADDI || ANDI || BGT || BGTZ || BLT || BLTZ || BEQ || BEQZ || BNE || BNEZ || Sv
- **RegSrc (Register Source):**
  - RegSrc = $\overline{(\text{BEQ +BNE +SW})}$
- **RegWR (Register Write):**
  - RegWR = ADD || SUB || ADDI || AND || ANDI || Sv
- **WBData (Write Back Data):**
  - WBData = $\overline{(\text{LW + LBu +LBs})}$
- **PCSrc (Program Counter Source):**
  - PCSrc = BEQ || BEQZ || BNE || BNEZ || JMP || CALL

3.5 state digram :

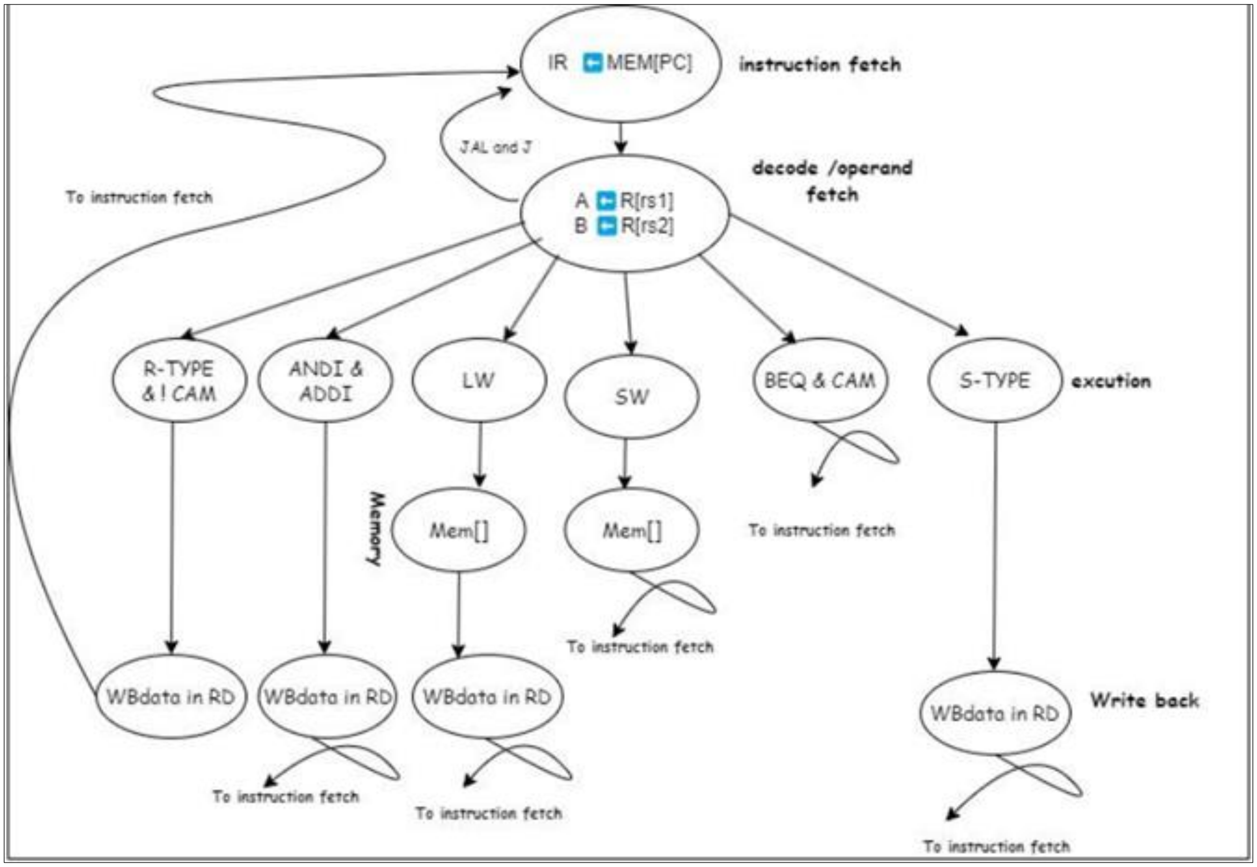

Figure 7: state digram

Project Final Datapath

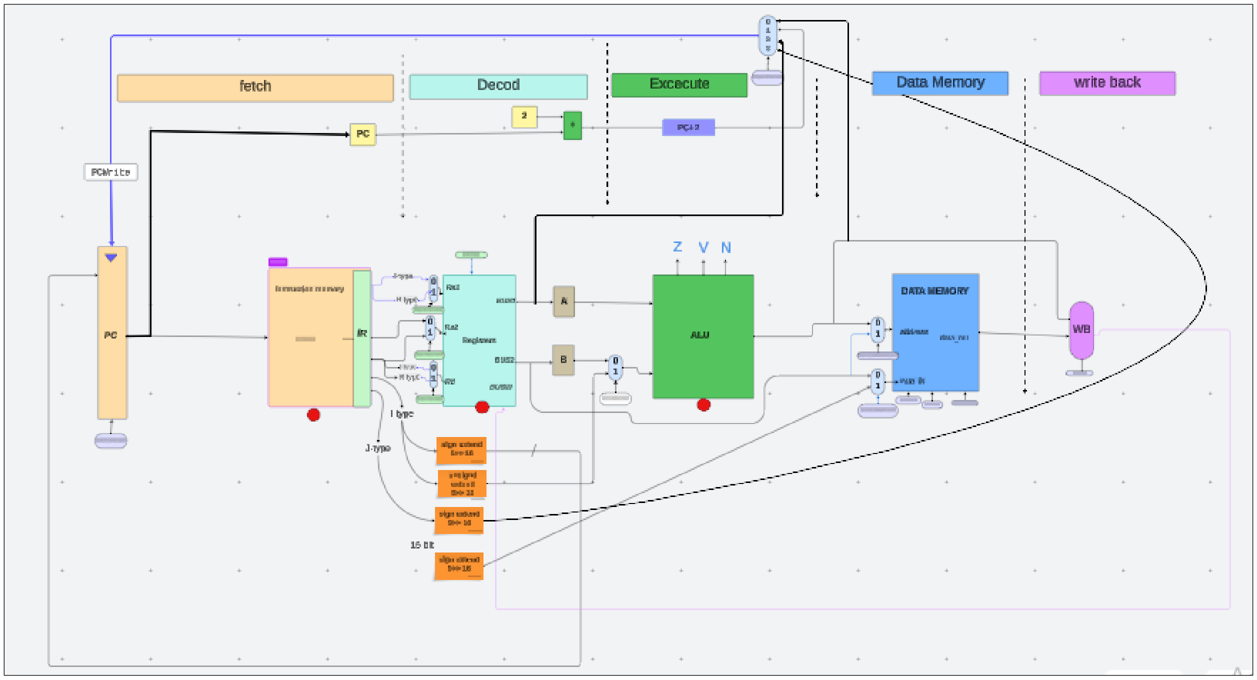

Figure 8: Final Datapath.

Simulation and Testing Results

This section summarizes the simulation and testing results for the encoding and execution of various instructions in the multi-cycle processor:

4.1 R-type instructios test

The R-type instruction test in the instruction memory includes three operations: **ADD**, **SUB**, and **AND**, and this test verifies the ALU's basic arithmetic and logical operations.

```
//-------------------R-type -test ------------------------------//

    instruction_memory[4] = { ADD, R1, R3, R2,3'b001};
    instruction_memory[6] = { SUB, R4, R3, R2,3'b001};
    instruction_memory[8] = { AND, R5, R3, R2,3'b001};
```

Figure 9: R-type test

4.1.1 ADD

When the instruction at address 4 is fetched and decoded, the processor recognizes it as an ADD operation due to the opcode 1. It takes the values in registers R3 and R2, adds them together using the ALU, and stores the result in register R1.

$$Reg(Rd) = Reg(Rs1) + Reg(Rs2)$$

Execution Details:

- **PC**: 4
- **Opcode (0001)**: This indicates a `ADD` operation.
- **Output Register**: `R1`
- **Operand A**: `R3`
- **Operand B**: `R2`

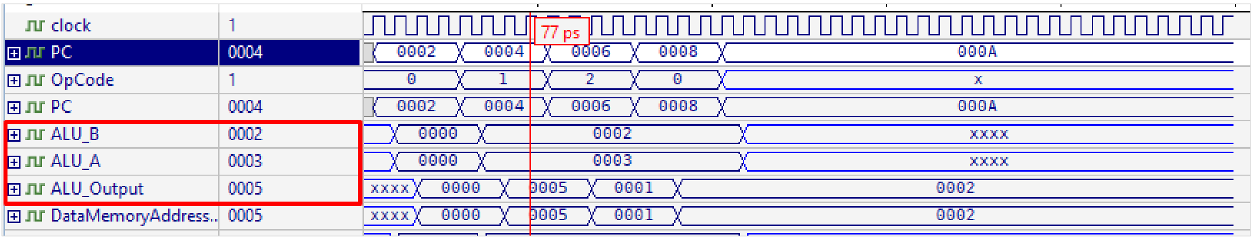

Figure 10: Simulation of ADD Instruction Execution

As shown in figure above, given the values in the registers:

- R3 = 3, R2 = 2

After executing the instruction, the value in R1 will be:

- R1 = R2 + R3 = 2 + 3 = 5

4.1.2 SUB

When the instruction at address 6 is fetched and decoded, the processor recognizes it as a SUB operation. It subtracts the value in R2 from the value in R3 and stores the result in R4.

