Department of Electrical & Computer Engineering

ENCS4370 - Computer Architecture

Design and Implementation of a Multi-Cycle Processor for MIPS Instructions

Prepared By:

Saja Shareef	1200901
Shereen Ibdah	1200373
Hamza Awashra	1201619

Date: January 22, 2024

Table of Contents

Table of Figures:

Table of Tables

Abstract

This report explores the creation and application of a Multi-Cycle Processor following a specified instruction set. We designed it by carefully looking at each instruction and identifying the essential parts it requires. The next step involved constructing these parts and figuring out how to control them. To ensure everything works as expected, we tested each component separately, connected these components, tested each instruction separately, and also tested instructions that depend on each other.

Introduction:

In this project, the aim is to design, model, and simulate a MIPS Multi-Cycle Processor using Verilog HDL. The approach taken involves breaking down the processor into sub-modules, each of which is individually designed, coded, and tested. Once these sub-modules are verified as fully functional, they are integrated into a structural module to constitute the complete processor.

RISC Overview:

RISC, or Reduced Instruction Set Computing, prioritizes enhanced CPU performance through a streamlined instruction set. With fixed-length instructions, identical registers, simple addressing, and one-cycle execution, RISC processors deliver speed and cost-effectiveness, widely adopted in the industry.

Multi-Cycle Processors:

Multi-cycle processors divide instruction execution into stages, allowing efficient processing of complex instructions. Unlike single-cycle counterparts, they enable varied instruction completion times, optimizing overall performance and accommodating diverse instruction types.

In a multi-cycle processor, each instruction undergoes several stages, with each stage completing within a single clock cycle. These stages include:

1. Instruction Fetch (IF):

Fetching the instruction from memory using the program counter (PC) and updating the PC for the next instruction.

2. Instruction Decode (ID):

Decoding the fetched instruction to identify the operation and fetching any necessary operands or data from registers.

3. Execution (EX):

Performing the actual operation specified by the instruction, which may involve arithmetic calculations, logical operations, or address computations.

4. Memory Access (MEM):

Accessing memory, such as loading or storing data, if the instruction requires it. This stage involves reading from or writing to memory.

5. Write Back (WB):

Writing the results of the previous stage back to the appropriate register(s), and updating the register file with the computed values.

Design Specifications and Implementation,

Processor Properties:

1. Instruction and Word Size: Instructions and words are uniformly set at 32 bits for consistent data handling.
2. General-Purpose Registers: Equipped with 16 versatile 32-bit general-purpose registers, labeled from R0 to R15, offering a storage for varied data.
3. Special Purpose Registers: Features a 32-bit program counter (PC) for efficient program sequencing.
4. Stack Management: Employs a 32-bit stack pointer (SP), visible to the programmer, directing to the topmost empty element of the Last in First Out (LIFO) stack. This stack records crucial information like return addresses and register values during function calls.
5. Program Memory Layout: Divided into three segments: Static data, Code, and Stack. The Stack, operating on a LIFO basis, incorporates explicit instructions enabling push/pop operations, facilitating the storage of essential data.
6. Physical Memories: Utilizes two separate physical memories—one for instructions and the other for data. The data memory caters to both the static data segment and the stack segment.
7. Instruction Types: Supports four instruction types: R-type, I-type, J-type, and S-type, accommodating diverse program needs.
8. Memory Structure: Adopts word-addressable memory, enhancing efficiency in data manipulation and retrieval.
9. ALU Signal Generation: Utilizes the Arithmetic Logic Unit (ALU) to generate essential signals—such as zero, carry, overflow—critical for calculating condition branch outcomes (taken/not taken) during program execution.

Instruction Types and Formats

The Instruction Set Architecture (ISA) features four distinct instruction formats: R-type, I-type, J-type, and S-type. These formats share a common 6-bit opcode field that determines the specific operation of the instruction.

R-Type (Register Type):

- Opcode (6 bits)
- Rd (4 bits) - Destination register
- Rs1 (4 bits) - First source register
- Rs2 (4 bits) - Second source register
- Unused (14 bits)

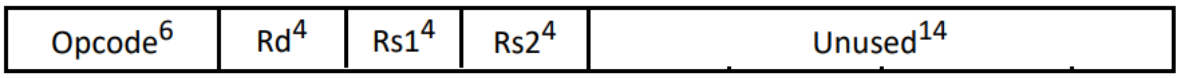

Figure 1:R-Type Format

I-Type (Immediate Type):

- Opcode (6 bits)
- Rd (4 bits) - Destination register
- Rs1 (4 bits) - First source register
- Immediate (16 bits) - Unsigned for logic instructions, signed otherwise
- Mode (2 bits) - Used with load/store instructions
- 00: No increment/decrement of the base register
- 01: Post increment the base register (e.g., LW.POI)
- 10-11: Unused

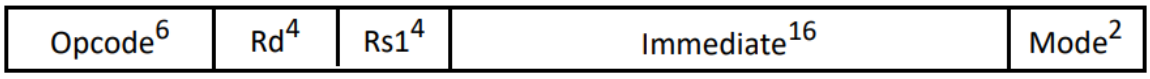

Figure 2:I-Type Format

J-Type (Jump Type):

- Opcode (6 bits)
- Jump Offset (26 bits)
- jmp L: Unconditional jump to target L

- call F: Call function F, pushing return address on the stack
- ret: Return from a function, next PC is the top element of the stack

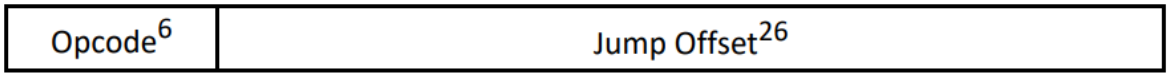

Figure 3: J-Type Format

S-Type (Stack):

- Opcode (6 bits)
- Rd (4 bits)
- Unused (22 bits)
- push Rd: Push the value of Rd onto the top of the stack
- pop Rd: Pop the stack and store the topmost element in Rd

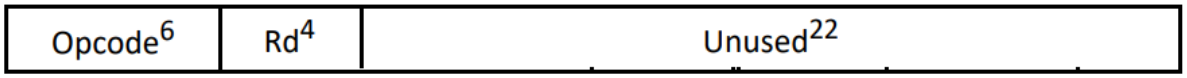

Figure 4: S-Type Format

Instruction Set

The figure below shows the instructions supported by this instruction set, with their meaning and decoding.

No.	Instr	Meaning	Opcode Value
		R-Type Instructions	
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	000000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	000001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	000010
		I-Type Instructions	
4	ANDI	Reg(Rd) = Reg(Rs1) & Imm^{16}	000011
5	ADDI	Reg(Rd) = Reg(Rs1) + Imm^{16}	000100
6	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{16})	000101
7	LW.POI	Reg(Rd) = Mem(Reg(Rs1) + Imm^{16}) Reg[Rs1] = Reg[Rs1] + 1	000110
8	SW	Mem(Reg(Rs1) + Imm^{16}) = Reg(Rd)	000111
9	BGT	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001000
10	BLT	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001001
11	BEQ	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001010
12	BNE	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001011
		J-Type Instructions	
13	JMP	Next PC = {PC[31:26], $Immediate^{26}$ }	001100
14	CALL	Next PC = {PC[31:26], $Immediate^{26}$ } PC + 1 is pushed on the stack	001101
15	RET	Next PC = top of the stack	001110
		S-Type Instructions	
16	PUSH	Rd is pushed on the top of the stack	001111
17	POP	The top element of the stack is popped, and it is stored in the Rd register	010000

Figure 5:Instruction Set

Data Path Design and details and description

The first step in designing a data path for a computer's microarchitecture is to create a Register Transfer Level (RTL) description for each instruction. This RTL description should determine the required Functional Units (FUs) and the necessary buses for each instruction.

