Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Computer Architecture – ENCS4370

Project #2: Design and Implementation of Multi-Cycle Processor

---

Prepared by:

Ahmad Bakri 1201509

Omar Masalmah 1200060

Batool Hammouda 1202874

Date:

20-6-2024

Abstract:

This report shows the steps of design and implement a multicycle processor for a given instructions, and with predefined specifications for the processor, starting from analyze the ISA, then designing the Datapath and building each unit in it including: Instructions Register, Instruction Memory, Register File, Data Memory, Control Units and Multiplexers. And finally connect the components together to have a full Datapath for a RISC Processor with 5 stages: Instruction Fetch, Instruction Decode, Execution, Memory and Write Back.

Table of Contents

Table of figures:

Introduction:

Designing, simulating, and modeling a MIPS multi-cycle processor with Verilog HDL is the goal of this project. The method is breaking down the processor into smaller components, each of which is created, implemented, and verified independently. After completion of their functional verification, these sub-modules are combined with a structural module to form the fully functional processor.

RISC Overview:

Reduced Instruction Set Computing (RISC) puts more emphasis on improving CPU performance by using a more condensed instruction set. Because of their one-cycle execution, identical registers, fixed-length instructions, easy addressing, and speed, RISC processors are extensively used in industry applications.

Multi-Cycle Processors:

Several stages of instruction execution are achieved by multi-cycle processors, which facilitate the effective processing of intricate instructions. They allow different instruction completion times, which improves overall performance and supports different types of instructions, in contrast to their single-cycle counterparts.

Within a single clock cycle, every instruction in a multi-cycle processor goes through multiple stages. Among these stages are:

1. Instruction Fetch (IF): Using the program counter (PC) to retrieve the instruction from memory and update it in preparation for the next instruction.
2. Instruction Decode (ID): identifying the operation by decoding the fetched instruction, and retrieving any required operands or data from registers.
3. Execution (EX): carrying out the real operation that the instruction specifies; this could entail address calculations, logical operations, or arithmetic calculations.
4. Memory Access (MEM): if the instruction calls for it, accessing memory to load or store data. Reading from or writing to memory is required at this level.
5. Write Back (WB): which involves writing the outcomes of the previous stage back to the relevant register or registers and updating the register file with the calculated values.

Processor Specifications:

- **Instruction Size and Word Size**: Both are 16 bits.
- **General-Purpose Registers**: Eight 16-bit registers (R0 to R7).
  - **Note**: R0 is hardwired to zero; any write attempts to R0 will be ignored.
- **Program Counter (PC)**: A 16-bit special purpose register.
- **Instruction Types**: Four types of instructions are supported:
  - R-type
  - I-type
  - J-type
  - S-type
- **Memory Architecture**:
  - Separate memories for data and instructions.
  - Byte-addressable memory.
- **Byte Ordering**: Little-endian.
- **ALU Signals**: The ALU generates necessary signals to determine the outcome of conditional branches (taken/not taken). These signals include:
  - Zero
  - Carry
  - Overflow
  - Etc.

Instruction Types and Formats:

R-Type (Register Type):

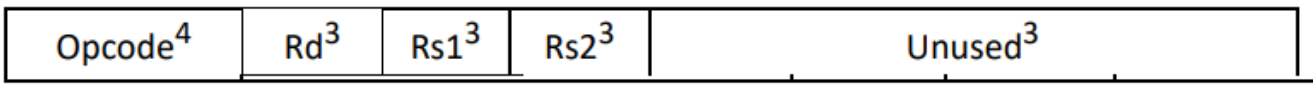


Figure 1:R-Type

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

I-Type (Immediate Type):

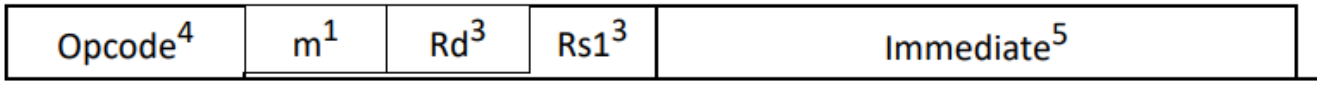


Figure 2: I_Type

• 3-bit Rd: destination register

• 3-bit Rs1: first source register

• 5-bit immediate: unsigned for logic instructions, and signed otherwise.

• 1-bit mode: this is used with load and branch instructions, such that:

For the load: 0: LBs load byte with zero extension 1: LBu load byte with sign extension.

For the branch: 0: compare Rd with Rs1 1: compare Rd with R.

J-Type (Jump Type):

- Jump and call :

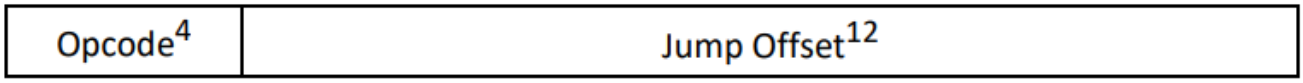
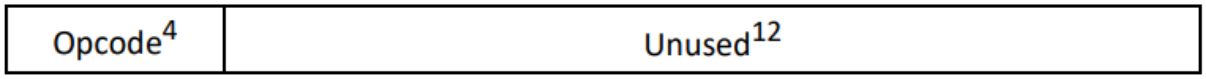


Figure 3: jump format

The target address is calculated by concatenating the most significant 7-bit of the current PC with the 12-bit offset after multiplying offset by 2.

- Ret instruction:



S-Type (Store):

Figure 4:S-Type

Sv rs, imm # M[rs] = imm

Instructions' Encoding:

No.	Instr	Format	Meaning	Opcode Value	m
1	AND	R-Type	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0000	
2	ADD	R-Type	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	0001	
3	SUB	R-Type	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	0010	
4	ADDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	0011	
5	ANDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	0100	
6	LW	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0101	
7	LBu	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	0
8	LBs	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	1
9	SW	I-Type	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111	
10	BGT	I-Type	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	0
11	BGTZ	I-Type	if (Reg(Rd) > Reg(0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	1
12	BLT	I-Type	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001	0
13	BLTZ	I-Type	if (Reg(Rd) < Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001	1
14	BEQ	I-Type	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	0
15	BEQZ	I-Type	if (Reg(Rd) == Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	1
16	BNE	I-Type	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011	0
17	BNEZ	I-Type	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011	1
18	JMP	J-Type	Next PC = {PC[15:10], Immediate}	1100	
19	CALL	J-Type	Next PC = {PC[15:10], Immediate} PC + 4 is saved on r15	1101	
20	RET	J-Type	Next PC = r7	1110	
21	Sv	S-Type	M[rs] = imm	1111	

Figure 5: Instructions

Designing the Datapath:

RTL for the Instructions:

- For R_Type Instructions:

IR ←Mem [PC]

Datal← Reg [Rs1]

Data2 ←Reg [Rs2]

Alu_res ←funct (datal, data2)

Reg [Rd] ← alu_res

PC PC+2

- For I_Type: (ANDI, ADDI):

IR ← MEM[PC]

Datal← Reg (Rs1)

Data2← Extend(imm)

ALU_result← op (data1, data2)

Reg (Rd) ← ALU_result

PC PC + 2

- For LW, LBu, LBs:

For LBu and LBs the opcode is the same, but the bit m in the first is 0 (unsigned extension) and 1 in the last (signed extension).

IR MEM[PC]

Base← Reg (Rs1)

Addres ← base+ extend(imm)

Data← MEM [address]

Reg (Rd) ← data

PC ← PC+2

- For SW:

IR ← MEM[PC]

Base← Reg (Rs1)

Address← base + sign_extend(imm)

MEM [address] ← Reg (Rd)

PC← PC +2

- For BGT, BGTZ:

In BGTZ Rs1 is R0.

IR← MEM[PC]

datal Reg (Rs1), data2 ← Reg (Rd)

zero subtract (datal, data2)

if (N! = V): PC← PC + sign_ext(offset)

else PC← PC + 2

- For BEQ and BEQZ

In BEQZ Rs1 is R0.

IR ← MEM[PC]

datal← Reg (Rs1), data2← Reg (Rd)

zero← subtract (datal, data2)

if (zero): PC← PC + sign_ext(offset)

else PC← PC+2

- For BLT, BLTZ:

In BLTZ Rs1 is R0.

IR← MEM[PC]

datal← Reg (Rs1), data2← Reg (Rd)

zero← subtract (datal, data2)

if (! Z && N == V): PC ← PC + sign_ext(offset)

else PC← PC +2

- For BNE, BNEZ:

In BNEZ Rs1 is R0

IR← MEM[PC]

datal← Reg (Rs1), data2 ← Reg (Rd)

zero← subtract (datal, data2)

if (! zero): PC← PC+sign_ext(offset)

else PC← PC +2

- For J_Type (JMP):

Instruction ← MEM[PC]

target← PC [15:10] || ext(imm)

PC ← target

- For CALL:

Instruction ← MEM[PC]

target← PC [15:10] || ext(imm)

PC ← target

R7← PC+2

- For RET Instruction:

Instruction ← MEM[PC]

PC← Reg(R7)

- For SV:

Instruction ← MEM[PC]

datal← Reg (Rs1)

MEM(Rs1) ←ext(imm)

PC← PC +2

Functional Units in the Datapath:

1- Instruction memory: It plays a crucial part in program execution and overall system performance by storing the machine instructions that the CPU retrieves and performs. It is byte addressable.

2- Register File: with 8 16-bit general-purpose registers: from R0 to R7.

3- ALU: It performs arithmetic and logical operations, such as addition, subtraction, AND, OR, and more, essential for executing instructions and processing data within the computer.

4- Data memory: includes the data that needed to perform some operation on it.

5- Extender: The extender is a component that is used to expand the width of the data to be 16 bits, and it can support both signed and unsigned data representations.

6- IR: maintains the execution of the active instruction, guaranteeing the right order of steps. It is essential to the fetch-decode-execute cycle that the CPU of a computer goes through.

The reason behind separating the Memory:

The implementation's memories are divided into two different types: data memory and instruction memory. In order to comply with the isolation principle, they must be divided into distinct memory elements. For example, one instruction may be retrieving an instruction from memory while another is loading or storing data. These conflicts were resolved by doing this.

Code and Waveform for the Instruction Memory:

```
module instruction_memory(
    input [15:0] addr,
    output reg [15:0] data_out
);
    // Define memory
    reg [15:0] mem [0:1023];

    // Read operation
    always @(addr) begin
        data_out <= mem[addr];
    end

    // Initialize the memory with instructions
    initial begin
        // R type
        mem[0]  = 16'b0000001010011000; // AND R1, R2, R3
        mem[1]  = 16'b0001101010100000; // ADD R5, R2, R4
        mem[2]  = 16'b0010011101001000; // SUB R3, R5, R1
        // I type
        mem[3]  = 16'b0011011001011010; // ADDI R6, R2, 26
        mem[4]  = 16'b0100000111001101; // ANDI R1, R6, 13
        mem[5]  = 16'b0101001010011010; // LW R2, R4, 26
        mem[6]  = 16'b0110001000110100; // LBu R2, R1, 20
        mem[7]  = 16'b0110101000110100; // LBS R2, R1, 20
    /*    mem[8]  = 4'b0111; // SW I-Type
        mem[9]  = 4'b1000; // BGT I-Type
        mem[10] = 4'b1010; // BGTZ I-Type
        mem[11] = 4'b1011; // BLT I-Type
        mem[12] = 4'b1100; // BLTZ I-Type
        mem[13] = 4'b1101; // BEQ I-Type
        mem[14] = 4'b1110; // BEQZ I-Type
        mem[15] = 4'b1111; // BNE I-Type
        mem[16] = 4'b0000; // BNEZ I-Type
        mem[17] = 4'b0001; // JMP J-Type
        mem[18] = 4'b0010; // CALL J-Type
        mem[19] = 4'b0011; // RET J-Type
        mem[20] = 4'b0100; // SV S-Type
   */
    end
endmodule

```

Figure 6:Instruction Memory code

Memory Instruction TestBench:

```
module instruction_memory_TB;
    reg [15:0] addr;
    wire [15:0] data_out;

    instruction_memory mem(addr, data_out);

    initial begin
        addr = 0;
        #10;
        $display("Instruction = %h at Address: %h", data_out, addr);
        #10;
        addr = 1;
        #10;
         $display("Instruction = %h at Address: %h", data_out, addr);
        #10;
        addr = 2;
        #10;
         $display("Instruction = %h at Address: %h", data_out, addr);
        #10;
/*      addr = 3;
        #10;
        $display("Instruction = %h", data_out);
        #10;
        addr = 4;
        #10;
        $display("Instruction = %h", data_out);
        #10;
        addr = 5;
        #10;
        $display("Instruction = %h", data_out);
        #10;
        */
        $finish;
    end
endmodule
```