Reg(Rd) = Reg(Rs1) - Reg(Rs2)

Execution Details:

- **PC**: 6
- **Opcode (0010)**: This indicates a SUB operation.
- **Output Register**: R4
- **Operand A**: R3
- **Operand B**: R2

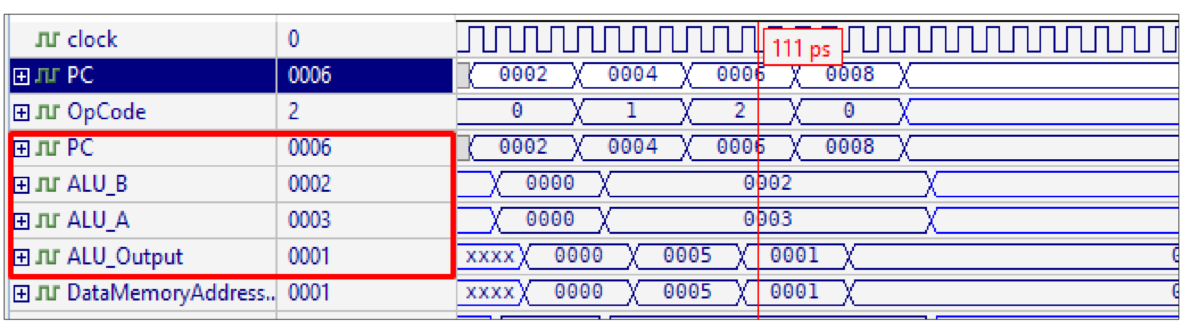

Figure 11: Simulation of SUB Instruction

As shown in figure above, given the values in the registers:

- R3 = 3
- R2 = 2

After executing the instruction, the value in R1 will be:

- R1 = R3 + R1 = 1

4.1.3 AND

When the instruction at address 8 is fetched and decoded, the processor recognizes it as an AND operation. It performs a bitwise AND between the values in R3 and R2, and stores the result in R5.

Reg(Rd) = Reg(Rs1) & Reg(Rs2)

Execution Details:

- **PC**: 8
- **Opcode (0000)**: This indicates an AND operation.
- **Output Register**: R5
- **Operand A**: R3
- **Operand B**: R2

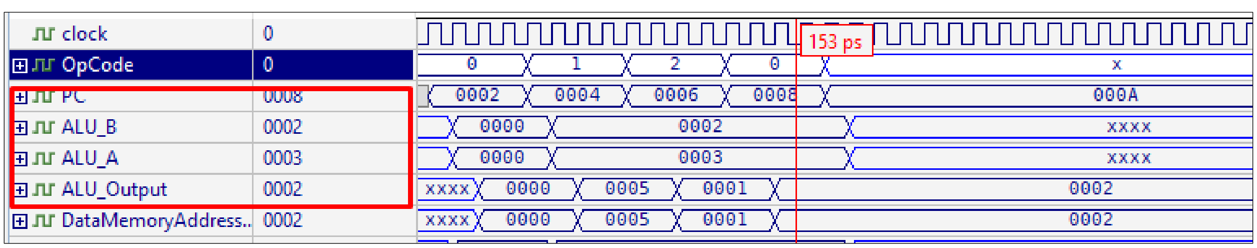

Figure 12: Simulation of AND Instruction

After executing the instruction, the result stored in R5 will be:

- R5 = R3 & R2 = 0011 & 0010 = 0010 (binary)
- R5 = 2 (decimal)

4.2 I-type instruction test

4.2.1 ADDI, ANDI Test :

Given the instructions for **ADDI** (Add Immediate) and **ANDI** (Logical AND Immediate) in the instruction memory, here are the details of the execution:

```
instruction_memory[4] = { ADDI,1'b1 ,R0, R0, 5'b00101 }
instruction_memory[6]={ ANDI,1'b1 ,R0, R2, 5'b00101 };
```

Figure 13: instructions for ADDI and ANDI

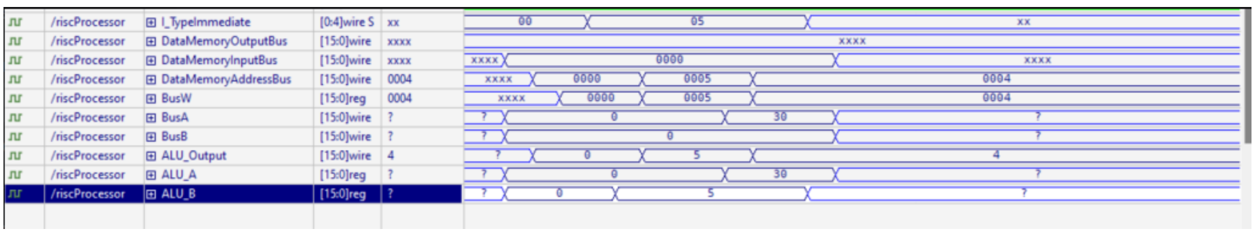

Figure 14: : Simulation for ADDI and ANDI

Instruction Execution Summary Table:-

Table 3 : Instructions for ADDI and ANDI

Instruction	ADDI Instruction	ANDI Instruction
Instruction	instruction_memory[4] = { ADDI, 1'b0, R0, R0, 5'b00101 };	instruction_memory[6] = { ANDI, 1'b0, R0, R2, 5'b00101 };
Opcode	**(0011)**: This indicates an ADDI (Add Immediate) operation	**(0100)**: This indicates an ANDI (AND Immediate) operation.
Mode	1'b0 (not used in this context for the ADDI operation)	1'b0 (not used in this context for the ANDI operation)
Destination Register	R0	R0
Source Register	R0	R2
Immediate Value	5 (binary: 00101)	5 (binary: 00101)
Execution Details	Since R0 is always 0, the addition result will be 5, but it won't be stored anywhere because R0 is hardwired to zero.	The value in R2 will be ANDed with the immediate value 5. The result will be attempted to be stored in R0.
Result	• R0 is hardwired to zero, so any attempt to write to R0 will be discarded. • The immediate value 5 will be added to the value in R0.	Since R0 is always 0 and cannot be written to, the result of the AND operation will be discarded.

4.2.2 Test load and store

```
instruction_memory[4] = { LW,1'b1 ,R1, R2, 5'b00001 };
instruction_memory[6]=  { SW,1'b1 ,R7, R2, 5'b0000};
instruction_memory[8]=  { LBs,1'b1 ,R1, R2, 5'b00010 };
instruction_memory[6]=  { SW,1'b1 ,R0, R2, 5'b00010 };
instruction_memory[10] = { LW,1'b1 ,R0, R1, 5'b00000 };
```

Figure 15: Load & store instructions test

4.2.2.1 LW (*Load Word)*

instruction_memory[4] = { LW,1'b1 ,R1, R2, 5'b00001 };

Immediate offset is 1.

Effective memory address= 2(value in R2) + 1 (immediate) = 3.

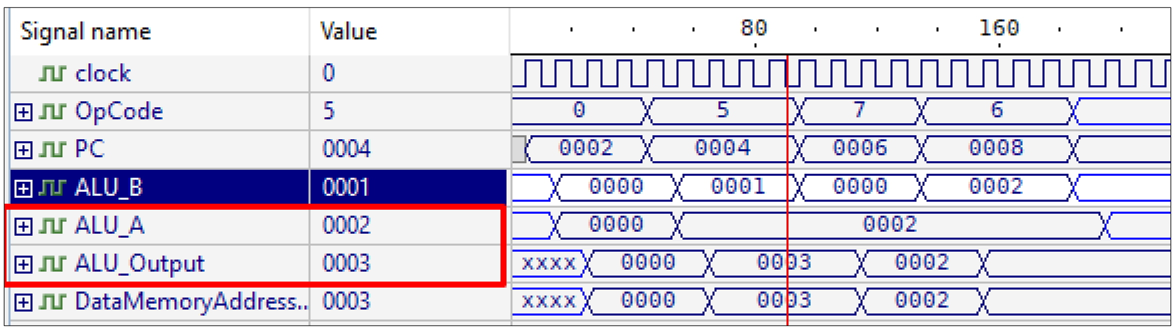

Figure 16: Simulation for LW

4.2.2.2 SW (*Store Word)*

instruction_memory[6]= { SW,1'b1 ,R7, R2, 5'b0000};

Effective memory address= 2(value in R2) + 0(immediate) = 2.

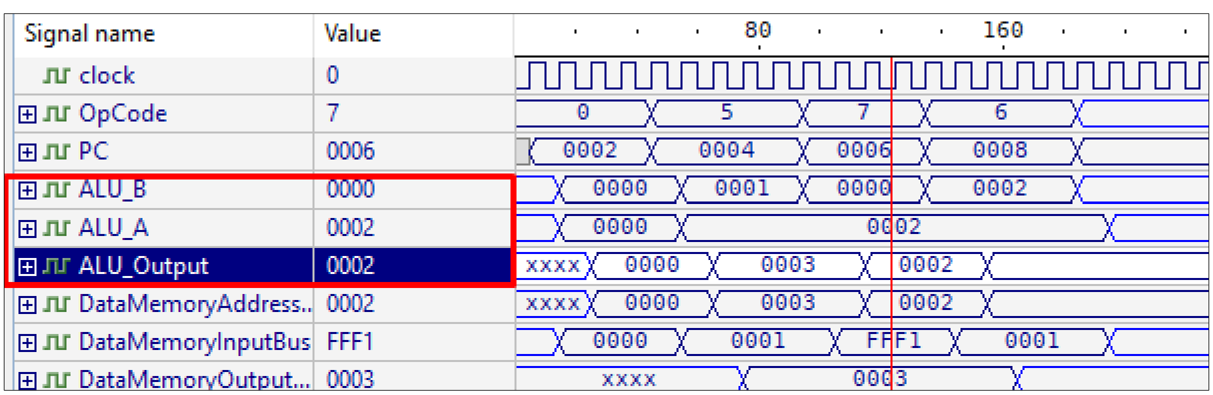

Figure 17: Simulation for SW

4.2.2.3 LBs (Load Byte Signed)

instruction_memory[8]= { LBs,1'b1 ,R1, R2, 5'b00010 };

R2 (registers_array[2]) contains 16'h0002.