- RTL for R-types Instructions:

IR ← Mem [PC]

Data1 ← Reg [Rs1]

Data2 ← Reg [Rs2]

Alu_res ← funct (data1, data2)

Reg [Rd] ← alu_res

PC ← PC+1

Where funct is and operation when the operation is AND and addition when the operation is ADD and subtraction when the operation is SUB.

- RTL for ANDI, ADDI(I-type):

IR ← MEM[PC]

data1 ← Reg (Rs1)

data2 ← Extend(imm16)

ALU_result ← op (data1, data2)

Reg (Rd) ← ALU_result

PC ← PC + 4

Where the operation is and when opcode is ANDI and addition when opcode is ADDI.

- RTL for I-type: LW instruction

IR ← MEM[PC]

Base ← Reg(Rs1)

Address ← base + sign_extend(imm16)

Data ← MEM[address]

Reg (Rd) ← data

PC ← PC + 1

- RTL for I-type: SW instruction

IR ← MEM[PC]

Base ← Reg(Rs1)

Address ← base + sign_extend(imm16)

MEM [address] ← Reg(Rd)

PC ← PC + 1

- RTL for I-type: LW.POI instruction

IR ← MEM[PC]

base ← Reg (Rs1)

address ← base + sign_extend(imm16)

data ← MEM [address]

Reg (Rd) ← data

Reg (Rs1) ← Reg (Rs1) +1

PC← PC + 1

- RTL for (I-type) Branch (BEQ)

IR ← MEM[PC]

data1 ← Reg (Rs1), data2 ← Reg (Rd)

zero← subtract (data1, data2)

if (zero) PC ← PC + sign_ext(offset16)

else PC ← PC + 1

- RTL for (I-type) Branch (BNE)

IR ← MEM[PC]

data1 ← Reg (Rs1), data2 ← Reg (Rd)

zero ← subtract (data1, data2)

if (! zero): PC ← PC + sign_ext(offset16)

else PC ← PC + 1

- RTL for (I-type) Branch (BGT)

IR ← MEM[PC]

data1 ← Reg (Rs1), data2 ← Reg (Rd)

zero ← subtract (data1, data2)

if (N! = V): PC ← PC + sign_ext(offset16)

else PC ← PC + 1

- RTL for (I-type) Branch (BLT)

IR ← MEM[PC]

data1 ← Reg (Rs1), data2 ← Reg (Rd)

zero ← subtract (data1, data2)

if (! Z && N == V): PC ← PC + sign_ext(offset16)

else PC ← PC + 1

- RTL for (J-type) JMP

Instruction ← MEM[PC]

target ← PC [31:26] || address26

PC ← target

- RTL for(S-type) CALL

Instruction ← MEM[PC]

target ← PC[31:26], Immediate26

Push (PC + 1)

PC ← target

- RTL for (S-type) RET

Instruction ← MEM[PC]

PC ← Top of the stack

Functional Units needed in data path:

1- **Instruction memory:** It stores the machine instructions that the CPU fetches and executes, playing a fundamental role in program execution and overall system performance, word addressable with size 4GB.
2- **Register File:** with 16 32-bit general-purpose registers: from R0 to R15.
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 a stack allocated as part of its structure, where each cell has a size of 32 bits (one word)
5- **Extender:** The extender is a component that is used to expand the width of data from 16 bits to 32 bits, and it can support both signed and unsigned data representations.

Instruction and Data memory

The memories in the implementation are separated into two parts, instruction memory, and data memory. This was done to solve some conflicts, such as, one instruction might be fetching the instruction from the memory and the other instruction is loading/storing some data from/to the memory, so in order to obey the isolation principle, they need to be separated into different memory elements.

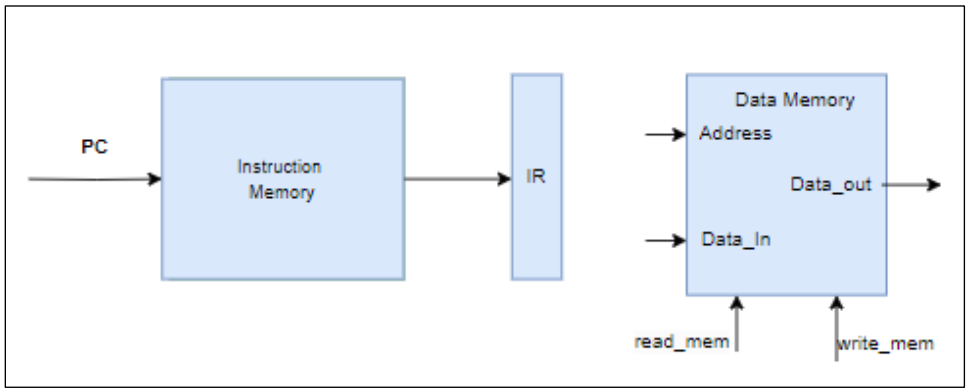

Figure 6 instruction and data memory

The input to the instruction memory is the PC, which is used to fetch and store instructions in the IP register. Conversely, the Data memory has two inputs, namely address and data_in. These inputs are managed by control units generated by the Main Control Unit.

Code and Test bench for Data memory:

```
parameter data_mem_size = 1024 ;

    module Data_mem (
        input clk,
        input [31:0] address,
        input [31:0] data_in,
        output reg [31:0] data_out,
        input mem_read, // signal for reading from memory
        input mem_write // signal for writing on memory
        );

        reg [31:0] mem[0:data_mem_size - 1];

        always @(posedge clk ) begin

            if (mem_write)
               mem[address] = data_in;

            else if (mem_read)
               data_out = mem[address];
        end

        initial begin

            // initialize the memory with temporary value.
            for (int i = 0; i < 256; i = i + 1) begin
              mem[i] <= 32'hAAAAAAAA;
              mem[9] <= 32'h99999999;
              mem[8] <= 32'h88888888;
            end

        end
```

```
module Data_mem_tb;
    // Inputs
    reg clk;
    reg [31:0] address;
    reg [31:0] data_in;
    reg mem_read;
    reg mem_write;
    wire [31:0] data_out;
    Data_mem uut (
        .clk(clk),
        .address(address),
        .data_in(data_in),
        .data_out(data_out),
        .mem_read(mem_read),
        .mem_write(mem_write)
    );
    always #20 clk = ~clk;
    initial begin
        // Initialize Inputs
        clk = 0;
        address = 0;
        data_in = 0;
        mem_read = 0;
        mem_write = 0;
        // Wait for global reset
        #50
        // Test Case 1: Write to Memory
        address = 10;
        data_in = 32'h12345678;
        mem_write = 1;
        mem_read=0;

        #50;
        mem_write = 0;
        mem_read=1;
        #20;
        // Test Case 2: Read from Memory
        mem_read = 1;
        #20;
        #20;
        $finish;
    end
```

Figure 7code and test bench for Data Memory

Test Bench Result:

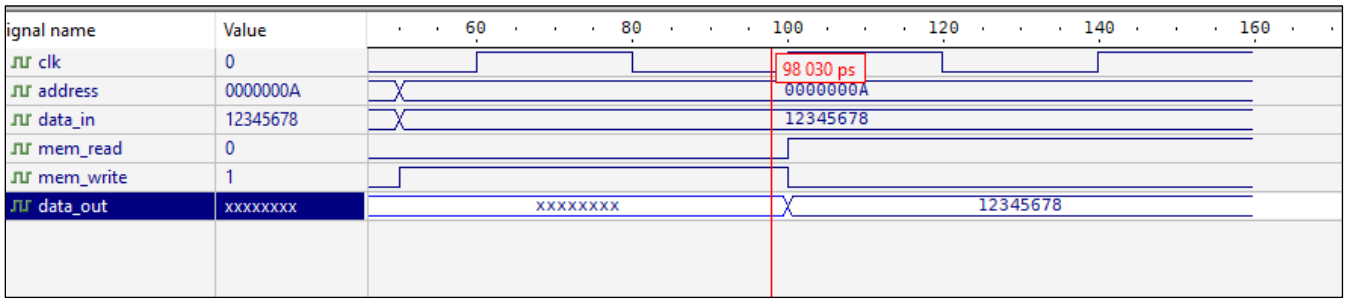

Figure 8waveform simulation for Data Memory

Read and Write operations done in the positive edge clock.