Figure 7:Memory Instruction TB

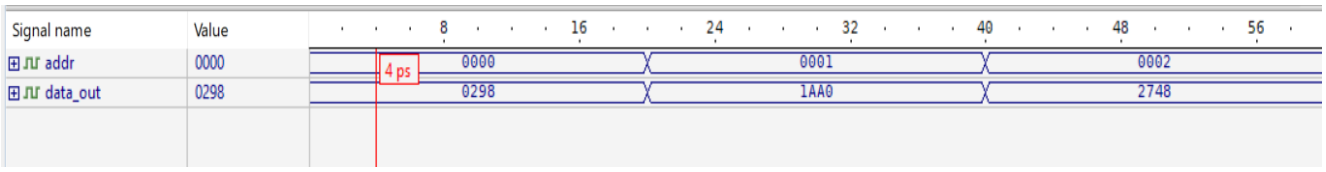


Figure 8:MI_WaveForm

Code and Waveform for Data Memory:

```
module memory (
    input clk,
    input  [15:0] addr,
    input  [15:0] data_in,
    input  write_signal,
    input  read_signal,
    output reg [15:0] data_out
);
    reg [15:0] mem [0:1023]; // // Define a memory array assuming 1024 16-bit words in memory

    always @(posedge clk or read_signal or write_signal) begin
        if (write_signal) begin
            mem[addr] <= data_in;
        end
        else if (read_signal) begin
            data_out <= mem[addr];
        end
    end

    // Initialize memory with some random values
    initial begin
        integer i;
        for (i = 0; i < 1024; i = i + 1) begin
            mem[i] = i;
        end
    end
endmodule
```

Figure 9:Data Memory

```
module memory_TB;
    reg clk;
    reg [15:0] addr;
    reg [15:0] data_in;
    reg write_signal;
    reg read_signal;
    wire [15:0] data_out;

    memory dut (
        .clk(clk),
        .addr(addr),
        .data_in(data_in),
        .write_signal(write_signal),
        .read_signal(read_signal),
        .data_out(data_out)
    );

    // Clock generation
    always begin
        #5 clk = ~clk;
    end

    // Stimulus
    initial begin
        clk = 0;
        addr = 0;
        data_in = 0;
        write_signal = 0;
        read_signal = 0;
        #10;
```

```
        //  Read data from memory
        write_signal = 0;
        read_signal = 1;
        addr = 10;
        #20;
        if (data_out == 123) begin
            $display("Test Case 1 Passed: Read data 123 from address 10");
        end else begin
            $display("Test Case 1 Failed: Expected 123, got %d", data_out);
        end

        // Test Case 2: Write data to memory
        write_signal = 1;
        addr = 13;
        data_in = 536;
        #10;

        //  Read data from memory
        write_signal = 0;
        read_signal = 1;
        addr = 13;
        #20;
        if (data_out == 536) begin
            $display("Test Case 2 Passed: Read data 536 from address 13");
        end else begin
            $display("Test Case 2 Failed: Expected 536, got %d", data_out);
        end

        $finish;
    end

    // Monitor
    always @(posedge clk) begin
        $display("CLK = %d, addr = %d, data_in = %d, write_signal = %d, rea
                 clk, addr, data_in, write_signal, read_signal, data_out);
    end
endmodule
```

Figure 10:Data Memory_TB

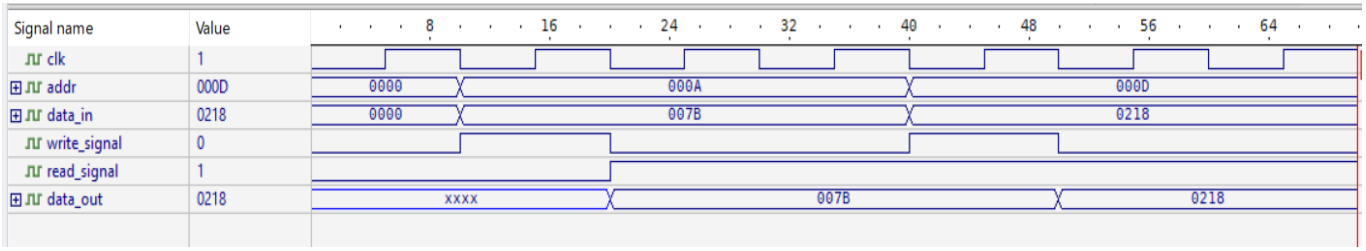


Figure 11:Data Memory Waveform

Register File code and Waveform:

It comprises three registers: RA, RB, and RW, each with a 3-bit width. These registers are connected to two primary data buses, BusA and BusB, both of which have a width of 16 bits. This architecture enables reading from and writing to these registers through BusW2 and BusW2 when their signal values one.

```
1  module registersFile(
2      input clk,
3      input [2:0] read_regA,
4      input [2:0] read_regB,
5      input [2:0] write_reg,
6      input [15:0] write_data,
7      input write_enable,
8      output reg [15:0] busA,
9      output reg [15:0] busB
10 );
11     reg [15:0] registers [0:15];
12
13     always @(posedge clk) begin
14         if (write_enable) begin
15             registers[write_reg] <= write_data;
16         end
17         busA = registers[read_regA];
18         busB = registers[read_regB];
19     end
20
21     // Initialize register to be random
22     initial begin
23         registers[0] = 16'h0000;
24         registers[1] = 16'h0001;
25         registers[2] = 16'h0100;
26         registers[3] = 16'h0010;
27         registers[4] = 16'h1100;
28         registers[5] = 16'h0110;
29         registers[6] = 16'h101;
30         registers[7] = 16'h0111;
31
32     end
33 endmodule
```

Figure 12:Register File code

```
module registersFile_TB;
    reg clk;
    reg [2:0] read_regA;
    reg [2:0] read_regB;
    reg [2:0] write_reg;
    reg [15:0] write_data;
    reg write_enable;
    wire [15:0] read_dataA;
    wire [15:0] read_dataB;

    registersFile reg_file(clk, read_regA, read_regB, write_reg, write_data, write_enable, read_dataA, read_dataB);

    initial begin
        clk = 0;
        read_regA = 0;
        read_regB = 0;
        write_reg = 0;
        write_data = 0;
        write_enable = 0;
        #10;
        write_enable = 1;
        write_data = 16'h1234;
        write_reg = 0;
        #10;
        write_enable = 0;
        read_regA = 0; // read from register 0
        read_regB = 1; // read from register 1
        #10;
        $display("read_dataA = %h", read_dataA);
        $display("read_dataB = %h", read_dataB);
        #10;

         // Test Case 1: Write to register 3 and read from register 3
        write_reg = 3'b011;
        write_data = 16'h1234;
        write_enable = 1;
        #10; // Wait for a clock cycle
        write_enable = 0;

        read_regA = 3'b011;
        #10; // Wait for a clock cycle
        if (read_dataA == 16'h1234) begin
            $display("Test Case 1 Passed: Read data 0x1234 from register 3");
        end else begin
            $display("Test Case 1 Failed: Expected 0x1234, got 0x%h", read_dataA);
        end

        $finish;
    end

    always begin
        #5;
        clk = ~clk;
    end
endmodule
```

Figure 13:Register File TB

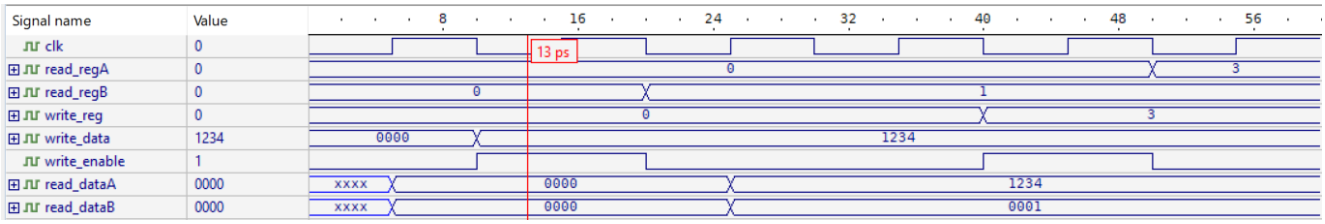


Figure 14:RF_Waveform

IR Code and Waveform:

```
module instruction_register (
    input wire [15:0] instruction,
    output reg [2:0] Rs1,
    output reg [2:0] Rs2,
    output reg [2:0] Rd,
    output reg [4:0] immediate,
    output reg [1:0] instruction_type,
    output reg [11:0] jump_offset,
    output reg [8:0] s_immediate,
    output reg mode // 1-bit mode field for I-Type

);


    always @* begin
        case (instruction[15:12])
            4'b0000, 4'b0001, 4'b0010: begin // R-Type Instructions: AND, ADD, SUB
                instruction_type <= 2'b00;
                Rd <= instruction[11:9];
                Rs1 <= instruction[8:6];
                Rs2 <= instruction[5:3];
         //       immediate = 6'b000000; // Not used in R-Type
         //       jump_offset = 12'b000000000000; // Not used in R-Type
         //       s_immediate = 9'b000000000; // Not used in R-Type
         //     mode = 1'b0; // Not used in R-Type

            end

            4'b0011, 4'b0100, 4'b0101, 4'b0110, 4'b0111,
            4'b1000, 4'b1001, 4'b1010, 4'b1011: begin // I-Type Instructions
                instruction_type <= 2'b01;
                mode <= instruction[11];
                Rd <= instruction[10:8];
                Rs2 <= 3'b000; // Not used in I-Type
                Rs1 <= instruction[7:5];
                immediate <= instruction[4:0];
        //        jump_offset = 12'b000000000000; // Not used in I-Type
        //        s_immediate = 9'b000000000; // Not used in I-Type
            end
```

Figure 15:IR code part1

```
            4'b1100: begin // J-Type: jmp,
                instruction_type = 2'b10;
                Rd = 3'b000; // Not used in J-Type
                Rs2 = 3'b000; // Not used in J-Type
                Rs1 = 3'b000; // Not used in J-Type
         //       immediate = 5'b00000; // Not used in J-Type
                jump_offset = instruction[11:0]; // Used in J-Type
         //       s_immediate = 9'b000000000; // Not used in J-Type
         //       mode = 1'b0; // Not used in J-Type
            end

             4'b1101: begin // J-Type:  call
                instruction_type = 2'b10;
                Rd = 3'b111; // R7
                Rs2 = 3'b000; // Not used in J-Type
                Rs1 = 3'b000; // Not used in J-Type
         //       immediate = 5'b00000; // Not used in J-Type
                jump_offset = instruction[11:0]; // Used in J-Type
         //       s_immediate = 9'b000000000; // Not used in J-Type
         //       mode = 1'b0; // Not used in J-Type
            end

            4'b1110: begin // J-Type: ret
                instruction_type = 2'b10; // J-Type
                Rd = 3'b000; // Not used in ret
                Rs1 = 3'b111; // R7
                Rs2 = 3'b000; // Not used in ret
          //      immediate = 5'b00000; // Not used in ret
          //      jump_offset = 12'b000000000000; // Not used in ret
          //      s_immediate = 9'b000000000; // Not used in ret
          //      mode = 1'b0; // Not used in ret
            end
```

Figure 16:IR code part2

```
            4'b1111: begin // S-Type
                instruction_type = 2'b11;
                Rd = 3'b000;
                Rs2 = 3'b000; // Not used in S-Type
                Rs1 = instruction[11:9];
                immediate = 5'b00000; // Not used in S-Type
                jump_offset = 12'b000000000000; // Not used in S-Type
                s_immediate = instruction[8:0]; // Used in S-Type
                mode = 1'b0; // Not used in S-Type
            end

            default: begin // Default case
                instruction_type = 2'b00;
                Rd = 3'b000;
                Rs2 = 3'b000;
                Rs1 = 3'b000;
                immediate = 5'b00000;
                jump_offset = 12'b000000000000;
                s_immediate = 9'b000000000;
            end
        endcase
    end

endmodule
```

Figure 17:IR code part3

```
`timescale 1ns / 1ps

module instruction_register_TB;

    // Parameters
    localparam CLK_PERIOD = 10; // Clock period in ns

    // Signals
    reg [15:0] instruction;
    reg clk;
    reg reset;

    // Outputs
    reg [2:0] Rs1_tb;
    reg [2:0] Rs2_tb;
    reg [2:0] Rd_tb;
    reg [4:0] immediate_tb;
    reg [1:0] instruction_type_tb;
    reg [11:0] jump_offset_tb;
    reg [8:0] s_immediate_tb;
    reg mode_tb;

    // Instantiate the module under test
    instruction_register dut (
        .instruction(instruction),
        .Rs1(Rs1_tb),
        .Rs2(Rs2_tb),
        .Rd(Rd_tb),
        .immediate(immediate_tb),
        .instruction_type(instruction_type_tb),
        .jump_offset(jump_offset_tb),
        .s_immediate(s_immediate_tb),
        .mode(mode_tb)
    );

    // Clock generation
    always #CLK_PERIOD clk = ~clk;

    // Test stimulus
    initial begin
```