Immediate offset is 2.

Effective memory address: 2 (value in R2) + 2 (immediate) = 4.

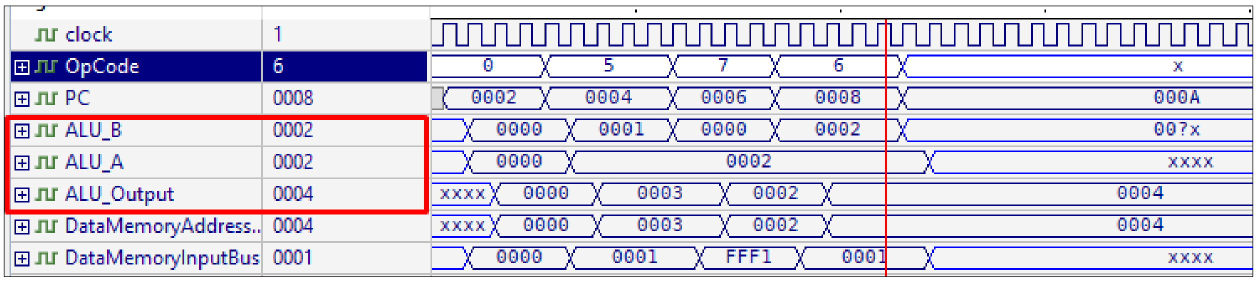

Figure 18: Simulation for LBs

4.2.2.4 LBu (Load Byte Unsigned)

instruction_memory[10] = { LBu,1'b1 ,R0, R1, 5'b00010 };

R1 (registers_array[2]) contains 16'h0001.

Immediate offset is 2.

Effective memory address: 1 (value in R1) + 2 (immediate) = 3.

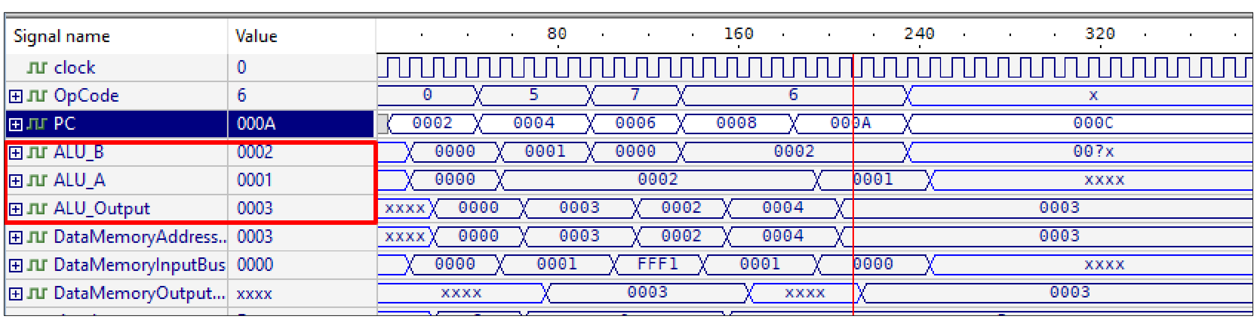

Figure 19: Simulation for LBu

4.2.3 Branch test

```
// --------Branch Test-------------------//


//BGT,BGTZ  ZERO & NEGATVE =0
//BLT,BLTZ NEGATIVE =1
//BEQ,BEQZ ZERO =1
//BNE.BNEZ ZERO=0
instruction_memory[4] = { BLT,1'b0 ,R2, R3, 5'b00001 };//Negative flag = 1
instruction_memory[5] = { BLTZ,1'b1 ,R7, R0, 5'b0001 };//Negative flag = 1
instruction_memory[6] = { BGT,1'b0 ,R2, R1, 5'b00001 }; //Negative flag = 0 zero flag =0
instruction_memory[7] = { BGTZ,1'b1 ,R6, R0, 5'b00010 }; //Negative flag = 0 zero flag =0
instruction_memory[9] = { BEQ,1'b0 ,R1, R1, 5'b00011 };
instruction_memory[12] = { BEQZ,1'b1 ,R5, R1, 5'b00001 };
instruction_memory[13] = { BNE,1'b0 ,R1, R1, 5'b00001 };
instruction_memory[15] = { BNEZ,1'b1 ,R1, R0, 5'b11110 };
```

Figure 20: Branch instructions test

1. BLT:

BLT	I-Type	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001

Figure 21: BLT instruction

instruction_memory[4] = { BLT,1'b0 ,R2, R3, 5'b00001 };

Branch if Less Than

opcode=9

pc=4

Results: next pc = pc+ sin_ex = 4 + FFE1 = FFE5

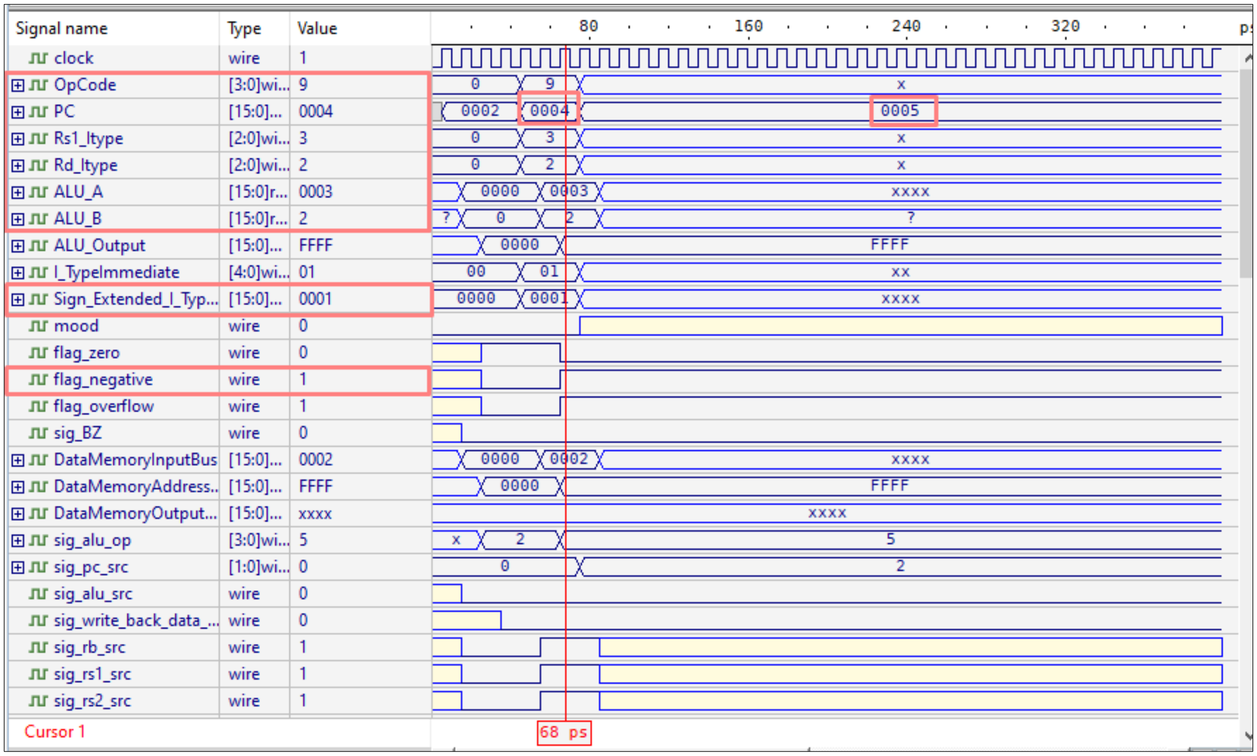

Figure 22: Simulation for BLT

2. BLTZ:

13	BLTZ	I-Type	if (Reg(Rd) < Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001

Figure 23: BLTZ instruction

instruction_memory[4] = { BLTZ,1'b1 ,R7, R0, 5'b0001 };

Branch if Less Than Zero.