Code and Test bench for Instruction memory:

```
module Inst_mem(
    input [31:0] address,
    output reg [31:0] inst
    );

    parameter MEM_SIZE = 1024;
    parameter WORD_SIZE = 32;

    reg [WORD_SIZE-1:0] mem [0:MEM_SIZE-1];

    initial begin

        mem[0] = 32'b00111110010000000000000000000000;    // PUSH R9 (Push the value of R9 on t
        mem[1] = 32'b01000000000000000000000000000000;   // POP R0 (Pop the value on top of the
        mem[2] = 32'b00000100010000001000000000000000;  // ADD R1, R0, R2
        mem[3] = 32'h0D44001B;                          // andi $5,$1,6     (I-Type)
        mem[4] = 32'h1E540043;                         //SW $9,$5,16  (unused mode bits 11)  mem
        /*

        mem[0] = 32'b00111110010000000000000000000000;    // PUSH R9 (Push the value of R9 on t
        mem[1] = 32'b01000000000000000000000000000000;   // POP R0 (Pop the value on top of the
        mem[2] = 32'b00000100010000001000000000000000;  // ADD R1, R0, R2
        mem[3] = 32'h19C40021;                          // LW.POI R7,R1,8    (Load mem(R1 + sign_
        mem[4] = 32'h158C0010;                         // LW R6,R3,4
        mem[5] = 32'h090C8000;                        // sub R4,R3,R2
        mem[6] = 32'h0D44001B;                       // andi $5,$1,6    (I-Type)
        mem[7] = 32'h1E540043;                      //SW $9,$5,16  (unused mode bits 11) mem[R5+

        mem[9] = 32'h34000004;                  // CALL 4 (Pushes PC+1 to the top of the stack
        mem[9] = 32'h38000000;              // RET  (return PC to the value stored on top of the
        mem[10] = 32'h33FFFFFD;                  // JMP -3 (constant represented in 2's com
        */

    end


    always @(address) begin
        inst <= mem[address];
    end
```

```
module instruction_memory_tb;
    reg [31:0] addr;
    wire [31:0] data_out;

    Inst_mem mem(addr, data_out);

    initial begin
        addr = 0;
        #20;
        $display("Instruction = %h", data_out);
        #10;
        addr = 1;
        #10;
        $display("Instruction = %h", data_out);
        #10;
        addr = 2;
        #10;
        $display("Instruction = %h", data_out);
        #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 9code and test bench for instruction memory

Test Bench Result:

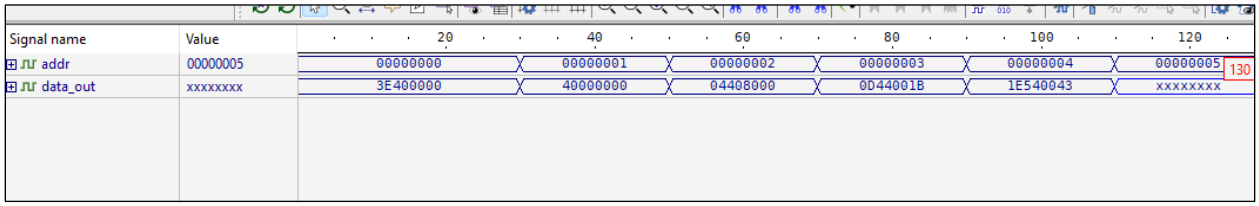

Figure 10 test bench for instruction memory

ALU:

In addition to performing arithmetic and logical operations, the Arithmetic Logic Unit (ALU) also plays a crucial role in setting flags that help control the flow of program execution and make decisions in a computer's central processing unit (CPU). These flags typically include the Zero Flag (Z), Carry Flag (C), Overflow Flag (V), and Sign Flag (S). The Zero Flag is set when the result of an operation is zero, the Carry Flag helps manage carry or borrow during addition and subtraction, the Overflow Flag signals when arithmetic overflow occurs, and the Sign Flag represents the sign of the result (positive or negative). These flags are used in conditional branching and decision-making instructions, allowing the CPU to respond to different conditions and perform various actions based on the results of ALU operations.

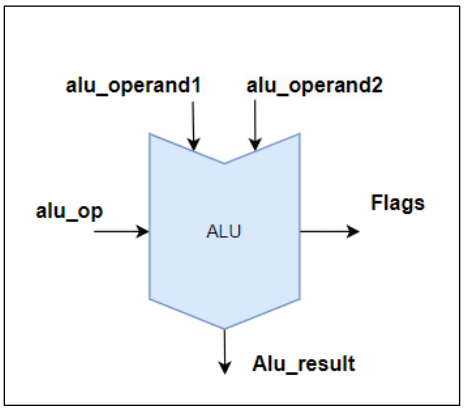

Figure 11 ALU

Code and Test bench for ALU:

```
module ALU(
    input [1:0] alu_op,
    input [31:0] data1,
    input [31:0] data2,
    output reg [31:0] alu_res,
    output reg C,
    output wire N, V, Z // negative, overflow, and zero
    );


    always @(*) begin
        case (alu_op)
            2'b00:  {C, alu_res} = data1 + data2;
            2'b01:  {C, alu_res} = data1 - data2;
            2'b10:  {C, alu_res} = data1 & data2;
        endcase
    end

    assign  Z = ~(|alu_res); // Zero flag
    assign  V = (alu_res[31] ^ C); // Overflow flag
    assign  N = alu_res[31]; // Negative flag