Figure 18:IR TB part 1

```
    initial begin
        // Initialize signals
        clk = 0;
        reset = 1;
        instruction = 16'h0000; // Default instruction

        // Reset
        #20 reset = 0;

        // Test case 1: R-Type instruction (AND R1, R2, R3)
        instruction = 16'b0000001010011000; // AND R1, R2, R3
        #30 $display("Test Case 1:");
        #40 $monitor("R-Type: AND R%d, R%d, R%d | instruction_type = %b, mode = %b", Rd_tb, Rs1_tb, Rs2_tb, instruction_type_tb, mode_tb);
        #50 $monitor("Rs1 = %b, jump_offset = %b, s_immediate = %b", Rs1_tb, jump_offset_tb, s_immediate_tb);

        #20;

        // Test case 2: I-Type instruction (ADDI R6, R2, 26)
        instruction = 16'b0011011001011010; // ADDI R6, R2, 26
        #70 $display("Test Case 2:");
        #80 $monitor("I-Type: ADDI R%d, R%d, %d | instruction_type = %b, mode = %b", Rd_tb, Rs1_tb, immediate_tb, instruction_type_tb, mode_tb);
        #90 $monitor("immediate = %b, jump_offset = %b, Rs1 = %b", immediate_tb, jump_offset_tb, Rs1_tb);
    /*
        // Test case 3: J-Type instruction (JMP to address 100)
        instruction = 16'h1100_0000_0000_0100; // JMP to address 100
        #90 $display("Test Case 3:");
        #100 $monitor("J-Type: JMP to offset %d | instruction_type = %b, mode = %b", jump_offset_tb, instruction_type_tb, mode_tb);
        #110 $monitor("Rd = %b, Rs1 = %b, Rs2 = %b, immediate = %b, s_immediate = %b", Rd_tb, Rs1_tb, Rs2_tb, immediate_tb, s_immediate_tb);

        // Test case 4: S-Type instruction (Store value 255 to memory)
        instruction = 16'h1111_000_111111111; // Sv R0, 255
        #120 $display("Test Case 4:");
        #130 $monitor("S-Type: Store R%d with immediate value %d | instruction_type = %b, mode = %b", Rs1_tb, s_immediate_tb, instruction_type_tb, mode_tb);
        #140 $monitor("Rd = %b, Rs2 = %b, Rs1 = %b, immediate = %b, jump_offset = %b", Rd_tb, Rs2_tb, Rs1_tb, immediate_tb, jump_offset_tb);
     */
        // End simulation
        #150 $finish;
    end

endmodule
```


Figure 19:IR TB Part2

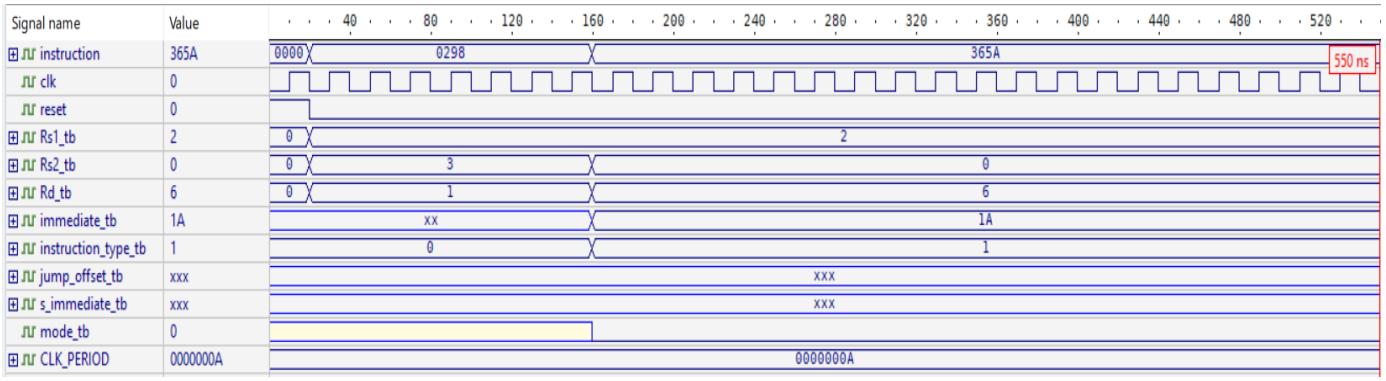


Figure 20:IR Waveform

Extender Code and Waveform:

Sign-extension for signed data: If the input represents signed data, the extender would copy the sign bit (the most significant bit) to fill the additional bits to create a valid 16-bit signed value.

Zero-extension for unsigned data: If the input represents unsigned data, the extender would simply pad the bits with zeros to create a 16-bit unsigned value.

```
module extender (
    input wire [8:0] s_immediate,             // 9-bit input immediate value
    input wire [4:0] immediate,              // 5-bit input immediate value
    input wire [1:0] instruction_type,
    input wire mode,                 //  0 for unsigned, 1 for signed
    output reg [15:0] extended_imm    // 16-bit extended output
);

    always @(*) begin
        if (instruction_type == 2'b11) begin      //  S-type
            // If imm_type is 1, it is an 9-bit immediate (S-type)
            if (mode) begin
                // Sign extension for 9-bit immediate
                extended_imm = {{7{s_immediate[8]}}, s_immediate};
            end else begin
                // Zero extension for 9-bit immediate
                extended_imm = {7'b0, s_immediate};
            end
        end

            if (instruction_type == 2'b01) begin      // I-type
            // If imm_type is 0, it is a 5-bit immediate (I-type)
            if (mode) begin
                // Sign extension for 5-bit immediate
                extended_imm = {{11{immediate[4]}}, immediate[4:0]};
            end else begin
                // Zero extension for 5-bit immediate
                extended_imm = {11'b0, immediate[4:0]};
            end
        end
    end

endmodule
```

Figure 21:Extender Code

```
module extender_TB;

    reg [8:0] s_immediate;              // 9-bit input immediate value for S-type
    reg [4:0] immediate;                // 5-bit input immediate value for I-type
    reg [1:0] instruction_type;         // Control signal for instruction type
    reg mode;                           // Control signal for signed or unsigned extension
    wire [15:0] extended_imm;           // 16-bit extended output

    // Instantiate the extender module
    extender uut (
        .s_immediate(s_immediate),
        .immediate(immediate),
        .instruction_type(instruction_type),
        .mode(mode),
        .extended_imm(extended_imm)
    );
```

Figure 22:Extender TB part1

```
    initial begin
        // Test case 1: S-type, unsigned extension
        s_immediate = 9'b000000000; instruction_type = 2'b11; mode = 0;  // Expected output: 0000000000000000
        #10;
        $display("Test case 1: s_immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", s_immediate, instruction_type, mode, extended_imm);

        // Test case 2: S-type, signed positive number extension
        s_immediate = 9'b000001010; instruction_type = 2'b11; mode = 1;  // Expected output: 0000000000001010 (+10)
        #10;
        $display("Test case 2: s_immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", s_immediate, instruction_type, mode, extended_imm);

        // Test case 3: S-type, signed negative number extension
        s_immediate = 9'b111110010; instruction_type = 2'b11; mode = 1;  // Expected output: 1111111111110010 (-14)
        #10;
        $display("Test case 3: s_immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", s_immediate, instruction_type, mode, extended_imm);

        // Test case 4: I-type, unsigned extension
        immediate = 5'b00000; instruction_type = 2'b01; mode = 0;  // Expected output: 0000000000000000
        #10;
        $display("Test case 4: immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", immediate, instruction_type, mode, extended_imm);

        // Test case 5: I-type, signed positive number extension
        immediate = 5'b01010; instruction_type = 2'b01; mode = 1;  // Expected output: 0000000000001010 (+10)
        #10;
        $display("Test case 5: immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", immediate, instruction_type, mode, extended_imm);

        // Test case 6: I-type, signed negative number extension
        immediate = 5'b11010; instruction_type = 2'b01; mode = 1;  // Expected output: 1111111111111010 (-6)
        #10;
        $display("Test case 6: immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", immediate, instruction_type, mode, extended_imm);

        // Additional test cases for further verification

        // Test case 7: S-type, unsigned positive number extension
        s_immediate = 9'b011100010; instruction_type = 2'b11; mode = 0;  // Expected output: 0000000011100010
        #10;
        $display("Test case 7: s_immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", s_immediate, instruction_type, mode, extended_imm);

        // Test case 8: I-type, unsigned positive number extension
        immediate = 5'b10001; instruction_type = 2'b01; mode = 0;  // Expected output: 0000000000010001
        #10;
        $display("Test case 8: immediate = %b, instruction_type = %b, mode = %b, extended_imm = %b", immediate, instruction_type, mode, extended_imm);

        $finish;  // End simulation
    end

endmodule
```

Figure 23:Extender TB part2

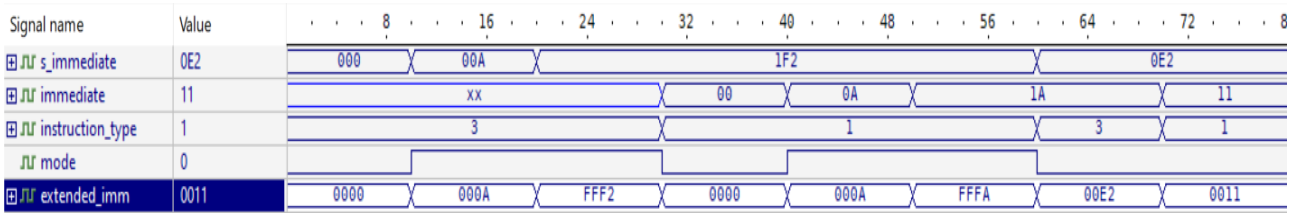


Figure 24:Extender Waveform

```
run
# KERNEL: Test case 1: s_immediate = 000000000, instruction_type = 11, mode = 0, extended_imm = 0000000000000000
# KERNEL: Test case 2: s_immediate = 000001010, instruction_type = 11, mode = 1, extended_imm = 0000000000001010
# KERNEL: Test case 3: s_immediate = 111110010, instruction_type = 11, mode = 1, extended_imm = 1111111111110010
# KERNEL: Test case 4: immediate = 00000, instruction_type = 01, mode = 0, extended_imm = 0000000000000000
# KERNEL: Test case 5: immediate = 01010, instruction_type = 01, mode = 1, extended_imm = 0000000000001010
# KERNEL: Test case 6: immediate = 11010, instruction_type = 01, mode = 1, extended_imm = 1111111111111010
# KERNEL: Test case 7: s_immediate = 011100010, instruction_type = 11, mode = 0, extended_imm = 0000000011100010
# KERNEL: Test case 8: immediate = 10001, instruction_type = 01, mode = 0, extended_imm = 0000000000010001
# RUNTIME: Info: RUNTIME 0068 extender TB.v (61): $finish called.
```

Figure 25:extender results

Multiplexers:

In this project different Multiplexers are used :

```
module mux2x1 (
    input wire [15:0] in0,   // First 16-bit input
    input wire [15:0] in1,   // Second 16-bit input
    input wire sel,          // Select signal
    output wire [15:0] out   // 16-bit output
);

    // Output the selected input based on the select signal
    assign out = sel ? in1 : in0;

endmodule
```

Figure 26:2x1 mux code

```
module mux2x1_tb;

    reg [15:0] in0;      // First 16-bit input for the MUX
    reg [15:0] in1;      // Second 16-bit input for the MUX
    reg sel;             // Select signal for the MUX
    wire [15:0] out;     // Output of the MUX

    // Instantiate the Mux2x1 module
    mux2x1 uut (
        .in0(in0),
        .in1(in1),
        .sel(sel),
        .out(out)
    );

    initial begin
        // Test case 1: sel = 0, out should be in0
        in0 = 16'hAAAA;  // 1010101010101010
        in1 = 16'h5555;  // 0101010101010101
        sel = 0;
        #10;
        $display("Test case 1: in0 = %h, in1 = %h, sel = %b, out = %h", in0, in1, sel, out);

        // Test case 2: sel = 1, out should be in1
        sel = 1;
        #10;
        $display("Test case 2: in0 = %h, in1 = %h, sel = %b, out = %h", in0, in1, sel, out);

        // Test case 3: Change inputs and sel = 0
        in0 = 16'h1234;  // 0001001000110100
        in1 = 16'h5678;  // 0101011001111000
        sel = 0;
        #10;
        $display("Test case 3: in0 = %h, in1 = %h, sel = %b, out = %h", in0, in1, sel, out);

        // Test case 4: Change inputs and sel = 1
        sel = 1;
        #10;
        $display("Test case 4: in0 = %h, in1 = %h, sel = %b, out = %h", in0, in1, sel, out);

        $finish;  // End simulation
    end

endmodule
```