OPCODE = 9

PC = 4

R7 = -15 , R0 =0

➔ **Result**: 4+1 = 5

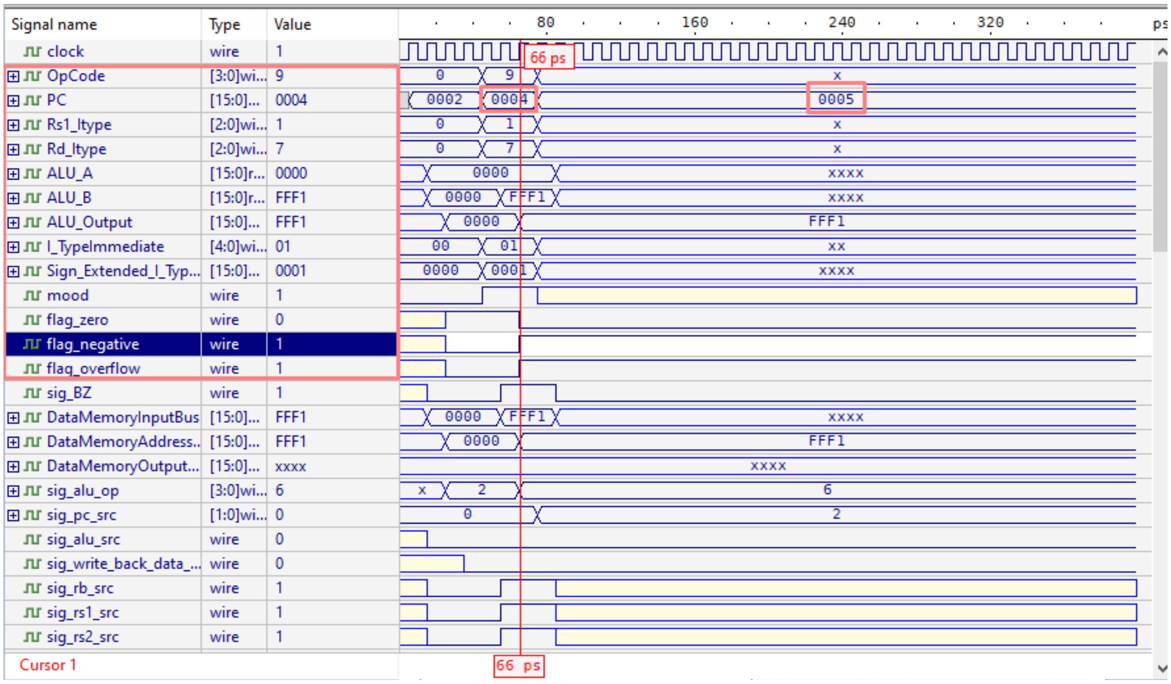

Figure 24: Simulation for BLTZ

Test else condithion:

instruction_memory[5] = { BLTZ,1'b1 ,R1, R7, 5'b0001 }

➔ Results :

pc = pc + 2 => 5+2 = 7

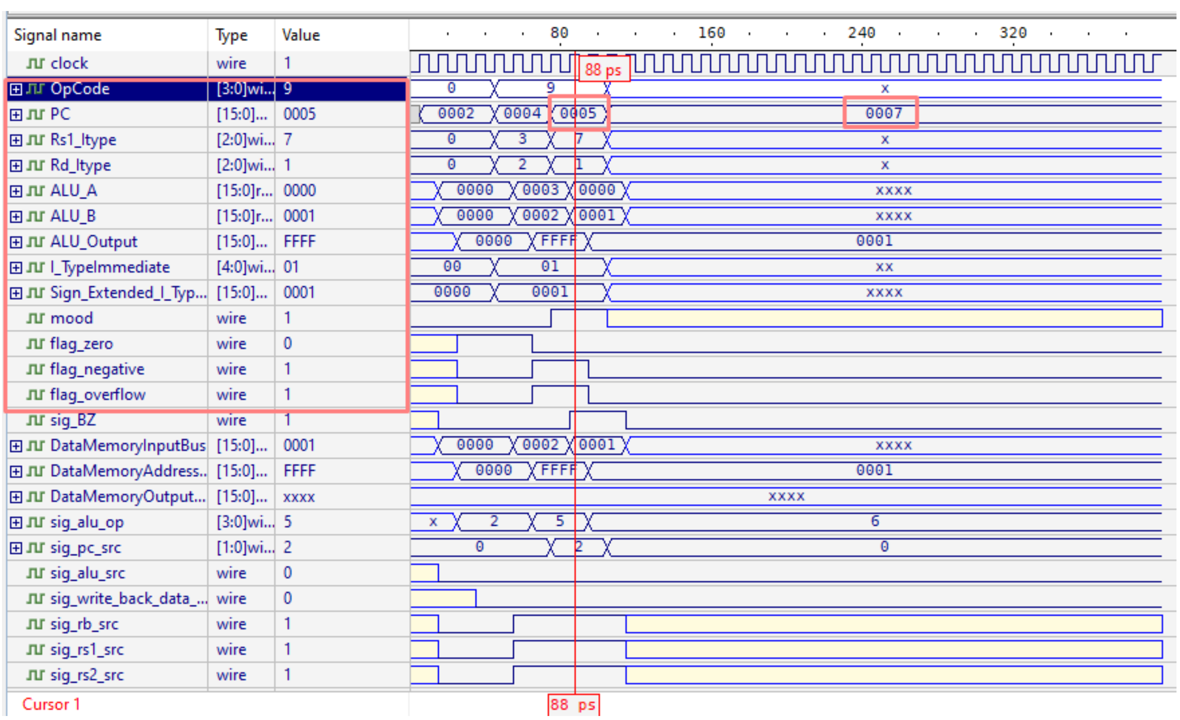

Figure 25: Simulation for BLTZ (2)

3. BGT

In this part we tested Branch if Greater Than and Branch if Greater Than Zero.

BGT	I-Type	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000
BGTZ	I-Type	if (Reg(Rd) > Reg(0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000

Figure 26: BGZ & BGTZ instructions

a. instruction_memory[4] = { BGT,1'b0 ,R2, R1, 5'b00001 };

- ✓ The BGT instruction checks if the value in R2 is greater than the value in R1.
- ✓ Given R2 = 2 and R1 = 1, the condition is true, so the branch is taken.
- ✓ The next PC is calculated by adding the immediate value (1) to the current PC (4), resulting in the next PC being 5.

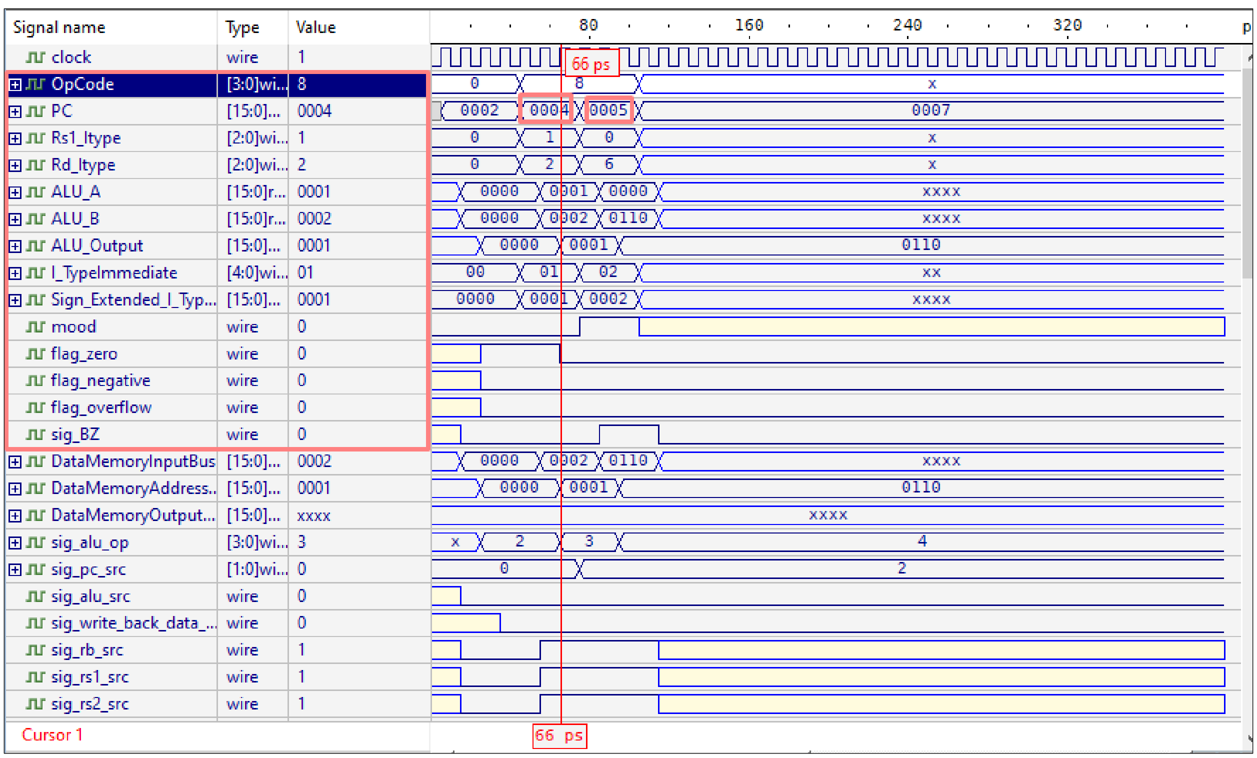

Figure 27: Simulation for BGT

b. instruction_memory[5] = { BGTZ,1'b1 ,R7, R0, 5'b00010 };

- ✓ The BGTZ instruction checks if the value in R7 is greater than zero.
- ✓ Given R7 = -15 and R0 = 0, the condition is false, so the branch is not taken.
- ✓ Therefore, the PC is incremented by 2, (Initial PC: 5) , resulting in the next PC being 7.