endmodule
```

```
module alu_tb;
  // Signals
  reg [1:0] alu_op;
  reg [31:0] a, b;
  wire [31:0] out;
  wire C, N, V, Z;
  // Instantiate ALU module
  ALU uut (
    .alu_op(alu_op),
    .data1(a),
    .data2(b),
    .alu_res(out),
    .C(C),
    .N(N),
    .V(V),
    .Z(Z)
  );
  initial begin
    a = 32'h0000000A; // 10 in hexadecimal
    b = 32'h00000005; // 5 in hexadecimal
    alu_op = 2'b00; // Addition
    #10;
    $display("a = %h, b = %h, operation = %h, out = %h", a, b, alu_op, out);
    alu_op = 2'b01; // Subtraction
    #10;
    $display("a = %h, b = %h, operation = %h, out = %h", a, b, alu_op, out);
    a = 32'h00001112;
    b = 32'h00000004;
    alu_op = 2'b10; // AND
    #10;
    $display("a = %h, b = %h, operation = %h, out = %h", a, b, alu_op, out);
    $finish;
  end
endmodule
```

Figure 12 code and test bench for ALU

Test Bench Result:

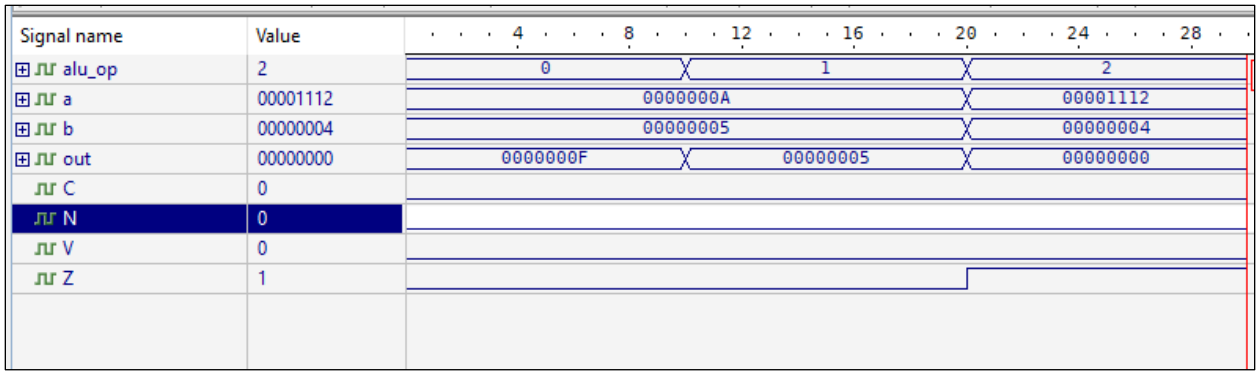

Figure 13 ALU test-bench

Register File

Register File architecture within a computer system. It comprises three registers: RA, RB, and RW, each with a 4-bit width. These registers are connected to two primary data buses, BusA and BusB, both of which have a width of 32 bits. This architecture enables reading from and writing to these registers through BusW2 and BusW2 when their signal values one.

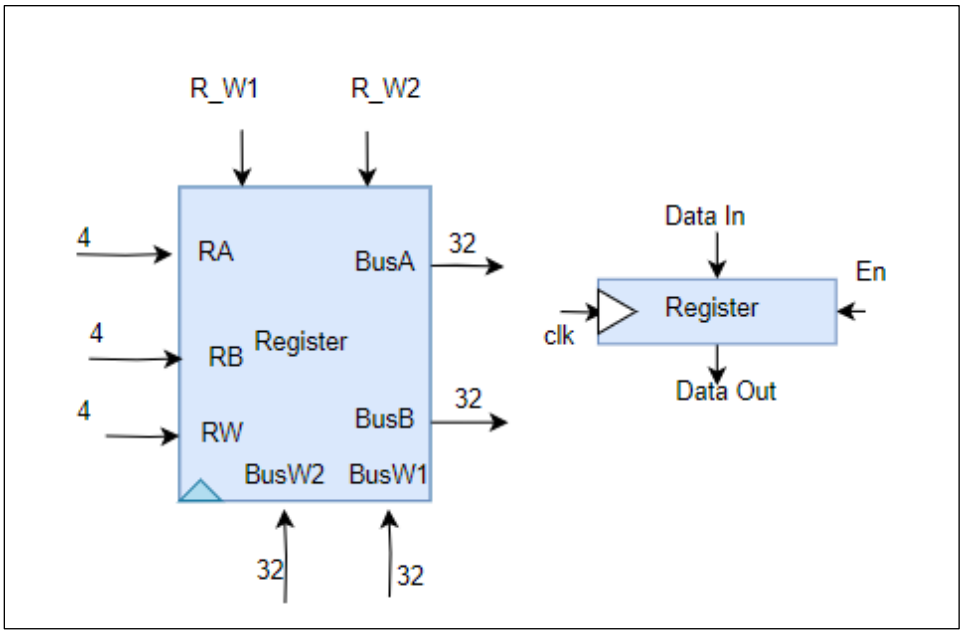

Figure 14 Register file

Code and Test bench for Register Filer:

```
module register_file(
    input clk,
    input [3:0] RA_address,
    input [3:0] RB_address,
    input [3:0] RW_address,
    input [31:0] busW1,
    input [21:0] busW2,
    output reg [31:0] busA,
    output reg [31:0] busB,
    input reg_write1,
    input reg_write2
    );

    reg [31:0] regs [15:0];

    always @(posedge clk) begin

         busB <= regs[RB_address];
         busA <= regs[RA_address];

        if (reg_write1) begin
             regs[RW_address] <= busW1;
        end

        if (reg_write2 ) begin
             regs[RA_address] <= busW2;
        end

    end

    initial begin

        regs[0] <= 32'h00000000;
        regs[1] <= 32'h00000001;
        regs[2] <= 32'h00000002;
        regs[3] <= 32'h00000003;
        regs[4] <= 32'h00000004;
        regs[5] <= 32'h00000005;
        regs[6] <= 32'h00000006;
        regs[7] <= 32'h00000007;
        regs[8] <= 32'h00000008;
        regs[9] <= 32'h00000009;
        regs[10] <= 32'h0000000A;
        regs[15] <= data_mem_size ;
    end

endmodule
```

```
module register_file_tb;

    // Inputs
    reg clk;
    reg [3:0] RA_address;
    reg [3:0] RB_address;
    reg [3:0] RW_address;
    reg [31:0] busW1;
    reg [31:0] busW2;
    reg reg_write1;
    reg reg_write2;

    // Outputs
    wire [31:0] busA;
    wire [31:0] busB;

    // Instantiate the Unit Under Test (UUT)
    register_file uut (
        .clk(clk),
        .RA_address(RA_address),
        .RB_address(RB_address),
        .RW_address(RW_address),
        .busW1(busW1),
        .busW2(busW2),
        .busA(busA),
        .busB(busB),
        .reg_write1(reg_write1),
        .reg_write2(reg_write2)
    );

    // Clock generation
    always #10 clk = ~clk;

    // Test procedure
    initial begin
        // Initialize Inputs
        clk = 0;
        RA_address = 0;
        RB_address = 0;
        RW_address = 0;
        busW1 = 0;
        busW2 = 0;
        reg_write1 = 0;
        reg_write2 = 0;

        // Wait 100 ns for global reset to finish
        #50;