Figure 27:2x1 muxTB

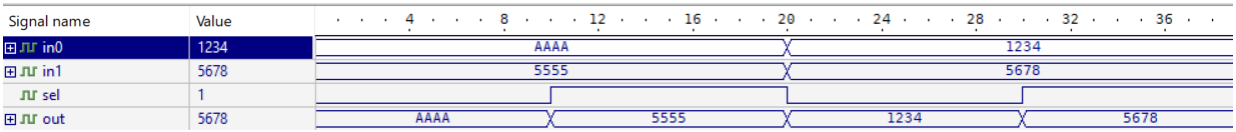


Figure 28:2x1mux Waveform

```verilog
module mux3x1 (
    input wire [15:0] in0,     // Input 0
    input wire [15:0] in1,     // Input 1
    input wire [15:0] in2,     // Input 2
    input wire [1:0] sel,      // 2-bit selector
    output reg [15:0] out      // Output
);

    always @(*) begin
        case (sel)
            2'b00: out = in0;  // Select input 0
            2'b01: out = in1;  // Select input 1
            2'b10: out = in2;  // Select input 2
            default: out = 16'b0; // Default case (should not happen)
        endcase
    end

endmodule
```

Figure 29: 3x1 mux code

```verilog
module mux_3x1_TB;

    reg [15:0] in0;  // Input 0
    reg [15:0] in1;  // Input 1
    reg [15:0] in2;  // Input 2
    reg [1:0] sel;   // 2-bit selector
    wire [15:0] out; // Output

    // Instantiate the mux_3x1 module
    mux_3x1 uut (
        .in0(in0),
        .in1(in1),
        .in2(in2),
        .sel(sel),
        .out(out)
    );

    initial begin
        // Test case 1: Select input 0
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; sel = 2'b00;  // Expected output: AAAA
        #10;
        $display("Test case 1: in0 = %h, in1 = %h, in2 = %h, sel = %b, out = %h", in0, in1, in2, sel, out);

        // Test case 2: Select input 1
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; sel = 2'b01;  // Expected output: BBBB
        #10;
        $display("Test case 2: in0 = %h, in1 = %h, in2 = %h, sel = %b, out = %h", in0, in1, in2, sel, out);

        // Test case 3: Select input 2
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; sel = 2'b10;  // Expected output: CCCC
        #10;
        $display("Test case 3: in0 = %h, in1 = %h, in2 = %h, sel = %b, out = %h", in0, in1, in2, sel, out);

        // Test case 4: Invalid selector
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; sel = 2'b11;  // Expected output: 0000 (default case
        #10;
        $display("Test case 4: in0 = %h, in1 = %h, in2 = %h, sel = %b, out = %h", in0, in1, in2, sel, out);

        $finish;  // End simulation
    end

endmodule
```

Figure 30:3x1 mux TB

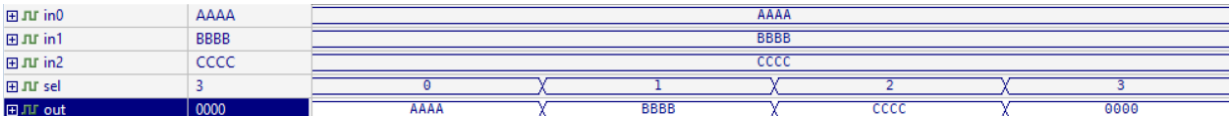


Figure 31:3x1 mux Waveform

```
module mux4x1 (
    input wire [15:0] in0,      // Input 0
    input wire [15:0] in1,      // Input 1
    input wire [15:0] in2,      // Input 2
    input wire [15:0] in3,      // Input 3
    input wire [1:0] sel,       // 2-bit selector
    output reg [15:0] out       // Output
);

    always @(*) begin
        case (sel)
            2'b00: out = in0;  // Select input 0
            2'b01: out = in1;  // Select input 1
            2'b10: out = in2;  // Select input 2
            2'b11: out = in3;  // Select input 3
            default: out = 16'b0; // Default case (should not happen)
        endcase
    end

endmodule
```

Figure 32:4x1 mux code

```
module mux4x1_TB;

    reg [15:0] in0;  // Input 0
    reg [15:0] in1;  // Input 1
    reg [15:0] in2;  // Input 2
    reg [15:0] in3;  // Input 3
    reg [1:0] sel;   // 2-bit selector
    wire [15:0] out; // Output

    // Instantiate the mux_4x1 module
    mux4x1 uut (
        .in0(in0),
        .in1(in1),
        .in2(in2),
        .in3(in3),
        .sel(sel),
        .out(out)
    );

    initial begin
        // Test case 1: Select input 0
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; in3 = 16'hDDDD; sel = 2'b00;  // Expected output: AAAA
        #10;
        $display("Test case 1: in0 = %h, in1 = %h, in2 = %h, in3 = %h, sel = %b, out = %h", in0, in1, in2, in3, sel, out);

        // Test case 2: Select input 1
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; in3 = 16'hDDDD; sel = 2'b01;  // Expected output: BBBB
        #10;
        $display("Test case 2: in0 = %h, in1 = %h, in2 = %h, in3 = %h, sel = %b, out = %h", in0, in1, in2, in3, sel, out);

        // Test case 3: Select input 2
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; in3 = 16'hDDDD; sel = 2'b10;  // Expected output: CCCC
        #10;
        $display("Test case 3: in0 = %h, in1 = %h, in2 = %h, in3 = %h, sel = %b, out = %h", in0, in1, in2, in3, sel, out);

        // Test case 4: Select input 3
        in0 = 16'hAAAA; in1 = 16'hBBBB; in2 = 16'hCCCC; in3 = 16'hDDDD; sel = 2'b11;  // Expected output: DDDD
        #10;
        $display("Test case 4: in0 = %h, in1 = %h, in2 = %h, in3 = %h, sel = %b, out = %h", in0, in1, in2, in3, sel, out);

        $finish;  // End simulation
    end

endmodule
```

Figure 33:4x1 mux TB

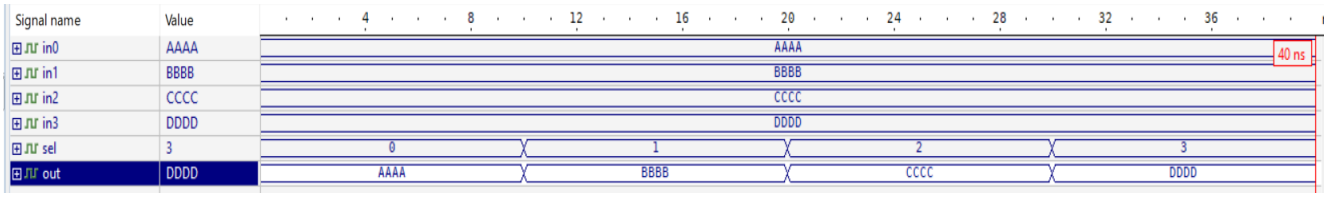


Figure 34: 4x1 mux Waveform

Multiplexers used in many positions in the Datapath:

- Next Instruction to fetch, we have 4 choices: PC+2, Jump address, Branch address, address saved in R7.
- Choose the first operand to be read from the RF: either RS1 or R7.
- Choose the second operand to be read rom RF: either RS2 or Rd.
- Choose the destination register: either Rd or R7.
- Choose either BUSb or immediate will be the second input to the ALU.
- Choose between immediate and BUSb to be as Data_InSrc to the Data Memory.
- Choose what will be written if there is a write back process: either output of ALU or Data out from Memory or Next PC.

Control Units:

Control units in this project are needed to generate the signals that control the correctness of the procedure of the CPU, there are 3 types of control units:

- Main control unit: this unit generates the Main control signals.
- ALU Control unit: this unit generates the ALU control Signals.
- PC Control unit: used to generates the signal PCSrc.

→ The following is how to implement the Main and ALU control unit:

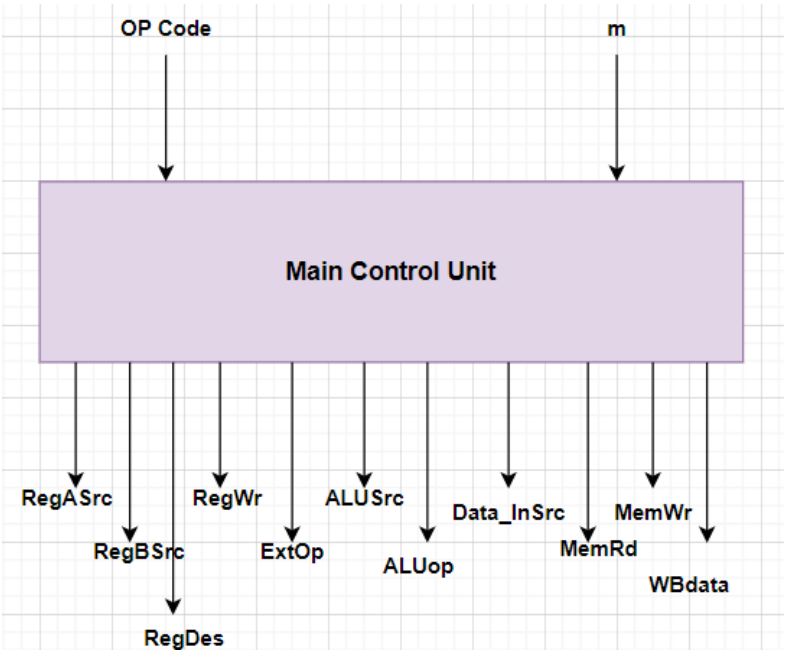


Figure 35:Main and ALU Control unit

- Truth Table:

opCode	m bit	RegAsrc	RegBSrc	RegDes	ExtOp	RegWr	ALUSrc	ALUOp	Data_InSrc	MemRd	MemWr	Wbdata
AND	X	0	0	0	X	1	1	AND	X	0	0	1
ADD/SUB	X	0	0	0	X	1	1	ADD/SUB	X	0	0	1
ADDI	X	0	X	0	1	1	0	ADD	X	0	0	1
ANDI	X	0	X	0	0	1	0	AND	X	0	0	1
LW	X	0	X	0	1	1	0	ADD	X	1	0	2
LBu	0	0	X	0	0	1	0	ADD	X	1	0	2
LBs	1	0	X	0	1	1	0	ADD	X	1	0	2
SW	X	0	1	X	1	0	0	ADD	1	0	1	X
BGT	0	0	1	X	1	0	1	SUB	X	0	0	X
BGTZ	1	0	1	X	1	0	1	SUB	X	0	0	X
BLT	0	0	1	X	1	0	1	SUB	X	0	0	X
BLTZ	1	0	1	X	1	0	1	SUB	X	0	0	X
BEQ	0	0	1	X	1	0	1	SUB	X	0	0	X
BEQZ	1	0	1	X	1	0	1	SUB	X	0	0	X
BNE	0	0	1	X	1	0	1	SUB	X	0	0	X
BNEZ	1	0	1	X	1	0	1	SUB	X	0	0	X
JMP	X	X	X	X	X	0	X	X	X	0	0	X
CALL	X	X	X	1	X	1	X	X	X	0	0	0
RET	X	1	X	X	X	0	X	X	X	0	0	X
SV	X	0	X	X	0	0	1	ADD	0	0	1	X

Figure 36:Main &ALU Truth table

- Boolean Expressions:

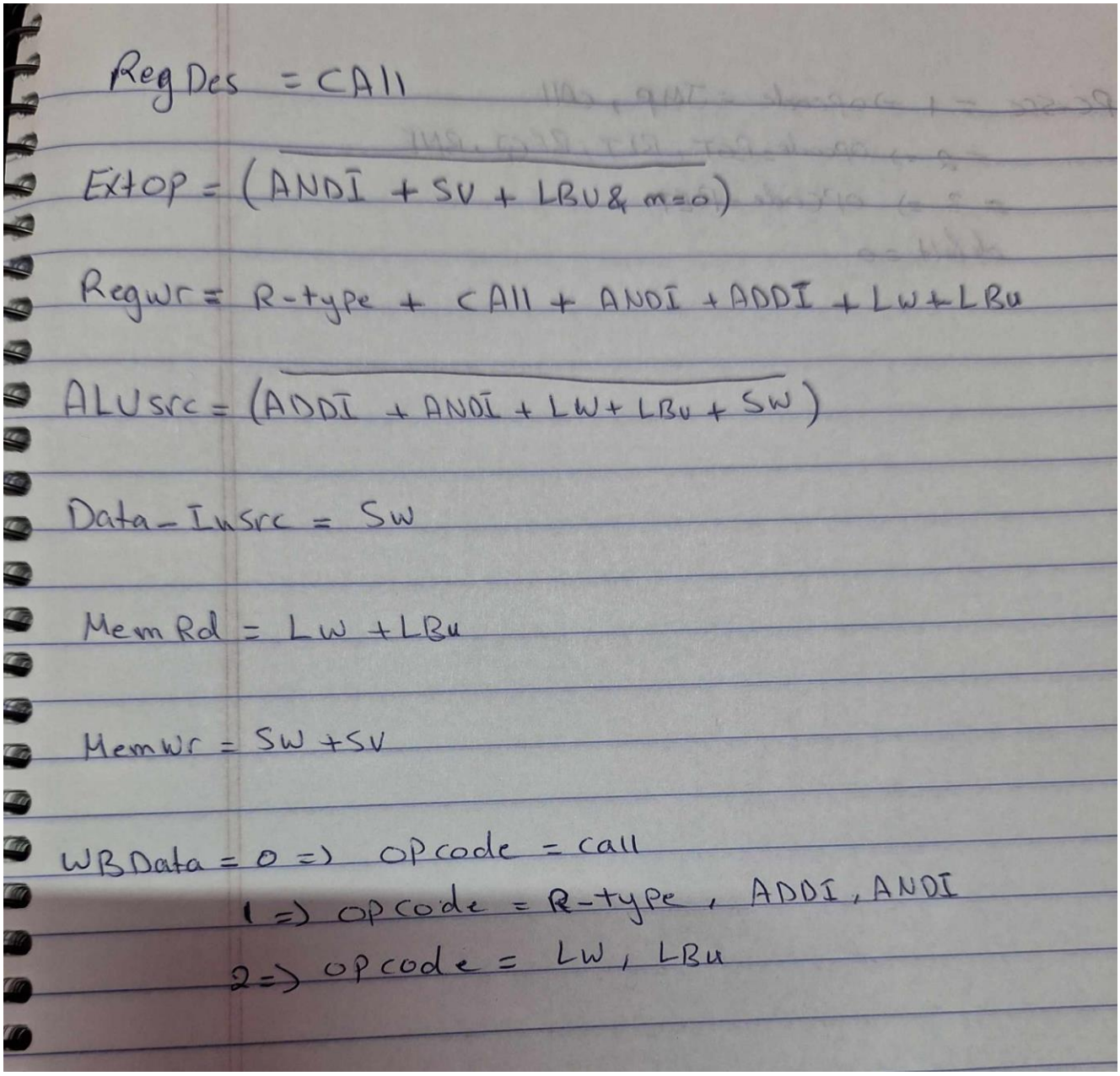


Figure 37: Boolean Expressions for Control unit signals

```
module ControlUnit (
    input [3:0] op,       // 4-bit operation code
    input m,              // Mode bit
    output reg RegASrc,
    output reg RegBSrc,
    output reg RegDes,
    output reg ExtOp,
    output reg RegWr,
    output reg ALUSrc,
    output reg [3:0] ALUOp, // 4-bit ALUOp
    output reg Data_InSrc,
    output reg MemRd,
    output reg MemWr,
    output reg [1:0] Wbdata  // 2-bit Wbdata
);

// Define operation codes (assuming 4-bit codes)
parameter AND   = 4'b0000;
parameter ADD   = 4'b0001;
parameter SUB   = 4'b0010;
parameter ADDI  = 4'b0011;
parameter ANDI  = 4'b0100;
parameter LW    = 4'b0101;
parameter LBu   = 4'b0110; // LBu and LBs have the same opc
parameter SW    = 4'b0111;
parameter BGT   = 4'b1000;
parameter BLT   = 4'b1001;
parameter BEQ   = 4'b1010;
parameter BNE   = 4'b1011;
parameter JMP   = 4'b1100;
parameter CALL  = 4'b1101;
parameter RET   = 4'b1110;
parameter SV    = 4'b1111;

always @(*) begin
    // Default values
    RegASrc = 1'b0;
    RegBSrc = 1'b0;
    RegDes = 1'b0;
    ExtOp = 1'b0;
    RegWr = 1'b0;
    ALUSrc = 1'b0;
    ALUOp = 4'b0000;
    Data_InSrc = 1'b0;
    MemRd = 1'b0;
    MemWr = 1'b0;
    Wbdata = 2'b00;

    // Set RegASrc based on RET instruction
    if (op == RET) begin
        RegASrc = 1'b1;
    end

    // Set RegBSrc
    if (!(op == AND || op == SUB || op == ADD)) begin
        RegBSrc = 1'b1;
    end

    // Set RegDes
    if (op == CALL) begin
        RegDes = 1'b1;
    end

    // Set ExtOp
    if (!(op == SV || (op == LBu && m == 1'b0) || op == ANDI)) begin
        ExtOp = 1'b1;
    end

    // Set RegWr
    if (!(op == SW || op==BGT ||op==BLT ||op==BEQ||op==BNE || op == JMP || op == RET || op == SV)) begin
        RegWr = 1'b1;
    end

    // Set ALUSrc
    if (!(op == ADDI || op == ANDI || op == LW || op == SW || op == LBu)) begin
        ALUSrc = 1'b1;
    end

    // Set Data_InSrc
    if (op == SW) begin
        Data_InSrc = 1'b1;
    end

    // Set MemRd
    if (op == LW || op == LBu ) begin
        MemRd = 1'b1;
    end
```

Figure 38:Main Control code part1

```
    // Set MemWr
    if (op == SW || op == SV) begin
        MemWr = 1'b1;
    end

    // Set Wbdata
    case (op)
        CALL: Wbdata = 2'b00;
        AND, ANDI, ADD, ADDI, SUB: Wbdata = 2'b01;
        LW, LBu: Wbdata = 2'b10;
        default: Wbdata = 2'b00;
    endcase

    // Set ALUOp
    case (op)
        AND, ANDI: ALUOp = 4'b0000;
        ADD, ADDI, LW, SW, LBu,SV: ALUOp = 4'b0001;
        SUB: ALUOp = 4'b0010;
        BGT, BLT, BEQ, BNE: ALUOp = 4'b0010; // SUB operation for comparison
        default: ALUOp = 4'b0000;
    endcase
end

endmodule
```

Figure 39:Main Control Code part2

```
module ControlUnit_tb;

    // Inputs
    reg [3:0] op;
    reg m;

    // Outputs
    wire RegASrc;
    wire RegBSrc;
    wire RegDes;
    wire ExtOp;
    wire RegWr;
    wire ALUSrc;
    wire [3:0] ALUOp;
    wire Data_InSrc;
    wire MemRd;
    wire MemWr;
    wire [1:0] Wbdata;

    // Instantiate the ControlUnit
    ControlUnit uut (
        .op(op),
        .m(m),
        .RegASrc(RegASrc),
        .RegBSrc(RegBSrc),
        .RegDes(RegDes),
        .ExtOp(ExtOp),
        .RegWr(RegWr),
        .ALUSrc(ALUSrc),
        .ALUOp(ALUOp),
        .Data_InSrc(Data_InSrc),
        .MemRd(MemRd),
        .MemWr(MemWr),
        .Wbdata(Wbdata)
    );

    // Test stimulus
    initial begin
        // Initialize Inputs
        op = 4'b0000; m = 0; // AND
        #10;
        $display("op=0000, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b0001; m = 0; // ADD
        #10;
        $display("op=0001, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);
```

Figure 40: Main Control TB

```
        op = 4'b0010; m = 0; // SUB
        #10;
        $display("op=0010, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b0011; m = 0; // ADDI
        #10;
        $display("op=0011, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b0100; m = 0; // ANDI
        #10;
        $display("op=0100, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b0101; m = 0; // LW
        #10;
        $display("op=0101, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b0110; m = 1; // LBu
        #10;
        $display("op=0110, m=1 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b0110; m = 0; // LBs
        #10;
        $display("op=0110, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b0111; m = 0; // SW
        #10;
        $display("op=0111, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1000; m = 0; // BGT
        #10;
        $display("op=1000, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1000; m = 1; // BGTZ
        #10;
        $display("op=1000, m=1 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1001; m = 0; // BLT
        #10;
        $display("op=1001, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);
```

Figure 41: TB part2

```
        op = 4'b1001; m = 1; // BLTZ
        #10;
        $display("op=1001, m=1 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1010; m = 0; // BEQ
        #10;
        $display("op=1010, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1010; m = 1; // BEQZ
        #10;
        $display("op=1010, m=1 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1011; m = 0; // BNE
        #10;
        $display("op=1011, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1011; m = 1; // BNEZ
        #10;
        $display("op=1011, m=1 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1100; m = 0; // JMP
        #10;
        $display("op=1100, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1101; m = 0; // CALL
        #10;
        $display("op=1101, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1110; m = 0; // RET
        #10;
        $display("op=1110, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        op = 4'b1111; m = 0; // SV
        #10;
        $display("op=1111, m=0 -> RegASrc=%b, RegBSrc=%b, RegDes=%b, ExtOp=%b, RegWr=%b, ALUSrc=%b, ALUOp=%b, Data_InSrc=%b, MemRd=%b, MemWr=%b, Wbdata=%b",
                 RegASrc, RegBSrc, RegDes, ExtOp, RegWr, ALUSrc, ALUOp, Data_InSrc, MemRd, MemWr, Wbdata);

        $finish;
    end

endmodule
```

Figure 42:TB part3

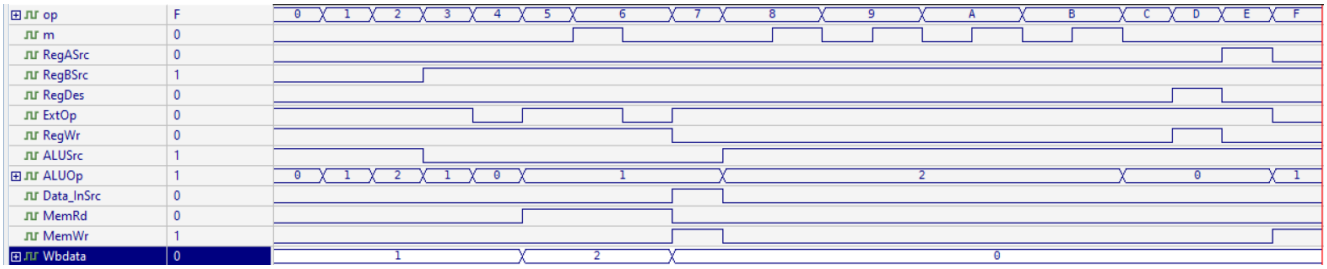


Figure 43: Main control Waveform

→ The following is how the implementation of the PC control unit was:

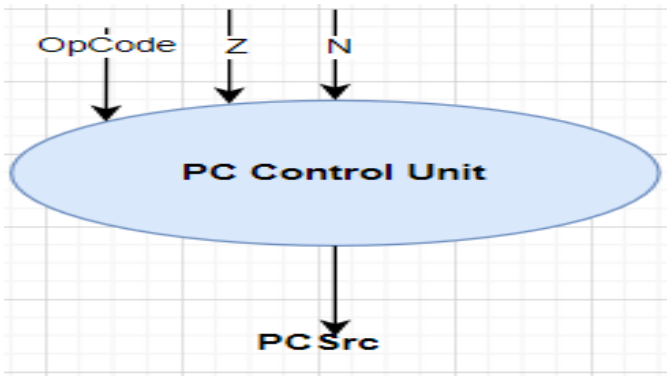


Figure 44:PC Control unit

- Truth Table:

op code	Z	N	PC_Src
AND	X	X	0
ADD/SUB	X	X	0
ADDI/AND	X	X	0
LW	X	X	0
Lbu/LBs	X	X	0
SW	X	X	0
BGT	X	0	2
BGTZ	X	0	2
BLT	X	1	2
BLTZ	X	1	2
BEQ	1	X	2
BEQZ	1	X	2
BNE	0	X	2
BNEZ	0	X	2
JMP	X	X	1
CALL	X	X	1
RET	X	X	3
SV	X	X	0

Figure 45: PC Control Truth table

- Boolean Expression:

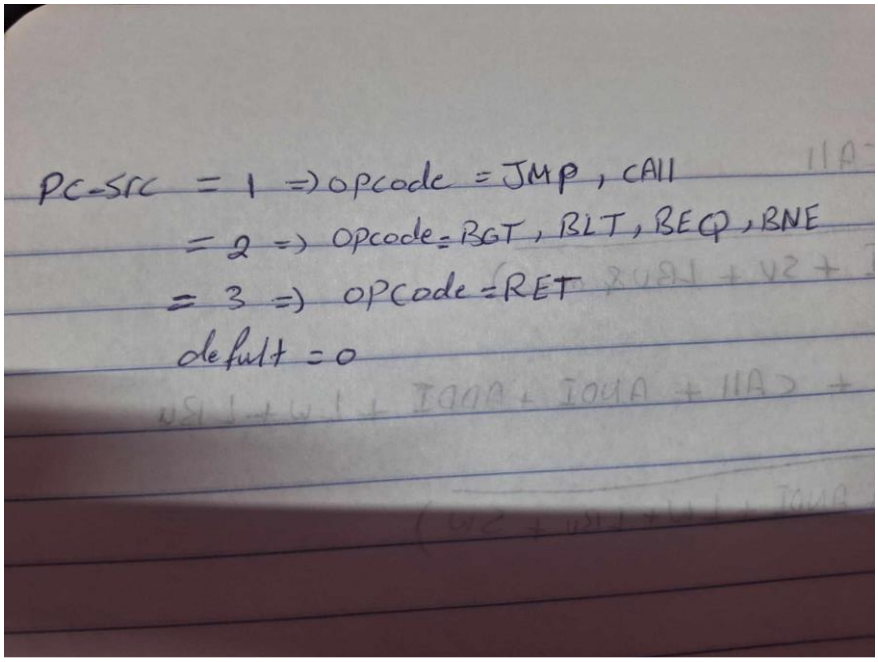


Figure 46:Expressions for PC-Sr

- Code:

```
module PC_Control(
    input [3:0] op_code,
    input Z,
    input N,
    output reg [1:0] PC_Src
);

    // Define operation codes (assuming 4-bit codes)
    parameter AND   = 4'b0000;
    parameter ADD   = 4'b0001;
    parameter SUB   = 4'b0010;
    parameter ADDI  = 4'b0011;
    parameter ANDI  = 4'b0100;
    parameter LW    = 4'b0101;
    parameter LBu   = 4'b0110;
    parameter SW    = 4'b0111;
    parameter BGT   = 4'b1000;
    parameter BLT   = 4'b1001;
    parameter BEQ   = 4'b1010;
    parameter BNE   = 4'b1011;
    parameter JMP   = 4'b1100;
    parameter CALL  = 4'b1101;
    parameter RET   = 4'b1110;
    parameter SV    = 4'b1111;

    always @(*) begin
        case (op_code)
            BGT,BLT,BEQ,BNE: begin
                PC_Src <= 2'b10; // set the pc_src to the value 2
            end
            JMP, CALL: begin
                PC_Src <= 2'b01; // Set PC_Src to 1 for JMP and CALL instructions
            end
            RET: begin
                PC_Src <= 2'b11; // Set PC_Src to 3 for RET instruction
            end
            default: begin
                PC_Src <= 2'b00; // Default case
            end
        endcase
    end

endmodule
```

Figure 47:PC control unit

```
module pc_Control_TB();

    // Inputs

    reg [3:0] op_code;
    reg Z, N;

    // Outputs
    wire [1:0] PC_Src;

    // Instantiate the module
    PC_Control uut (
        .op_code(op_code),
        .Z(Z),
        .N(N),
        .PC_Src(PC_Src)
    );


    // Stimulus
    initial begin
        //$dumpfile("pc_Control_TB.vcd");
        //$dumpvars(0, pc_Control_TB);


        // Test default case (AND, ADD, SUB, ADDI, ANDI, LW, LBu, SW, SV)
        op_code = 4'b0000; Z = 0; N = 0; #10; // AND
        op_code = 4'b0001; Z = 0; N = 0; #10; // ADD
        op_code = 4'b0010; Z = 0; N = 0; #10; // SUB
        op_code = 4'b0011; Z = 0; N = 0; #10; // ADDI
        op_code = 4'b0100; Z = 0; N = 0; #10; // ANDI
        op_code = 4'b0101; Z = 0; N = 0; #10; // LW
        op_code = 4'b0110; Z = 0; N = 0; #10; // LBu
        op_code = 4'b0111; Z = 0; N = 0; #10; // SW
        op_code = 4'b1111; Z = 0; N = 0; #10; // SV
```

Figure 48: PC control unit TB part1

```
        // Test BGT instruction (PC_Src = 2 if N = 0)
        op_code = 4'b1000; Z = 0; N = 0; #10;
        op_code = 4'b1000; Z = 0; N = 1; #10;

        // Test BLT instruction (PC_Src = 2 if N = 1)
        op_code = 4'b1001; Z = 0; N = 1; #10;
        op_code = 4'b1001; Z = 0; N = 0; #10;

        // Test BEQ instruction (PC_Src = 2 if Z = 1)
        op_code = 4'b1010; Z = 1; N = 0; #10;
        op_code = 4'b1010; Z = 0; N = 0; #10;

        // Test BNE instruction (PC_Src = 2 if Z = 0)
        op_code = 4'b1011; Z = 0; N = 0; #10;
        op_code = 4'b1011; Z = 1; N = 0; #10;

        // Test JMP instruction (PC_Src = 1)
        op_code = 4'b1100; Z = 0; N = 0; #10;

        // Test CALL instruction (PC_Src = 1)
        op_code = 4'b1101; Z = 0; N = 0; #10;

        // Test RET instruction (PC_Src = 3)
        op_code = 4'b1110; Z = 0; N = 0; #10;

        $finish;
    end

endmodule
```

Figure 49: PC control unit TB part2

ALU Implementation:

The Arithmetic Logic Unit (ALU) is responsible for carrying out arithmetic and logical operations as well as setting flags that regulate program execution flow and make decisions in the central processing unit (CPU) of a computer. The Zero Flag (Z), Carry Flag (C), and Overflow Flag (V) are examples of these flags. The Overflow Flag indicates when an arithmetic overflow occurs, the Carry Flag assists in managing carry or borrow during addition and subtraction, and the Zero Flag is set when the outcome of an operation is zero. These flags enable the CPU to react to various circumstances and carry out different actions depending on the outcomes of ALU operations. They are utilized in conditional branching and decision-making instructions.

```
module alu (
    input [15:0] A,         // First operand
    input [15:0] B,         // Second operand
    input [3:0] ALUop,      // ALU operation code
    output reg [15:0] Result,  // Result of the operation
    output Zero,            // Zero flag
    output CarryOut,        // Carry out flag
    output Overflow,        // Overflow flag
    output Negative         // Negative flag
);

    reg [16:0] temp;        // Temporary variable to store intermediate results

    always @(*) begin
        case (ALUop)
            4'b0000: Result <= A & B;        // AND operation
            4'b0001: begin                   // ADD operation
                temp <= A + B;
                Result <= temp[15:0];
            end
            4'b0010: begin                   // SUB operation
                temp <= A - B;
                Result <= temp[15:0];
            end
            4'b0011: Result <= A | B;        // OR operation
            // Add other operations here
            default: Result <= 16'b0;        // Default case
        endcase
    end

    assign Zero = (Result == 16'b0) ? 1'b1 : 1'b0;
    assign CarryOut = temp[16];
    assign Overflow = (A[15] & B[15] & ~Result[15]) | (~A[15] & ~B[15] & Result[15]);
    assign Negative = Result[15];           // Negative flag is set if the most significant bit is 1

endmodule
```

Figure 50: ALU code

```
module alu_TB;
    reg [15:0] a;
    reg [15:0] b;
    reg [3:0] aluop;
    wire [15:0] result;
    wire zero;
    wire carryOut;
    wire overflow;
    wire negative;

    // Instantiate the ALU
    alu alu (
        .A(a),
        .B(b),
        .ALUop(aluop),
        .Result(result),
        .Zero(zero),
        .CarryOut(carryOut),
        .Overflow(overflow),
        .Negative(negative)
    );
```

Figure 51: ALU TB part1

```
initial begin
    // Initialize Inputs
    a = 0;
    b = 0;
    aluop = 0;
    #10;

    // Test ADD operation
    a = 16'h000F;
    b = 16'h0004;
    aluop = 4'b0001;  // ADD
    #10;
    $display("ADD: a = %h, b = %h, aluop = %h, result = %h, zero = %b, carryOut = %b, overflow = %b, negative = %b", a, b, aluop, result, zero, carryOut, overflow, negative);
    #10;

    // Test SUB operation
    aluop = 4'b0010;  // SUB
    #10;
    $display("SUB: a = %h, b = %h, aluop = %h, result = %h, zero = %b, carryOut = %b, overflow = %b, negative = %b", a, b, aluop, result, zero, carryOut, overflow, negative);
    #10;

    // Test AND operation
    a = 16'h1111;
    b = 16'h0004;
    aluop = 4'b0000;  // AND
    #10;
    $display("AND: a = %h, b = %h, aluop = %h, result = %h, zero = %b, carryOut = %b, overflow = %b, negative = %b", a, b, aluop, result, zero, carryOut, overflow, negative);
    #10;

    // Test OR operation
    a = 16'h0001;
    b = 16'h0002;
    aluop = 4'b0011;  // OR
    #10;
    $display("OR: a = %h, b = %h, aluop = %h, result = %h, zero = %b, carryOut = %b, overflow = %b, negative = %b", a, b, aluop, result, zero, carryOut, overflow, negative);
    #10;
```

Figure 52:ALU TB part2

```
        // Test if Zero flag is set correctly
        a = 16'h0000;
        b = 16'h0000;
        aluop = 4'b0001;  // ADD
        #10;
        $display("Zero Flag: a = %h, b = %h, aluop = %h, result = %h, zero = %b, carryOut = %b, overflow = %b, negative = %b", a, b, aluop, result, zero, carryOut, overflow, negative
        #10;

        // Test if Negative flag is set correctly
        a = 16'h8000;
        b = 16'h0000;
        aluop = 4'b0001;  // ADD
        #10;
        $display("Negative Flag: a = %h, b = %h, aluop = %h, result = %h, zero = %b, carryOut = %b, overflow = %b, negative = %b", a, b, aluop, result, zero, carryOut, overflow, nega
        #10;

        $finish;
    end
endmodule
```

Figure 53: ALU TB part3

Figure 54: ALU Waveform

State Diagram:

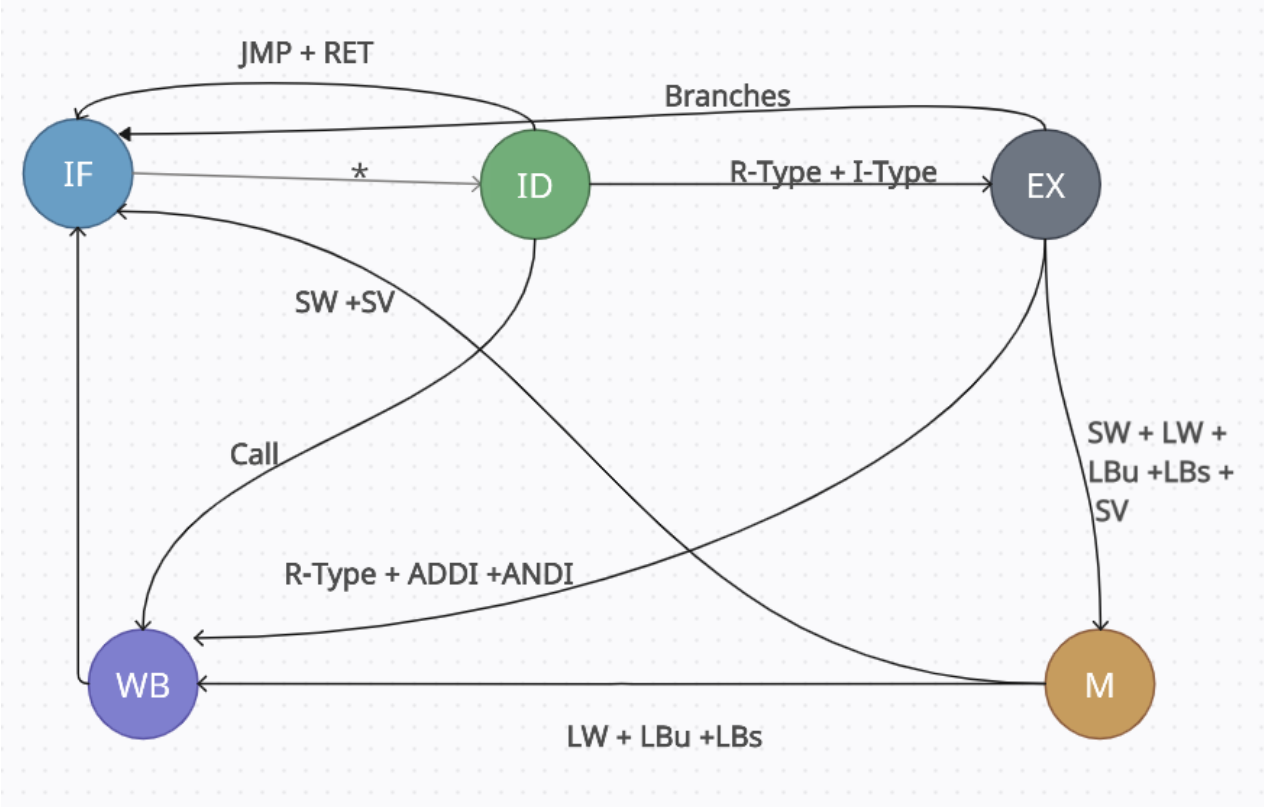


Figure 55: State Diagram

Datapath:

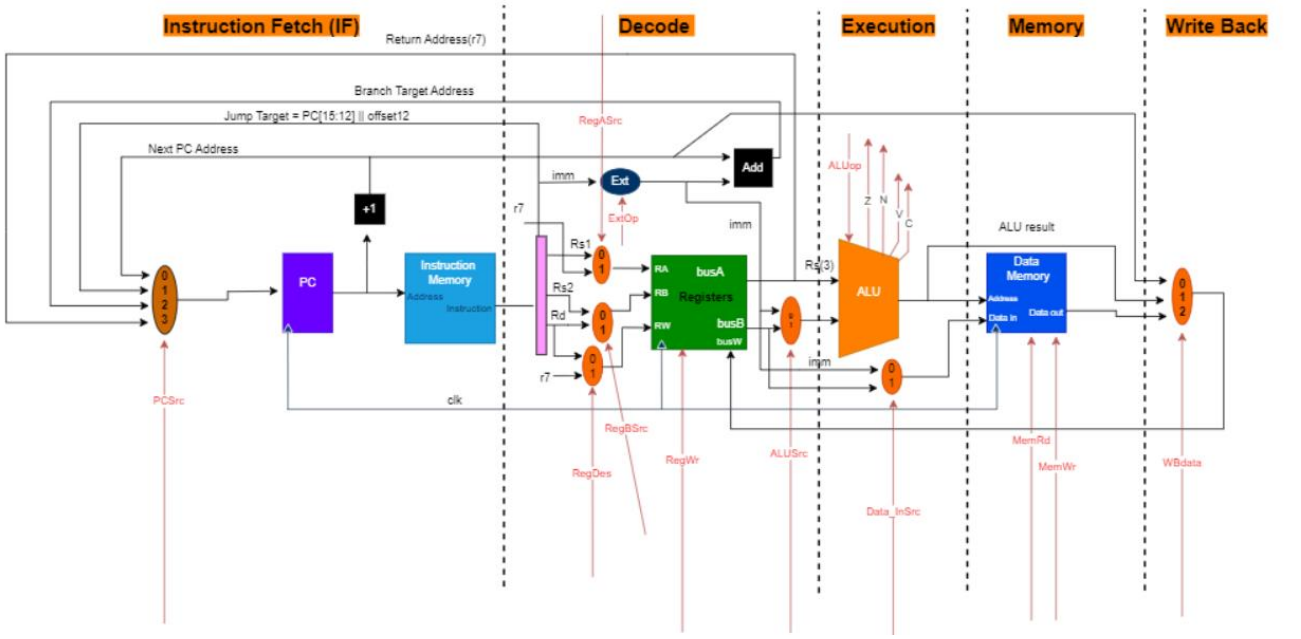


Figure 56: Datapath

CPU Code:

```
module cpu(
  input clk,
  input [15:0] InstructionMemory [0:255],
  output reg [15:0] result,
  output reg [15:0] currentInstruction,
  output reg [15:0] operand1,
  output reg [15:0] operand2,
  output reg [15:0] data_out,
  output reg [15:0] data_in
);

reg [15:0] DataMemory [0:255]; // 16-bit data memory with 256 locations
reg [15:0] registers [0:7]; // Array of 8 16-bit registers
reg [15:0] pc = 16'b0; // Program counter

// Instruction Fields
reg [3:0] opcode;
reg [2:0] rs1, rs2, rd;
reg [4:0] immediate;
reg mode;
reg [11:0] jumpOffset;
reg [8:0] svImmediate;
reg [1:0] Type;
reg stop;
reg MemRead;
reg MemWrite;
reg RegWrite;
reg ExtOP;
reg Z, C, N;
reg EnableFetch = 1'b1;
reg EnableDecode = 1'b0;
reg EnableALU = 1'b0;
reg EnableMemoryAccess = 1'b0;
reg EnableWriteBack = 1'b0;

// Initialize data memory and registers
initial begin
  integer i;
  for (i = 0; i < 256; i = i + 1) begin
    DataMemory[i] = 0;
  end
  for (i = 0; i < 8; i = i + 1) begin
    registers[i] = i;
  end
end
```

```
// Instruction Fetch Stage
always @ (posedge clk) begin
  if (EnableFetch) begin
    currentInstruction = InstructionMemory[pc];
    #1;
    EnableFetch = 1'b0;
    EnableDecode = 1'b1;
  end
end

// Instruction Decode Stage
always @ (posedge clk) begin
  if (EnableDecode) begin
    opcode = currentInstruction[15:12];
    //Type = currentInstruction[1:0];
     //mode = currentInstruction[11];

   case ( currentInstruction[15:12])
            4'b0000, 4'b0001, 4'b0010: begin // R-Type Instructions: AND, ADD,
                Type = 2'b00;

              end

            4'b0011, 4'b0100, 4'b0101, 4'b0110, 4'b0111,
            4'b1000, 4'b1001, 4'b1010, 4'b1011: begin // I-Type Instructions
                Type = 2'b01;
                  //mode <= instruction[11];


            end

            4'b1100: begin // J-Type: jmp,
                Type = 2'b10;

              end

               4'b1101: begin // J-Type:  call
                Type = 2'b10;

              end

              4'b1110: begin // J-Type: ret
                Type = 2'b10; // J-Type

            end
```

Figure 57: CPU code part1

```
            4'b1111: begin // S-Type
                Type = 2'b11;

              end

            default: begin // Default case
                Type = 2'b00;

            end
        endcase

   // Generate control signals
    if (opcode == 4'b0000 && Type== 2'b00) begin // AND
         rs1 = currentInstruction[8:6];
        rs2 = currentInstruction[5:3];
        rd = currentInstruction[11:9];
        operand1 = registers[rs1];
        operand2 = registers[rs2];
      MemRead = 1'b0;
      MemWrite = 1'b0;
      RegWrite = 1'b1;
    end

else if (opcode == 4'b0001 &&  Type== 2'b00) begin // ADD
       rs1 = currentInstruction[8:6];
       rs2 = currentInstruction[5:3];
       rd = currentInstruction[11:9];
       operand1 = registers[rs1];
       operand2 = registers[rs2];
     MemRead = 1'b0;
     MemWrite = 1'b0;
     RegWrite = 1'b1;
end
```

```
else if (opcode == 4'b0011 && Type== 2'b01) begin // ADDI
      rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
      operand1 = registers[rs1];
      operand2 = currentInstruction[4:0];
     MemRead = 1'b0;
     MemWrite = 1'b0;
     RegWrite = 1'b1;
   end
else if (opcode == 4'b0100 && Type== 2'b01) begin // ANDI
      rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
      operand1 = registers[rs1];
      operand2 = currentInstruction[4:0];
     MemRead = 1'b0;
     MemWrite = 1'b0;
     RegWrite = 1'b1;
end

else if (opcode == 4'b0101 && Type== 2'b01) begin // LW
      rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
      operand1 = registers[rs1];
      operand2 = currentInstruction[4:0];
     MemRead = 1'b1;
     MemWrite = 1'b0;
     RegWrite = 1'b1;
end
```

Figure 58: CPU code Part2

```
else if (opcode == 4'b0110 && Type== 2'b01) begin // LBu / LBs
       rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     operand1 = registers[rs1];
     operand2 = currentInstruction[4:0];
    MemRead = 1'b1;
    MemWrite = 1'b0;
    RegWrite = 1'b1;
  end

else if (opcode == 4'b0111 && Type== 2'b01) begin // SW
       rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     operand1 = registers[rs1];
     operand2 = currentInstruction[4:0];
    MemRead = 1'b0;
    MemWrite = 1'b1;
    RegWrite = 1'b0;
    data_in = registers[rd];
end

else if (opcode == 4'b1000 && Type== 2'b01 && mode ==0) begin // BGT
       rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     immediate = currentInstruction[4:0];
     operand1 = registers[rd];
     operand2 = registers[rs1];
    MemRead = 1'b0;
    MemWrite = 1'b0;
    RegWrite = 1'b0;
  end
else if (opcode == 4'b1000 && Type== 2'b01 && mode ==1) begin // BGT
       rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     immediate = currentInstruction[4:0];
     operand1 = registers[rd];
     operand2 = registers[0];
    MemRead = 1'b0;
    MemWrite = 1'b0;
    RegWrite = 1'b0;
end
```

```
else if (opcode == 4'b1001 && Type== 2'b01  && mode ==0) begin // BL
       rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     immediate = currentInstruction[4:0];
     operand1 = registers[rd];
     operand2 = registers[rs1];
    MemRead = 1'b0;
    MemWrite = 1'b0;
    RegWrite = 1'b0;
  end

else if (opcode == 4'b1001 && Type== 2'b01  && mode ==1) begin // BL
     rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     immediate = currentInstruction[4:0];
     operand1 = registers[rd];
     operand2 = registers[0];
    MemRead = 1'b0;
    MemWrite = 1'b0;
    RegWrite = 1'b0;
  end

else if (opcode == 4'b1010 && Type== 2'b01 && mode ==0) begin // BEQ
       rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     immediate = currentInstruction[4:0];
     operand1 = registers[rd];
     operand2 = registers[rs1];
    MemRead = 1'b0;
    MemWrite = 1'b0;
    RegWrite = 1'b0;
end

 else if (opcode == 4'b1010 && Type== 2'b01 && mode ==1) begin // BE
       rs1 = currentInstruction[7:5];
      rd = currentInstruction[10:8];
       mode = currentInstruction[11];
     immediate = currentInstruction[4:0];
     operand1 = registers[rd];
     operand2 = registers[0];
    MemRead = 1'b0;
    MemWrite = 1'b0;
```

Figure 59: CPU code part3

```
260        MemWrite = 1'b0;
261        RegWrite = 1'b0;
262      end
263
264    else if (opcode == 4'b1011 && Type== 2'b01 && mode ==0) begin // BN
265          rs1 = currentInstruction[7:5];
266         rd = currentInstruction[10:8];
267          mode = currentInstruction[11];
268         immediate = currentInstruction[4:0];
269         operand1 = registers[rd];
270         operand2 = registers[rs1];
271        MemRead = 1'b0;
272        MemWrite = 1'b0;
273        RegWrite = 1'b0;
274    end
275
276    else if (opcode == 4'b1011 && Type== 2'b01 && mode ==1) begin // BN
277          rs1 = currentInstruction[7:5];
278         rd = currentInstruction[10:8];
279          mode = currentInstruction[11];
280         immediate = currentInstruction[4:0];
281         operand1 = registers[rd];
282         operand2 = registers[0];
283        MemRead = 1'b0;
284        MemWrite = 1'b0;
285        RegWrite = 1'b0;
286      end
287
288    else if (opcode == 4'b1100 && Type== 2'b10) begin // JMP
289        jumpOffset = currentInstruction[11:0];
290        MemRead = 1'b0;
291        MemWrite = 1'b0;
292        RegWrite = 1'b0;
293        pc = {pc[15:12], jumpOffset};
294    end
295
296    else if (opcode == 4'b1101 && Type== 2'b10) begin // CALL
297        jumpOffset = currentInstruction[11:0];
298        MemRead = 1'b0;
299        MemWrite = 1'b0;
300        RegWrite = 1'b0;
301        pc = {pc[15:12], jumpOffset};
302        registers[7] = pc + 1; // Save return address in r7
303    end
```

```
305      else if (opcode == 4'b1110 && Type== 2'b10) begin // RET
306        MemRead = 1'b0;
307        MemWrite = 1'b0;
308        RegWrite = 1'b0;
309        pc = registers[7]; // Return to address in r7
310      end
311    else if (opcode == 4'b1111 && Type== 2'b11) begin // Sv
312          rs1 = currentInstruction[11:9];
313        svImmediate = currentInstruction[8:0];
314        operand1 = registers[rs1];
315        operand2 = svImmediate;
316        MemRead = 1'b0;
317        MemWrite = 1'b1;
318        RegWrite = 1'b0;
319        data_in = operand2; // Store immediate value
320      end
321
322
323     if (opcode != 4'b1100 && opcode != 4'b1110) begin
324          #1
325          EnableFetch = 1'b0;
326          EnableDecode = 1'b0;
327         EnableALU = 1'b1;
328          EnableMemoryAccess = 1'b0;
329          EnableWriteBack = 1'b0;
330      end
331    end
332  end
333
334  // ALU Stage
335  always @ (posedge clk) begin
336    if (EnableALU) begin
337        if (opcode == 4'b0000 && Type== 2'b00) begin // AND
338        result = operand1 & operand2;
339      end
340
341      else if (opcode == 4'b0001 && Type== 2'b00) begin // ADD
342        result = operand1 + operand2;
343        C = (result < operand1);
344      end
345
346      else if (opcode == 4'b0010 && Type== 2'b00) begin // SUB
347        result = operand1 - operand2;
348        C = operand1 < operand2;
349      end
350
```