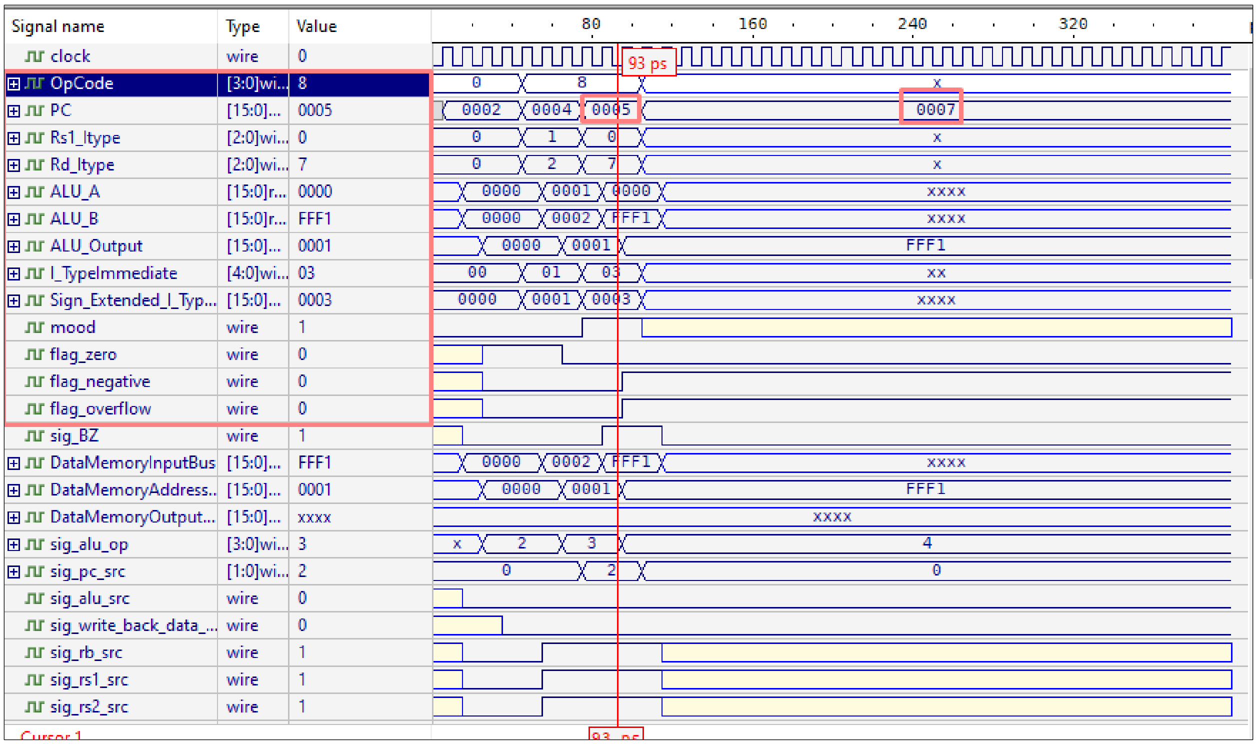

Figure 28: Simulation for BGTZ

4. BEQ & BEQZ

In this part : we was tested Branch if Equal and Branch if Equal to Zero

BEQ	I-Type	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010
BEQZ	I-Type	if (Reg(Rd) == Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010

Figure 29: BEQ & BEQZ instructions

a. instruction_memory[4] = { BEQ,1'b0 ,R1, R1, 5'b00011 };

- ✓ The BEQ instruction checks if the value in R1 is equal to the value in R1.
- ✓ Since R1 is always equal to R1, the condition is true, so the branch is taken.
- ✓ Initial PC: 4 , Immediate Value: 5'b00011 (3 in decimal)
- ✓ Branch Target Address: PC + sign_extended (Immediate) = 4 + 3 = 7

➔ Therefore, the next instruction executed will be at PC = 7.

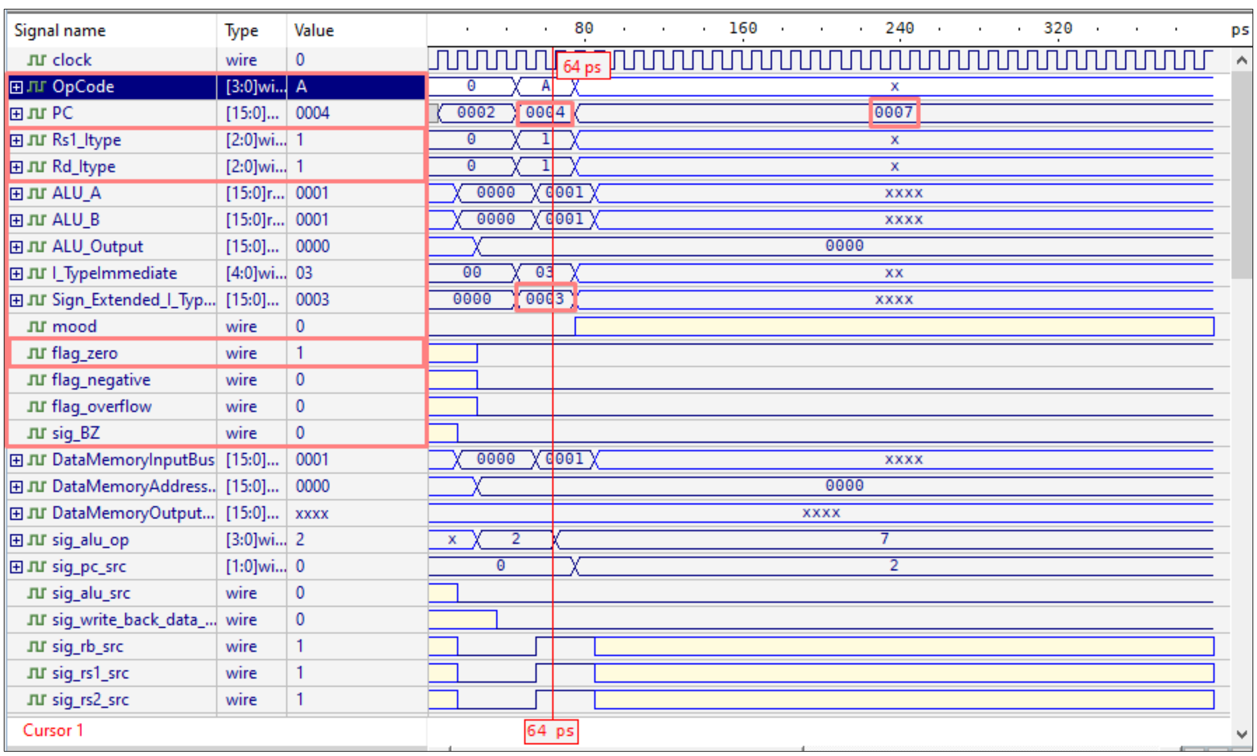

Figure 30: Simulation for BEQ

b. instruction_memory[4] = { BEQZ,1'b1 ,R5, R1, 5'b00001 };

- ✓ The BEQZ instruction checks if the value in R5 is equal to zero.
- ✓ If R5 is zero, the branch is taken, and the next PC is PC + Immediate (5).
- ✓ If R5 is not zero, the branch is not taken, and the next PC is PC + 2 (6).
- ✓ Since R5 is not zero, the branch is not taken.
- ✓ Initial PC: 4
- ✓ Next PC: PC + 2 = 4 + 2 = 6

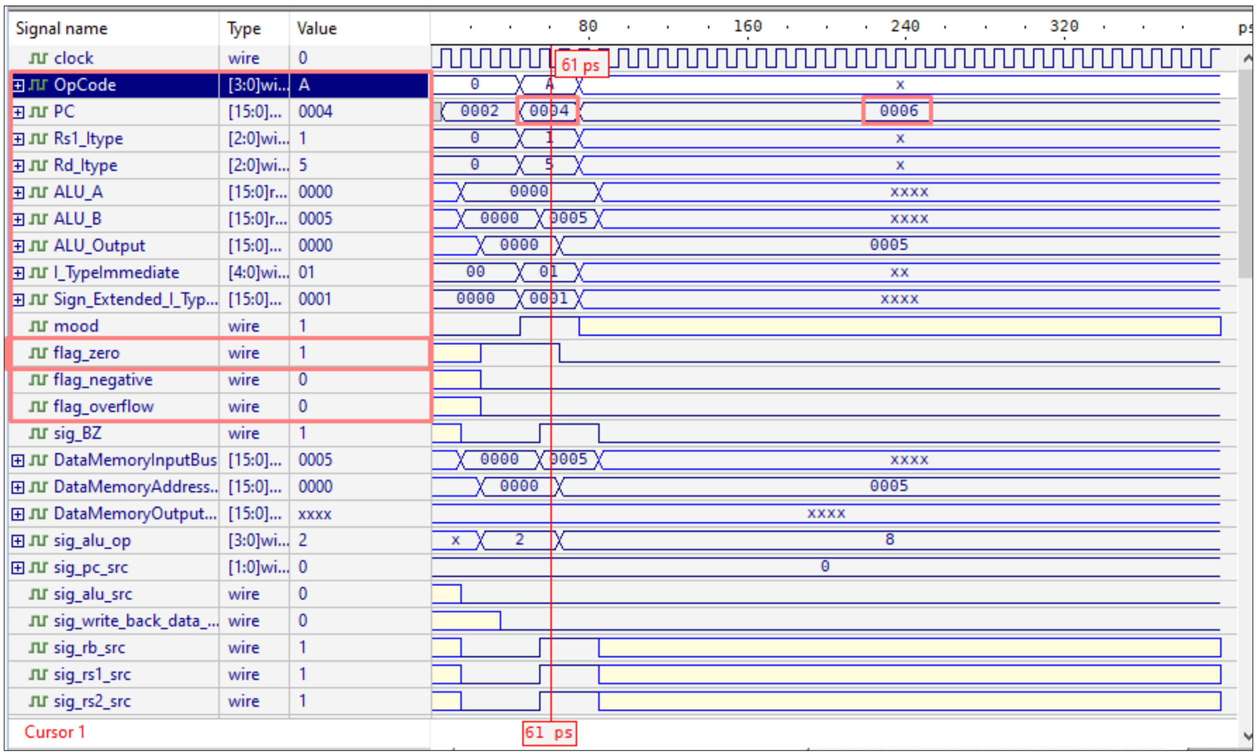

Figure 31: Simulation for BEQZ

5. BNE & BNEZ

In this part we was tested **BNE** (Branch if Not Equal).

BNE	I-Type	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011
BNEZ	I-Type	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011

Figure 32: BNE instruction

a. instruction_memory[4] = { BNE,1'b0 ,R1, R1, 5'b00001 };

The BNE instruction checks if the value in R1 is not equal to the value in R1.