        // Add stimulus here
        // Example: Writing to a register and reading from it
        reg_write1 = 1;
        RW_address = 4;
        busW1 = 32'hA5A5A5A5;
        #20;
        reg_write1 = 0;
        RA_address = 4;
```

Figure 15 code and test bench for Register File

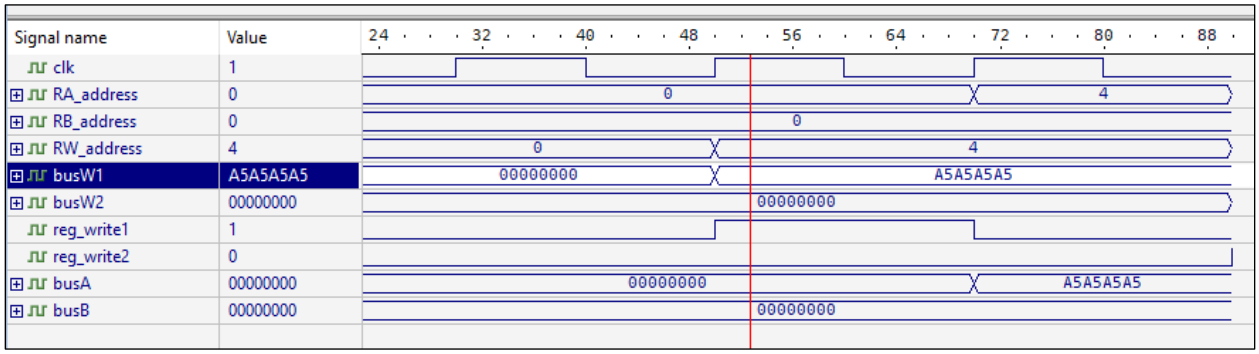

Figure 16 Register File wave form

Extenders

The extender is versatile in that it can support both signed and unsigned data representations. This implies that it can handle data values that are interpreted as either signed integers (with a sign bit) or unsigned integers (without a sign bit). Depending on the specific implementation and configuration, the extender would appropriately sign-extend or zero-extend the 16-bit input to a 32-bit output.

- Sign-extension for signed data: If the 16-bit input represents signed data, the extender would copy the sign bit (the most significant bit) to fill the additional 16 bits to create a valid 32-bit signed value.
- Zero-extension for unsigned data: If the 16-bit input represents unsigned data, the extender would simply pad the lower 16 bits with zeros to create a 32-bit unsigned value.

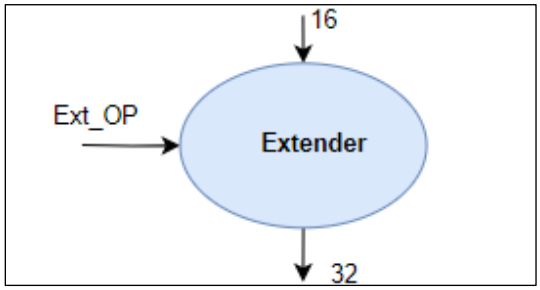

Figure 17 Extender

Code and Test bench for Extender:

```
module extender(
    input [15:0] A,
    output reg [31:0] B,
    input extend_op
    );

    always @(*) begin

        if (extend_op == 0)
              B <= {16'b0, A[15:0]}; // unsigned immedia
        else
              B <= {{16{A[15]}}, A[15:0]}; // signed ext

    end
```

```
module extender_tb;

    // Inputs
    reg [15:0] A;
    reg extend_op;

    // Outputs
    wire [31:0] B;
    extender uut (
        .A(A),
        .B(B),
        .extend_op(extend_op)
    );

    // Test procedure
    initial begin
        // Initialize Inputs
        A = 0;
        extend_op = 0;

        #10;
        A = 16'h1234;
        extend_op = 0; // Zero extension
        #20;


        // Test Case 3: Sign Extension with negative value
        A = 16'hF234; // MSB is 1, indicating a negative value
        extend_op = 1; // Sign extension
        #20;