Figure 60:CPU code part4

```
    else if (opcode == 4'b0011 && Type== 2'b01) begin // ADDI
      result = operand1 + operand2;
      C = (result < operand1) || (result < operand2);
    end

    else if (opcode == 4'b0100 && Type== 2'b01) begin // ANDI
      result = operand1 & operand2;
    end

    else if (opcode == 4'b0101 && Type== 2'b01) begin // LW
      result = operand1 + operand2;
       C = (result < operand1) || (result < operand2);
    end

    else if (opcode == 4'b0110 && Type== 2'b01) begin // LBu / LBs
      result = operand1 + operand2;
       C = (result < operand1) || (result < operand2);
    end

    else if (opcode == 4'b0111 && Type== 2'b01) begin // SW
      result = operand1 + operand2;
       C = (result < operand1) || (result < operand2);
    end

    else if (opcode == 4'b1000 && Type== 2'b01 && mode ==0) begin
      if (operand1 > operand2) begin
        pc = pc + immediate;
         end
      else begin
        pc = pc + 2;
      end

        #1
        EnableFetch = 1'b1;
        EnableDecode = 1'b0;
        EnableALU = 1'b0;
        EnableMemoryAccess = 1'b0;
        EnableWriteBack = 1'b0;

    end

    else if (opcode == 4'b1000 && Type== 2'b01 && mode ==1) begin
      if (operand1 > operand2)begin
        pc = pc + immediate;
         end
      else begin
        pc = pc + 2;
      end

      #1
         EnableFetch = 1'b1;
         EnableDecode = 1'b0;
         EnableALU = 1'b0;
         EnableMemoryAccess = 1'b0;
         EnableWriteBack = 1'b0;


    end

    else if (opcode == 4'b1001 && Type== 2'b01 && mode ==0) begin // BLT
      if (operand1 < operand2)begin
        pc = pc + immediate;
         end
      else begin
        pc = pc + 2;
      end

      #1
         EnableFetch = 1'b1;
         EnableDecode = 1'b0;
         EnableALU = 1'b0;
         EnableMemoryAccess = 1'b0;
         EnableWriteBack = 1'b0;


    end

    else if (opcode == 4'b1001 && Type== 2'b01 && mode ==1) begin //BLTZ
      if (operand1 < operand2)begin
        pc = pc + immediate;
         end
      else begin
        pc = pc + 2;
      end
          #1
         EnableFetch = 1'b1;
         EnableDecode = 1'b0;
         EnableALU = 1'b0;
         EnableMemoryAccess = 1'b0;
         EnableWriteBack = 1'b0;
```

Figure 61: CPU code part5

```
end
else if (opcode == 4'b1010 && Type== 2'b01 && mode ==0) begin // BE
  if (operand1 == operand2)begin
    pc = pc + immediate;
     end
  else begin
    pc = pc + 2;
  end
     #1
     EnableFetch = 1'b1;
     EnableDecode = 1'b0;
     EnableALU = 1'b0;
     EnableMemoryAccess = 1'b0;
     EnableWriteBack = 1'b0;


end

else if (opcode == 4'b1010 && Type== 2'b01 && mode ==1) begin // BE
  if (operand1 == operand2)begin
    pc = pc + immediate;
     end
  else begin
    pc = pc + 2;
  end
     #1
     EnableFetch = 1'b1;
     EnableDecode = 1'b0;
     EnableALU = 1'b0;
     EnableMemoryAccess = 1'b0;
     EnableWriteBack = 1'b0;


end

else if (opcode == 4'b1011 && Type== 2'b01 && mode ==0) begin // BN
  if (operand1 != operand2)begin
    pc = pc + immediate;
     end
  else begin
    pc = pc + 2;
  end
     #1
     EnableFetch = 1'b1;
     EnableDecode = 1'b0;
```

```
     EnableALU = 1'b0;
     EnableMemoryAccess = 1'b0;
     EnableWriteBack = 1'b0;


end

else if (opcode == 4'b1011 && Type== 2'b01 && mode ==1) begin // BNEZ
 if (operand1 != operand2)begin
    pc = pc + immediate;
     end
  else begin
    pc = pc + 2;
  end
     #1
     EnableFetch = 1'b1;
     EnableDecode = 1'b0;
     EnableALU = 1'b0;
     EnableMemoryAccess = 1'b0;
     EnableWriteBack = 1'b0;


end


if (!(opcode >= 4'b1000 &&  opcode <= 4'b1011 &&Type== 2'b10))    // if no
                                begin
     #1
     EnableFetch = 1'b0;
     EnableDecode = 1'b0;
     EnableALU = 1'b0;
     EnableMemoryAccess = 1'b1;
     EnableWriteBack = 1'b0;
     end
  end
end

// Memory Access Stage
always @ (posedge clk) begin
  if (EnableMemoryAccess) begin
    if (MemRead) begin
      data_out = DataMemory[result];
    end else if (MemWrite) begin
      DataMemory[result] = data_in;
    end
```

Figure 62: CPU code part6

```
    end
      else begin
     data_out = result;
     end


   if (opcode == 4'b0111 && Type== 2'b01 || opcode == 4'b1111 && Type== 2'b11) begin     // if SW -> fetch is next
         #1
         pc = pc +1;
         EnableFetch = 1'b1;
         EnableDecode = 1'b0;
         EnableALU = 1'b0;
         EnableMemoryAccess = 1'b0;
         EnableWriteBack = 1'b0;
         end
         else begin    // if anything other than SW -> go to WriteBack
         #1
         EnableFetch = 1'b0;
         EnableDecode = 1'b0;
         EnableALU = 1'b0;
         EnableMemoryAccess = 1'b0;
         EnableWriteBack = 1'b1;
         end
  end
end

// Write Back Stage
always @ (posedge clk) begin
  if (EnableWriteBack) begin
    if (RegWrite) begin
      registers[rd] = data_out;
    end

    #1
    pc = pc+1;
    EnableFetch = 1'b1; // go back to fetch
    EnableDecode = 1'b0;
    EnableALU = 1'b0;
    EnableMemoryAccess = 1'b0;
    EnableWriteBack = 1'b0;

  end
```

Figure 63: CPU code part7

```
  end

  endmodule

  `timescale 1ns / 1ns

  module testbench;
    reg clk;
    reg [15:0] InstructionMemory [0:255];
    reg [15:0] result;
    reg [15:0] currentInstruction;
    reg [15:0] operand1;
    reg [15:0] operand2;
    reg [15:0] data_out;
    reg [15:0] data_in;

    cpu myCpu(.clk(clk), .InstructionMemory(InstructionMemory), .result(result),
              .currentInstruction(currentInstruction), .operand1(operand1),
              .operand2(operand2), .data_out(data_out), .data_in(data_in));

    initial begin
      clk = 0;
      repeat(500) begin
        #10 clk = ~clk;
      end
    end

    initial begin
      // Initialize Instruction Memory with test instructions

        InstructionMemory[0]  = 16'b0000_001_010_011_000; // AND R1, R2, R3
       InstructionMemory[1]  = 16'b0001_101_010_001_000; // ADD R5, R1, R2
       InstructionMemory[2]  = 16'b0010011101001000; // SUB R3, R5, R1
           // I type
          InstructionMemory[3]  = 16'b0011_0_110_010_11010; // ADDI R6, R2, 26
          //InstructionMemory[4]  = 16'b0100000111001101; // ANDI R1, R6, 13
          //InstructionMemory[5]  = 16'b0101001010011010; // LW R2, R4, 26
          //InstructionMemory[6]  = 16'b0110001000110100; // LBu R2, R1, 20
          //InstructionMemory[7]  = 16'b0110101000110100; // LBS R2, R1, 20

      /*InstructionMemory[0] = 16'b001100100100001; // ADDI R1, R1, 1
      InstructionMemory[1] = 16'b001101001000001; // ADDI R2, R2, 1
      InstructionMemory[2] = 16'b101000100000001; // BEQ R1, R0, 1 (PC relative address)
      InstructionMemory[3] = 16'b011100100000000; // SW R1, 0 (R0)
      InstructionMemory[4] = 16'b010101000000000; // LW R2, 0 (R0)
      InstructionMemory[5] = 16'b0010001010000;   // SUB R1, R2
```

Figure 64: CPU code part 8

```
620        InstructionMemory[5] = 16'b0010001010000;   // SUB R1, R2
621        InstructionMemory[6] = 16'b1100000000000000;  // JMP to address 0
622        InstructionMemory[7] = 16'b0000001010000;   // AND R1, R2
623
624
625        */
626
627        // Additional instructions for testing can be added here
628    end
629    endmodule
```

Figure 65: CPU code part 9

```
InstructionMemory[0]  = 16'b0000_001_010_011_000; // AND R1, R2, R3
InstructionMemory[1]  = 16'b0001_101_010_001_000; // ADD R5, R1, R2
InstructionMemory[2]  = 16'b0010011101001000; // SUB R3, R5, R1
   // I type
InstructionMemory[3]  = 16'b0011_0_110_010_11010; // ADDI R6, R2, 26
```

```
InstructionMemory[4] = 16'b1111_101_000000110; // SV R5 6   // R5 value is 4, so the address in memory will be 4
InstructionMemory[5]  = 16'b0101_0_010_000_00100; // LW R2, R0,4
InstructionMemory[6]  = 16'b0001_001_010_100_000; // ADD R1, R2, R4
```

Figure 66: Instructions

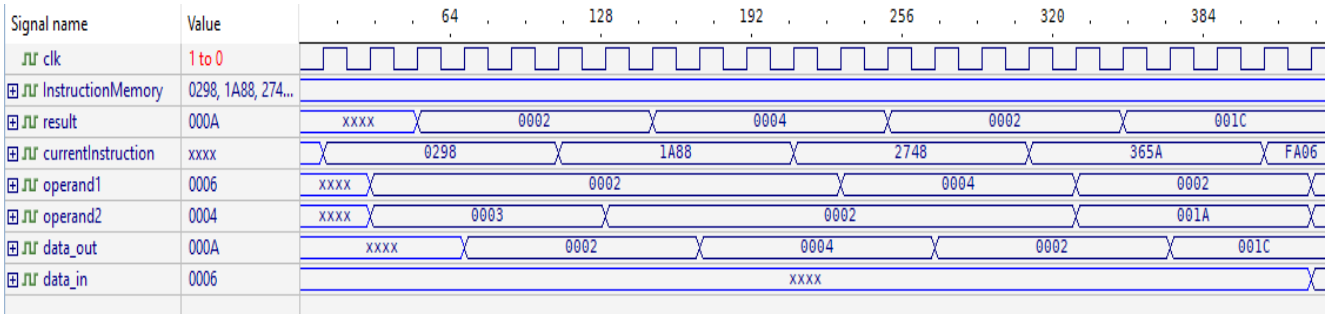


Figure 67: Waveform for CPU

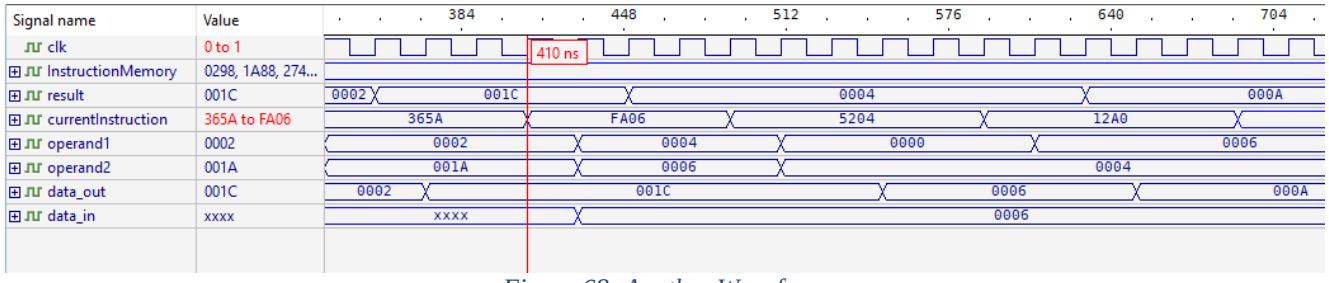


Figure 68: Another Waveform