Since R1 is always equal to R1, the condition is false, so the branch is not taken.

The next PC is calculated by adding 2 to the current PC (4), resulting in the next PC being 6.

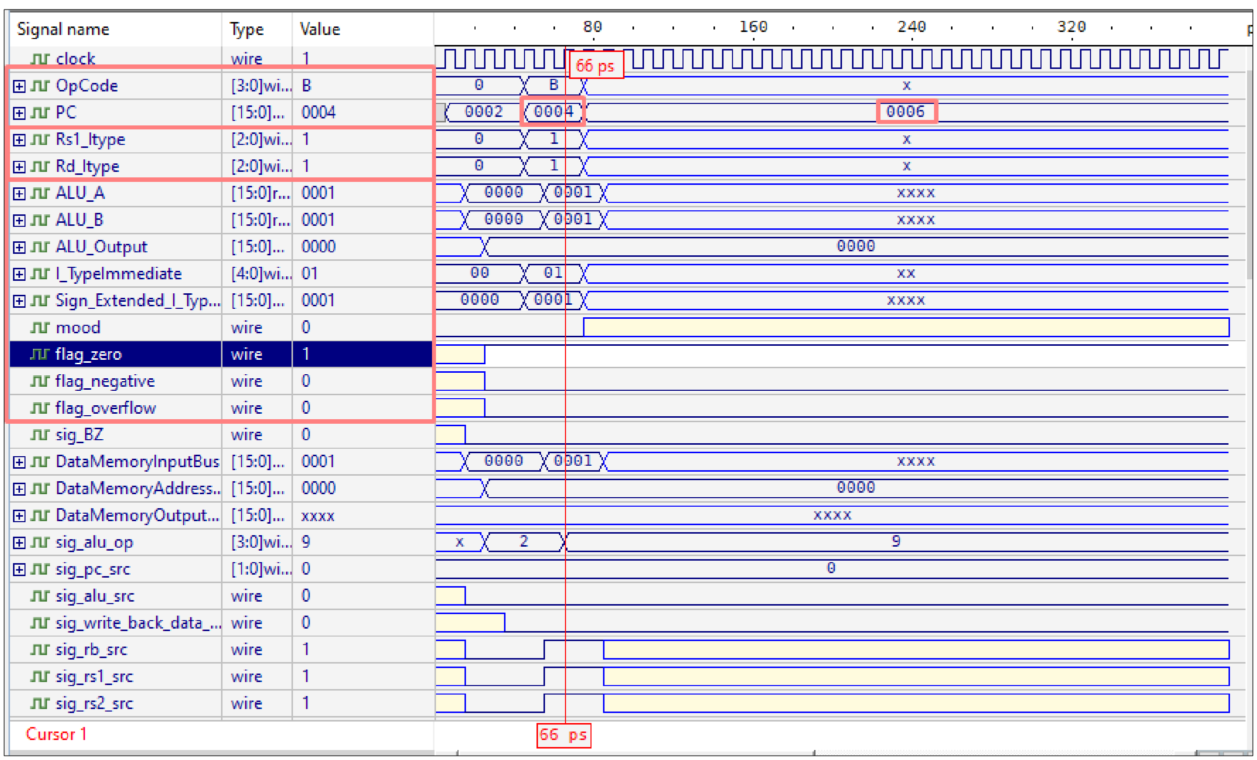

Figure 33: Simulation for BNE

b. instruction_memory[4] = { BNE,1'b0 ,R2, R1, 5'b00001 };

- ✓ The BNE instruction checks if the value in R2 is not equal to the value in R1.
- ✓ Since R2 (2) is not equal to R1 (1), the condition is true, so the branch is taken.
- ✓ The next PC is calculated by adding the sign-extended immediate value (1) to the current PC (4), resulting in the next PC being 5.

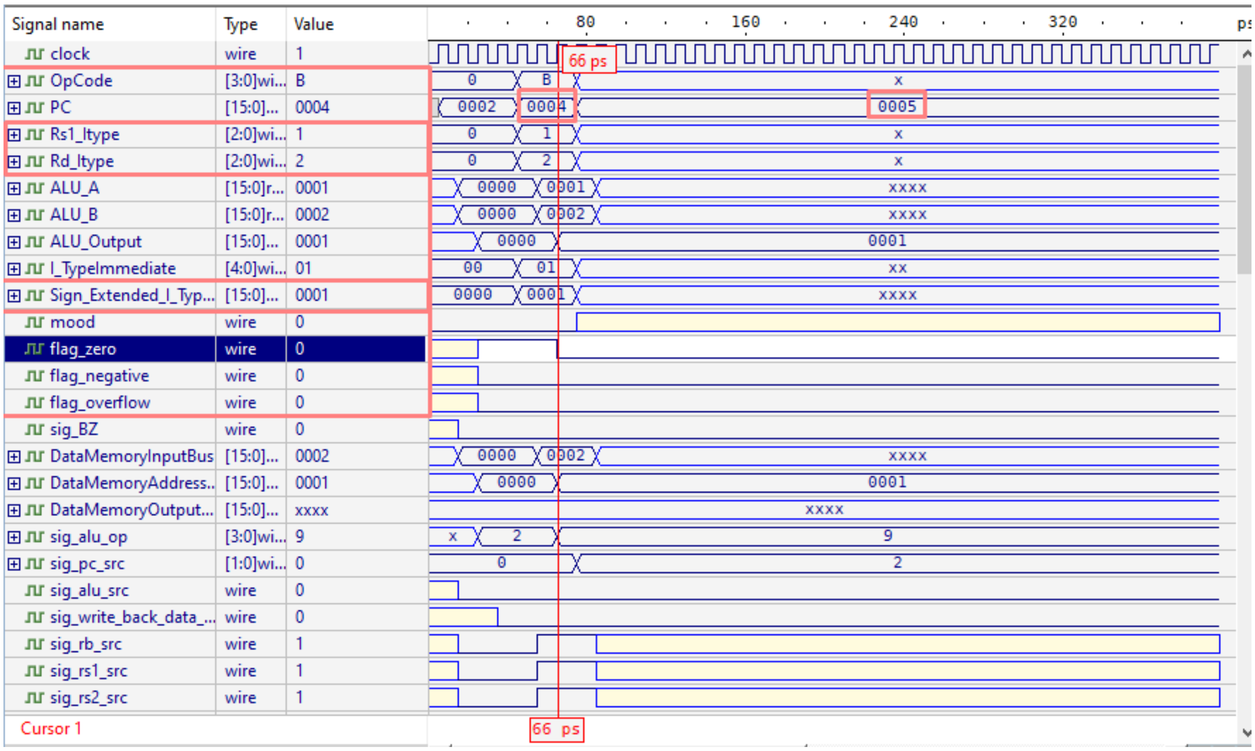

Figure 34: simulation for BNE.

c. instruction_memory[4] = { BNEZ,1'b1 ,R0, R1, 5'b11110 };

- ✓ The BNEZ instruction checks if the value in R0 is not equal to zero.
- ✓ Since R0 is zero, the condition is false, so the branch is not taken.
- ✓ The PC increments by 2 to skip the next instruction, resulting in the next PC being 6.

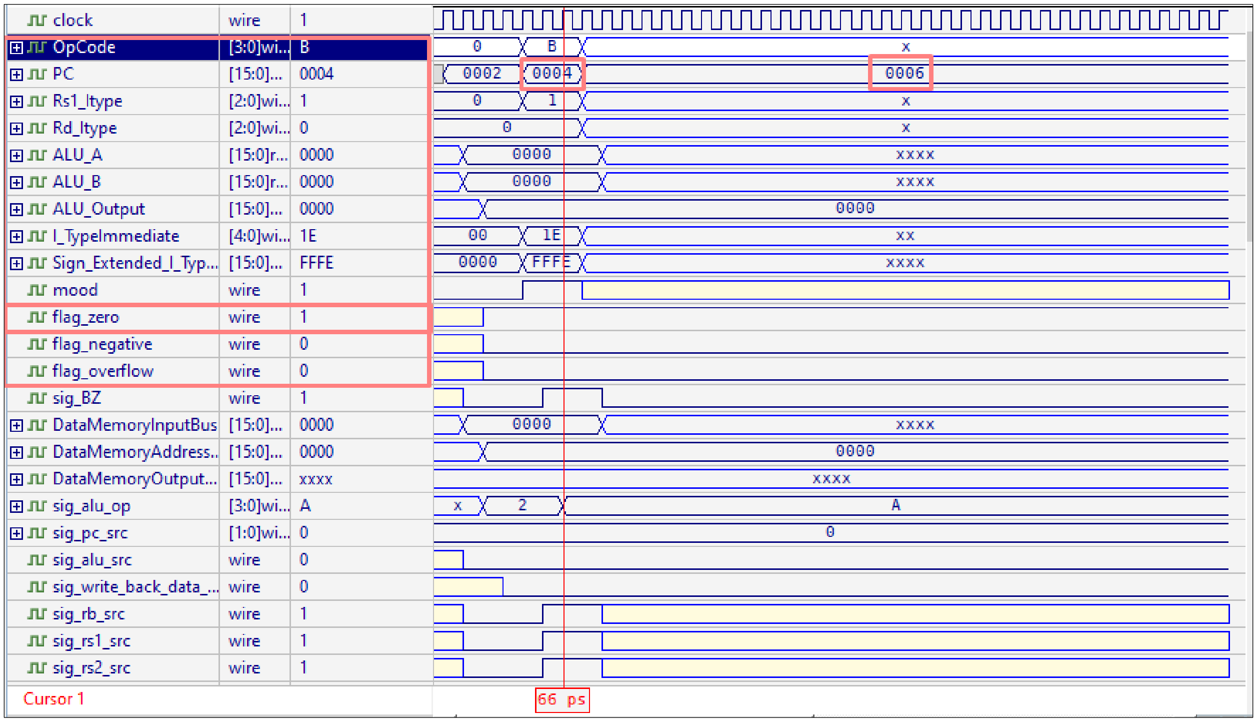

Figure 35: BNEZ simulation

4.3 J-type instruction test

```
// -----------------------j-type test---------------------------------------//

instruction_memory[4]= { JMP ,12'b000001100};
instruction_memory[12]= { CALL ,12'b000001111};
instruction_memory[14] = { LW,1'b0 ,R1, R4, 5'b00001 };
instruction_memory[15]= { RET ,12'b000001111};
```

Figure 36: j-type instructions test

1. JUM

The JMP instruction is a jump (unconditional) instruction in the processor's instruction set architecture. It directs the processor to unconditionally jump to a new target address calculated based on the current program counter (PC) and an immediate value.