        $finish;
    end

endmodule
```

Figure 18 code and test bench for extender

Test Bench Result:

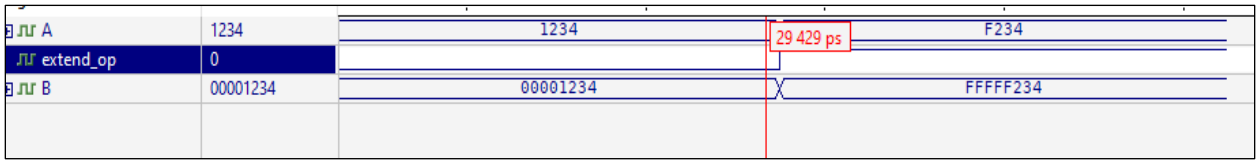

Figure 19 simulation for extender

Multiplexers

A multiplexer is a combinational digital circuit that chooses one of 2^n input lines and feeds it to a single output line based on their values. The number of selection lines, often known as "select lines" or "control lines," specifies how many input lines may be chosen. There are two types of multiplexers used on the Datapath 2x1Mux and 4x1 Mux.

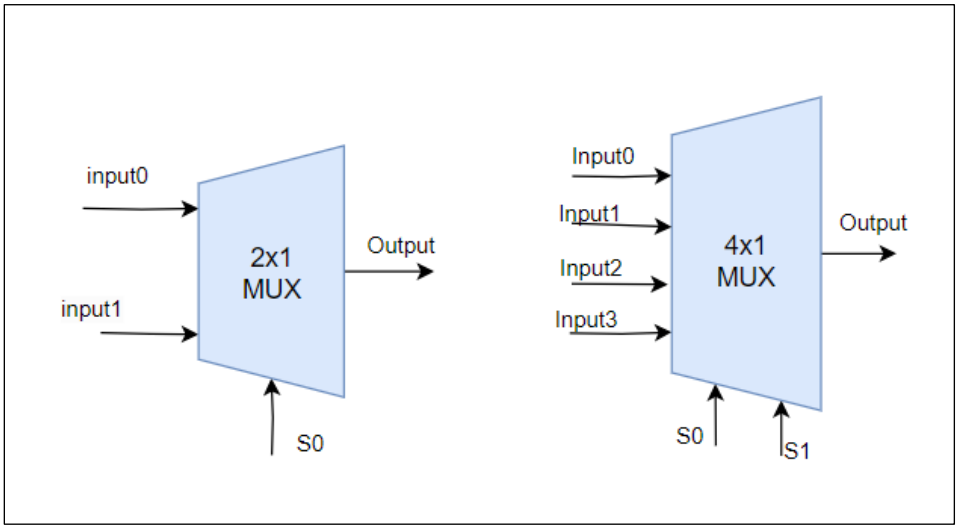

Figure 20Multiplexers

Six of 2x1Mux used in the data path design:

1- The selection of the source register for BusA is determined by the value of "sel_RA," a control line generated by the main control unit. The multiplexer (MUX) at this stage offers a choice between two possible values: Rs1 and sp_index, and it selects one of them based on the value of "sel_RA ".

2- The selection of the source register for BusB is determined by the value of "sel_RB," a control line generated by the main control unit. The multiplexer (MUX) at this stage offers a choice between two possible values: Rs2 and Rd and it selects one of them based on the value of "sel_RB ".

3- The selection of the alu_operand2 is made based on the value of "sel_aluOp." This choice determines whether the second operand for the ALU (Arithmetic Logic Unit) will be either an immediate value or the value from BusB.
4- The selection of data to be written on Register file from BusW1, selected from memory data out or ALU result by the signal “write_back”.
5- The selection of data to be written on Register file from BusW2, selected from BusA +1 or from BusA value -1 by the signal “sel_BusW2”.
6- The selection of the data in for memory is made based on the value of "M_DIN." This choice determines whether the data in for the Memory will be either an PC+1 or BusB value.

Two of 4x1Mux used in the data path design:

1- To select the pc value from four choices branch target address, jump address, PC+1 and the PC value for the RET instruction this done by the “pc_src” signal.
2- To select the address for the memory which has three choices ALU result, SP value and SP-1 value this done by the “sel_mem_address”.

Instruction Register and Program counter

IR: 32-bit register stores instructions fetched during the fetch stage.

PC: The Program Counter is a special-purpose register in a CPU (Central Processing Unit) that keeps track of the memory address of the next instruction to be fetched and executed.

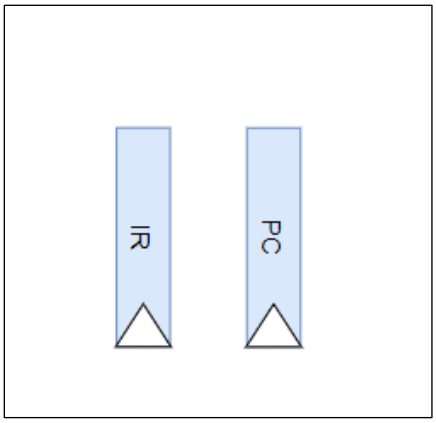

Figure 21 IR and PC

State diagram

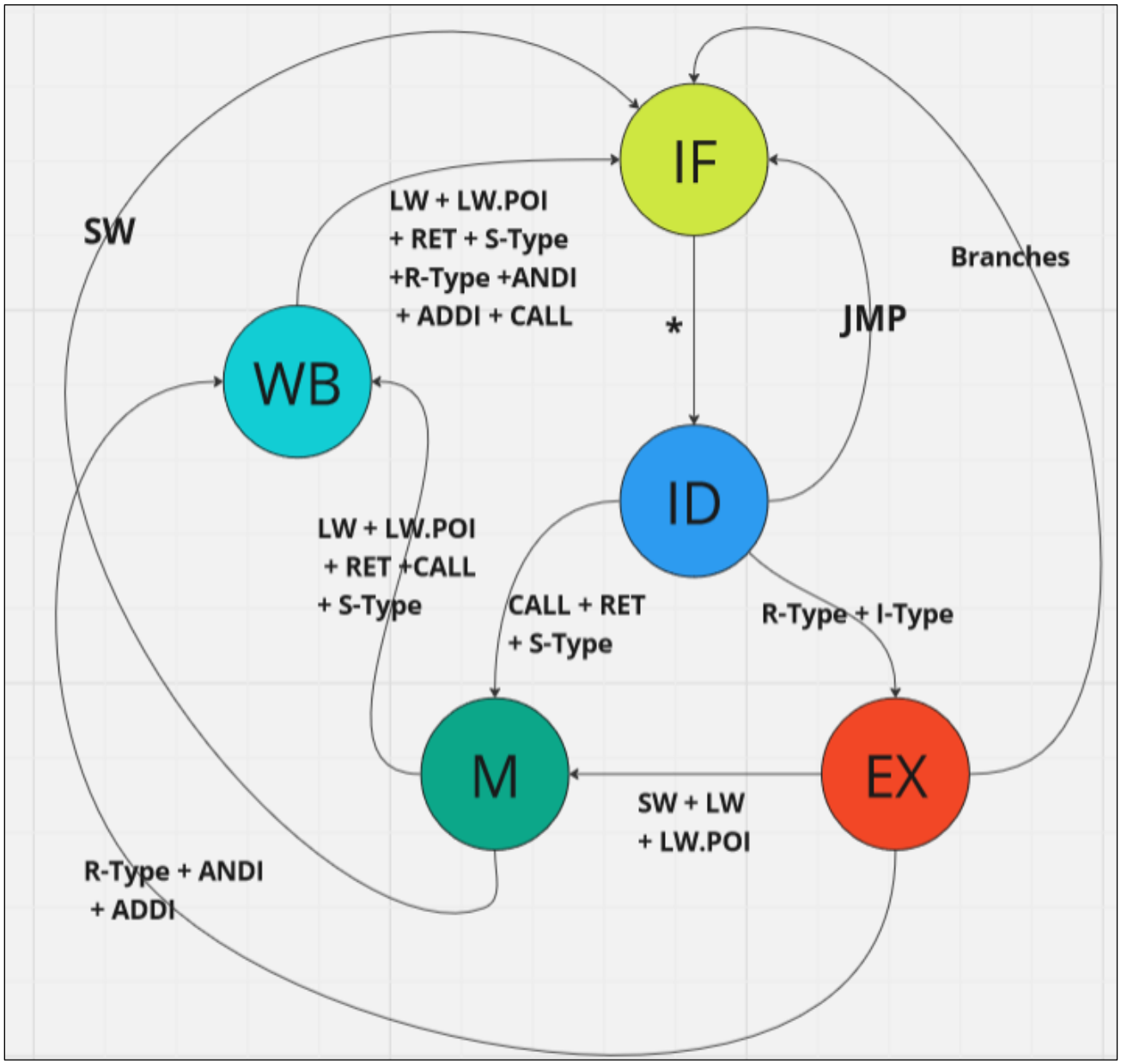

Figure 22: state diagram

DATA PATH

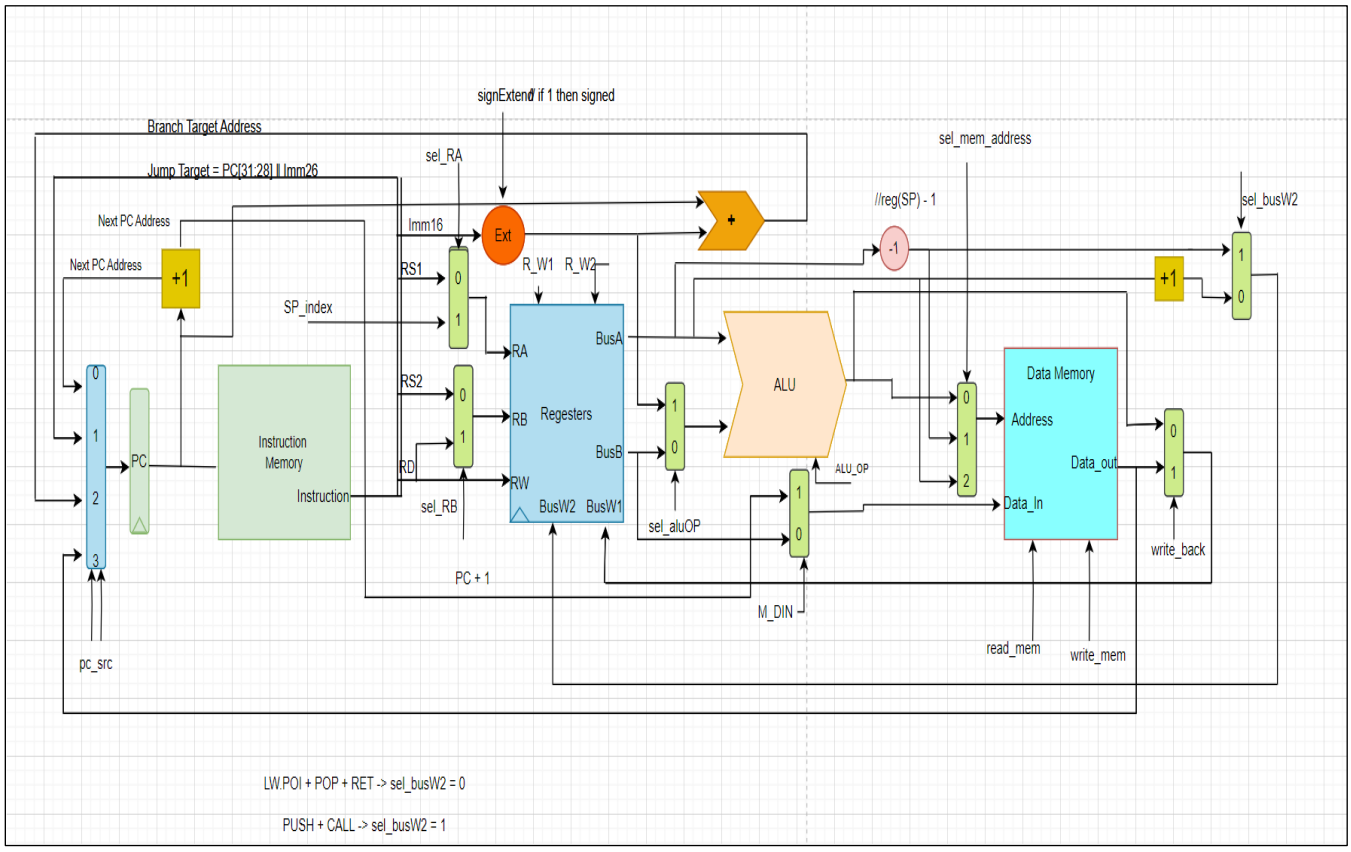

Figure 23: Data Path

Design Procedure

Main Control Signals

Main Control Input

1. Z (Zero Flag)
2. V (Overflow Flag)
3. C (Carry Flag)
4. N (Negative Flag):
5. opcode [5:0]:

- 6-bit opcode specifying the type of operation to be performed.

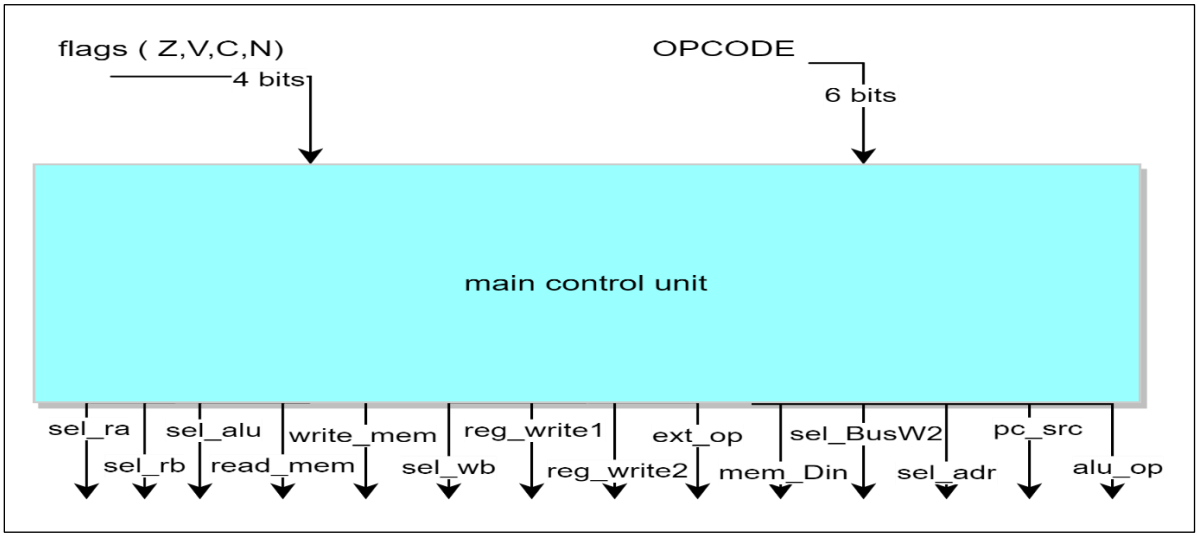

Figure 24: Control Unit

Main Control Signals

sel_RA	When 0 →First operand is RS1 (to be read from RF) When 1 → First operand is SP_index(to be read from RF)
sel_RB	When 0 →Second operand is Rs2 (to be read from RF) When 1 →Second Operand is Rd (to be read from RF)

sel_alu_operand	When 0→ Second ALU operand is the value of register (Rs2/Rd) that appears on BusB When 1 →Second ALU operand is the value of the extended 16-bit immediate
sel_address_mem [1:0]	When 00→The ALU result is the Address to the data memory When 01 →The minus 1 from the value of register RA (SP_INDEX) that appears on BusA is the Address to the data memory When 10 → The value of register RA (SP_INDEX) that appears on BusA is the Address to the data memory
read_mem	When 0 →Data Memory is NOT read When 1→ Data Memory is read: Data_Out ← Memory [Address]
write_mem	When 0 → Data Memory is NOT written When 1 → Data Memory is written: Mem [Address] ← Data_in
sel_wb_data	When 0 → The ALU result will be written on the register rd: BusW1 = ALU result When 1 → Data from memory will be written on the register rd: BusW1 = Data_out
reg_write1	When 0→ no write in the rd Register When 1→ Destination Register Rd is written with the data on BusW1
reg_write2	When 0→ no write in the RA (RS1 OR SP_INDEX) Register When 1→ RA (RS1 OR SP_INDEX) Register is written with the data on BusW2
extend_op	When 0 →16-bit immediate is Zero-Extended When 1 →16-bit immediate is Sign-Extended
mem_Din	When 0 →The (data in) to the data memory is pc +1 (when the instruction is call)

	When 1 → The (data in) to the data memory is the value of register (Rs2/Rd) that appears on BusB
sel_BusW2	When 0 →The plus of the Data on the BusA will be written on the register RA (RS1 OR SP_INDEX),(BusW2 = BusA +1) When 0 →The plus of the Data on the BusA will be written on the register RA (RS1 OR SP_INDEX),(BusW2 = BusA +1)
alu_op [1:0]	When 00 → ADD operation will be performed in the ALU When 01→ SUB operation will be performed in the ALU When 10 → AND operation will be performed in the ALU

Table 1: Main Control Signals

Main Control Truth Table

R_TYPE	sel_RA	sel_rb	sel_aluOPR	adr_mem	read_mem	write_mem	write_back_data	reg_write1	reg_write2	pc_src	EXT_OP	M_Din	sel_busW2
AND_000000	0	0	0	X	0	0	0	1	0	0	X	x	X
ADD_000001	0	0	0	X	0	0	0	1	0	0	X	x	X
SUB_000010	0	0	0	X	0	0	0	1	0	0	X	x	X
I TYPE INSTRUCTIONS													
ANDI_000011	0	x	1	X	0	0	0	1	0	0	0	x	X
ADDI_000100	0	X	1	X	0	0	0	1	0	0	1	x	X
LW_000101	0	X	1	0	1	0	1	1	0	0	1	x	X
LW.POI_000110	0	X	1	0	1	0	1	1	1	0	1	x	0
SW_000111	0	X	1	0	0	1	X	0	0	0	1	0	X
BGT_001000	0	1	0	X	0	0	X	0	0	(Z == 0 && N == V) ? 2 : 0	1	x	X
BLT_001001	0	1	0	X	0	0	X	0	0	N != V ? 2 : 0	1	x	X
BEQ_001010	0	1	0	X	0	0	X	0	0	(Z == 1) ? 2	1	x	X
BNE_001011	0	1	0	X	0	0	X	0	0	(Z == 0) ? 2 : 0	1	x	X

Table 2: Main Control Truth Table

J TYPE INSTRUCTIONS													
JMP 001100	x	x	x	x	0	0	x	0	0	1	1	x	X
CALL 001101	1	x	x	1	0	1	x	0	0	1	1	1	1
RET 001110	1	x	x	2	1	0	0	0	0	3	x	x	0
S TYPE INSTRUCTIONS													
PUSH 001111	1	1	x	1	0	1	0	0	1	0	x	0	1
POP 010000	1	1	x	2	1	0	1	1	1	0	x	x	0

Logic Equation for Main Control (control_unit module)

```
module control_unit(
    input Z, V, C, N, // flags, needed for Branch conditions (for PC source signal)
    input [5:0] opcode,
    output reg sel_RA, sel_RB, sel_alu_operand, read_mem, write_mem, sel_wb_data, reg_write1, reg_write2, extend_op, mem_Din, sel_BusW2,
    output reg [1:0] sel_address_mem, pc_src, alu_op
    );
                                        // conditions for LT and GT are swapped here, since it's RD (>/</==) RS1 (we do RS1 - RD,..)
    reg branch_taken;
    assign  branch_taken = (opcode == BEQ && Z) || (opcode == BNE && !Z) || (opcode == BLT && !Z && N == V) || (opcode == BGT && N != V);
    assign  sel_RA = (opcode == CALL || opcode == RET || opcode == PUSH || opcode == POP);
    assign  sel_RB = ~(opcode == AND || opcode == ADD || opcode == SUB);
    assign  sel_alu_operand = (opcode == ANDI || opcode == ADDI || opcode == LW || opcode == LW_POI || opcode == SW);
    assign  sel_address_mem = ( (opcode == RET || opcode == POP) ? 2 : ( opcode == CALL || opcode == PUSH) ? 1 : 0 );
    assign  read_mem = (opcode == LW || opcode == LW_POI || opcode == POP || opcode == RET);
    assign  write_mem = (opcode == SW || opcode == CALL || opcode == PUSH);
    assign  sel_wb_data = (opcode == LW || opcode == LW_POI || opcode == POP);
    assign  reg_write1 = (opcode == ADD || opcode == AND || opcode == SUB || opcode == ANDI || opcode == ADDI || opcode == LW || opcode == LW_POI || opcode == POP);
    assign  reg_write2 = (opcode == LW_POI || opcode == POP || opcode == PUSH);
    assign  extend_op = (opcode != ANDI);
    assign  mem_Din = (opcode == CALL);
    assign  pc_src = (opcode == JMP || opcode == CALL) ? 1 :
                (opcode == RET) ? 3 :
                (branch_taken) ? 2 : 0;