JMP	J-Type	Next PC = {PC[15:10], Immediate}	1100 C

Figure 37: JMP instruction

- instruction_memory [4]= { JMP ,12'b000001100};
- **Initial PC**: 4
- **Immediate Value**: 000001100 (12 in decimal) This value is calculated by concatenating the most significant 7 bits of the current PC with the 12-bit immediate value after adjusting for byte addressing.

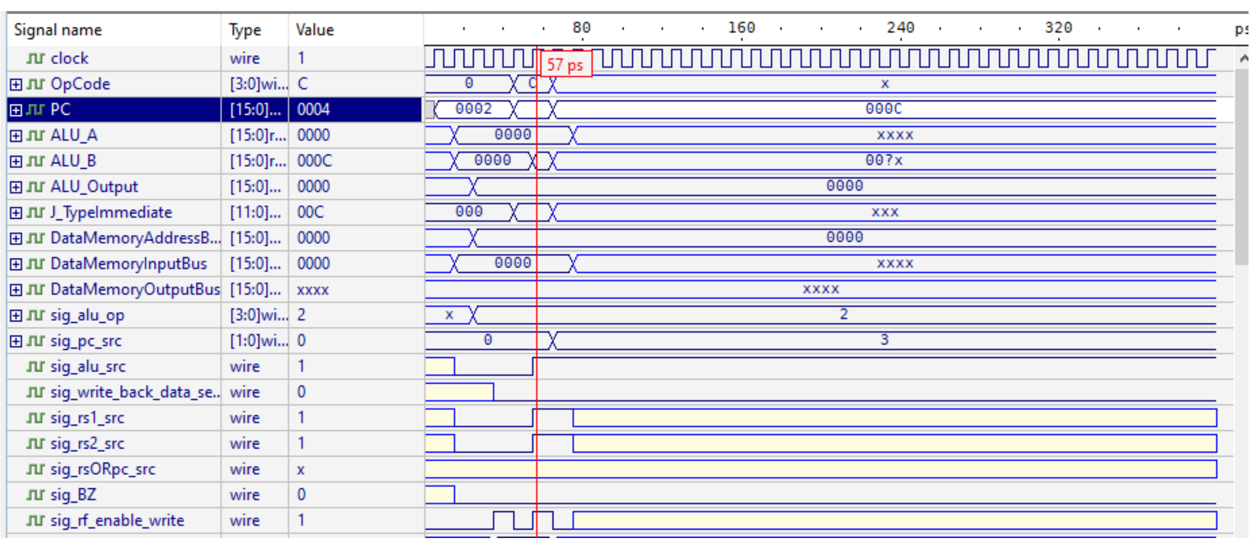

Figure 38: JMP instruction simulation

2. CALL

In this section we test the call function instruction:-

CALL	J-Type	Next PC = {PC[15:10], Immediate} PC + 4 is saved on r15	1101 D

Figure 39: CALL instruction

- instruction_memory[12] = { CALL ,12'b000001111};
- PC = 12 = C
- OPCODE = D
- The CALL instruction provides a mechanism for executing subroutines or functions in the processor. By saving the return address onto R15 and jumping to a specified target address.

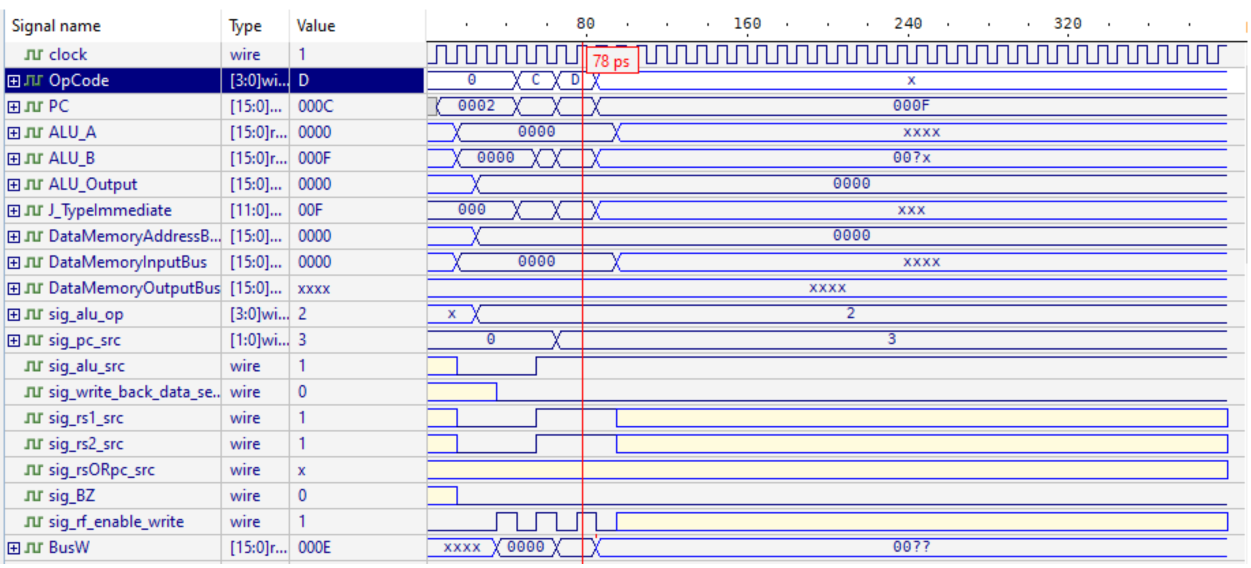

Figure 40: CALL simulation results

3. RES

In this section we test the Return from Function instruction:-

RET	J-Type	Next PC = r7	1110 E

Figure 41: RES instruction

instruction_memory[15]= { RET ,12'b000001111};

PC = 15 = F / Opcode: 12'b000001111

registers_array[7] <= -16'h000F; R7 = -15.

- **Behavior**: Next PC = R7 (PC is set to the value stored in register `R7`)

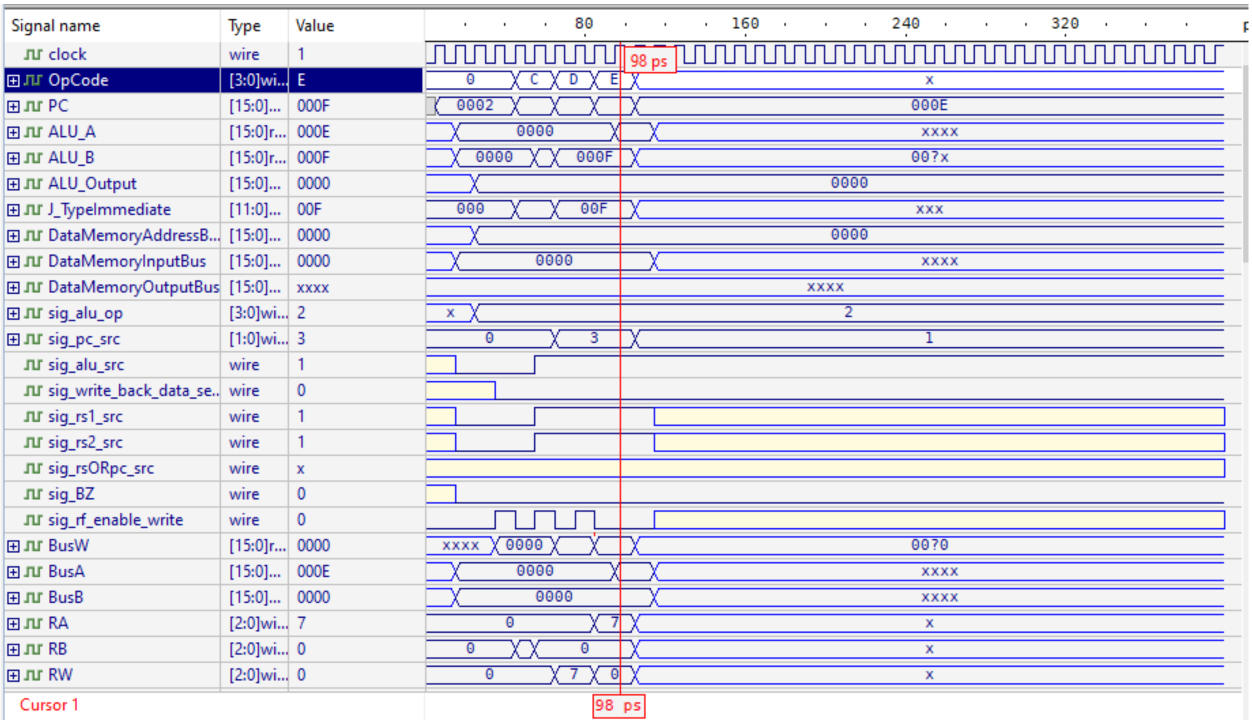

Figure 42: RES similation results

Test load results after return from runction:

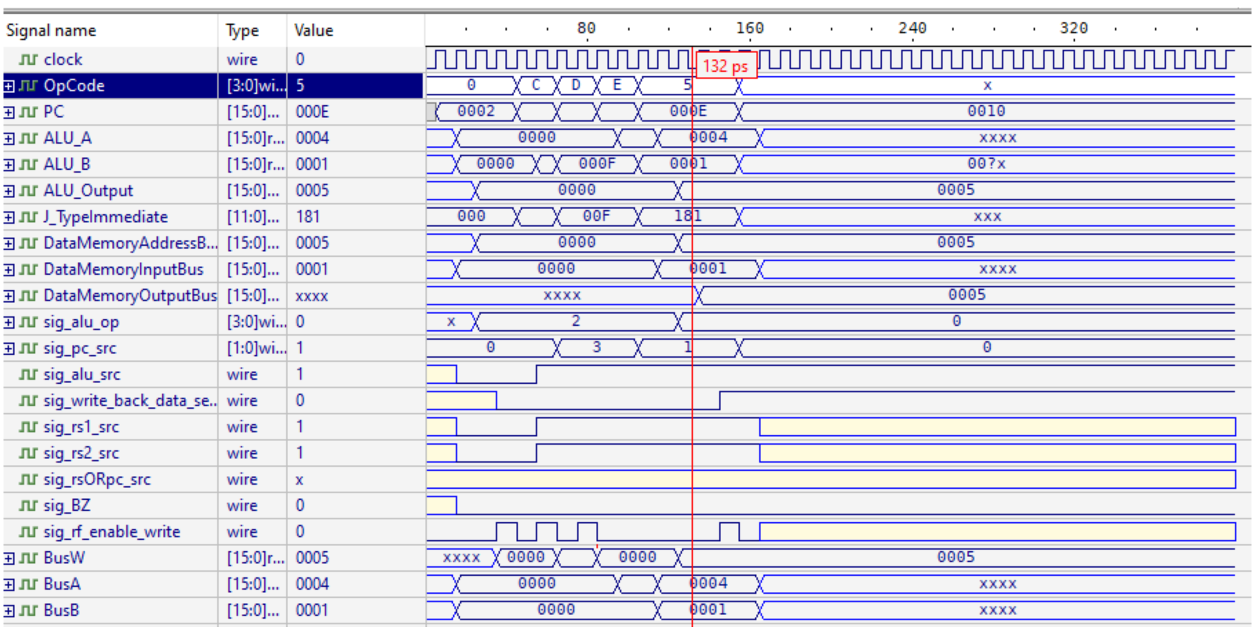

Figure 43: RES similation results

4.4 S-type instruction test

Sv	S-Type	M[rs] = imm	1111 F

Figure 44: Sv instruction

```
        //------------------S-type test---------------------------//

instruction_memory[4]=  { Sv ,R5, 9'b000000011};
instruction_memory[6] = { LW,1'b0 ,R1, R4, 5'b00001 }; // check value BU
```

Figure 45: s-type test

- instruction_memory[4]= { Sv ,R5, 9'b000000011};
- Opcode (1111): F
- Meaning: The Sv instruction stores a value (imm) into memory at the address specified by the contents of register R5 (M[rs] = imm).
- Immediate (9'b000000011): This represents an immediate value of 3 in decimal, used as the data to be stored in memory.
- PC (Program Counter): 4
- R5 = 5

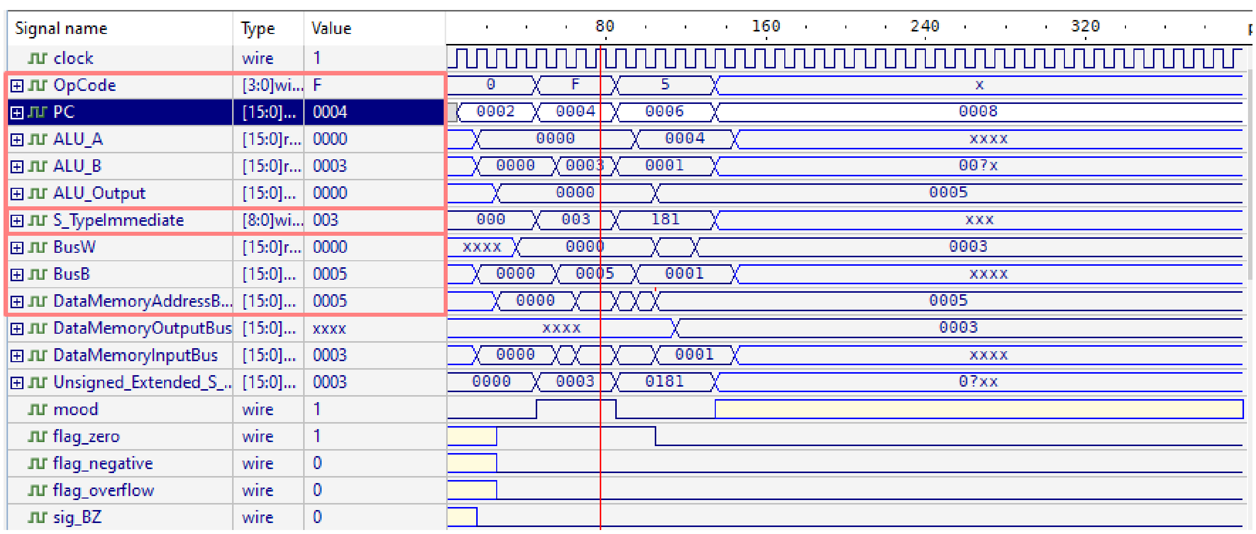

Figure 46: Sv simulation results

✓ The value of BUSW Or value Data output = 5

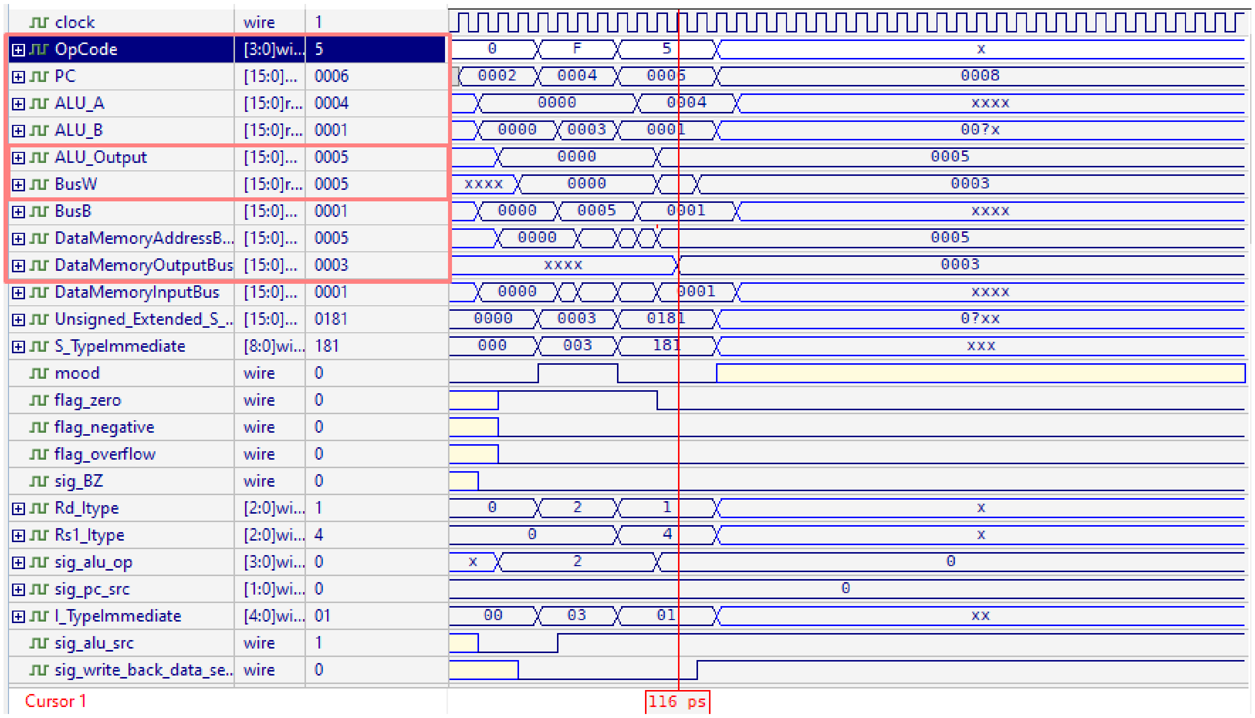

Figure 47: Sv simulation results

Conclusion

In conclusion, the design and verification of a multi-cycle RISC processor using Verilog has provided valuable insights into the principles of computer architecture and the practical implementation of digital logic. By adopting a multi-cycle execution model, the processor architecture was able to optimize resource utilization and simplify the control logic, while still supporting a comprehensive instruction set including register, immediate, jump, and store operations. The rigorous verification process, involving thorough testbenches and code sequences, has ensured the accurate functionality of the processor design, validating its adherence to the specified Instruction Set Architecture. This project has successfully demonstrated the application of digital logic and computer organization concepts within a Verilog-based simulation environment, offering a strong foundation for further exploration of processor design and hardware description languages, and equipping the designer with a deeper understanding of the fundamental building blocks and design considerations that go into the development of computer processors.

References

1) https://github.com/ibraheemalayan/Verilog_RISC_Processor/tree/main

2) https://www.youtube.com/playlist?list=PLTbIxs_QUEveo7eSqDHtFxef3Kx7lUlv