    assign  alu_op = ((opcode == AND || opcode == ANDI ) ? 2 : (opcode == SUB || opcode == BEQ || opcode == BLT || opcode == BGT || opcode == BNE) ? 1 : 0);
    assign sel_BusW2 = (opcode == PUSH || opcode == CALL) ? 1 : 0;
                //  (opcode == LW_POI || opcode == POP || opcode == RET) ? 0 : 0;

endmodule
```

Figure 25: control unit module

The Test Bench When sets opcode to different values (6'b000000 = AND, 6'b000001 = ADD, 6'b000011 = ANDI, 6'b001000 = BGT)

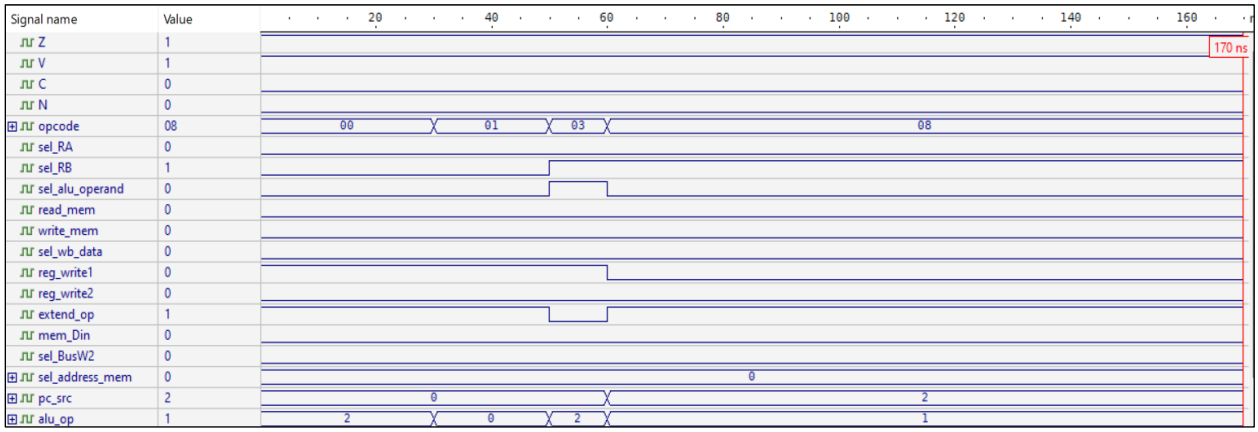

Figure 26: test bench (control unit)

PC Control Unit

opCode	Z	N	V	PC_src
R_type && ANDI && ADDI&&LW&&LW.POI	X	X	X	0
JMP && CALL	X	X	X	1
BNE	0	X	X	2
BEQ	1	X	X	2
BNE	0	X	X	2
BLT	0	N == V	V == N	2
BGT	X	N != V	V != N	2
RET	X	X	X	3

Table 3: PC_TABLE

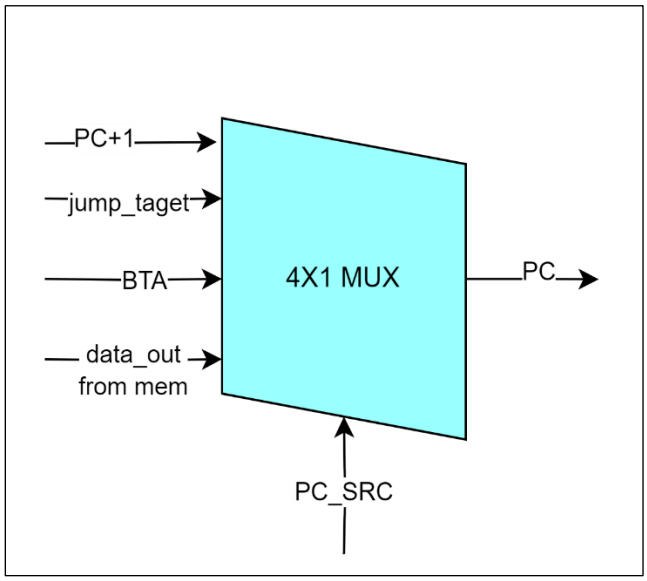

Figure 27: PC Source Signal Block

Testing

Sample Instructions for testing

```
mem[0] = 32'b00111110010000000000000000000000;    // PUSH R9 (Push the value of R9 on the top of the stack)
mem[1] = 32'b01000000000000000000000000000000;    // POP R0 (Pop the value on top of the stack to register R0)
mem[2] = 32'b00000100010000001000000000000000;  // ADD R1, R0, R2
mem[3] = 32'h19C40021;                           // LW.POI R7,R1,8    (Load mem(R1 + sign_extend(8)) to R7)
mem[4] = 32'h158C0010;                          // LW R6,R3,4
mem[5] = 32'h090C8000;                         // sub R4,R3,R2
mem[6] = 32'h0D44001B;                        // andi $5,$1,6    (I-Type)
mem[7] = 32'h1E540043;                       //SW $9,$5,16  (unused mode bits 11)   mem[5+16 =21] <= 9
mem[8] = 32'h33FFFFFD;                      // JMP -3 (constant represented in 2's complement)
mem[9] = 32'h34000004;                     // CALL 4 (Pushes PC+1 to the top of the stack and next_pc = {pc[31:26], sign_extend(4))
mem[10] = 32'h38000000;                   // RET  (return PC to the value stored on top of the stack)
```

Figure 28:instructions in memory

Data memory locations hold the same value as the address of each location.

PUSH Instruction

Instruction: PUSH R9

As we can see, first the stack pointer is incremented to point to the new value we want to push into the stack, then the value of R9 (which is 9) is pushed on top of the stack.

Initially, the value of the SP (R15) is 1024 (which is the highest memory address in the data memory).

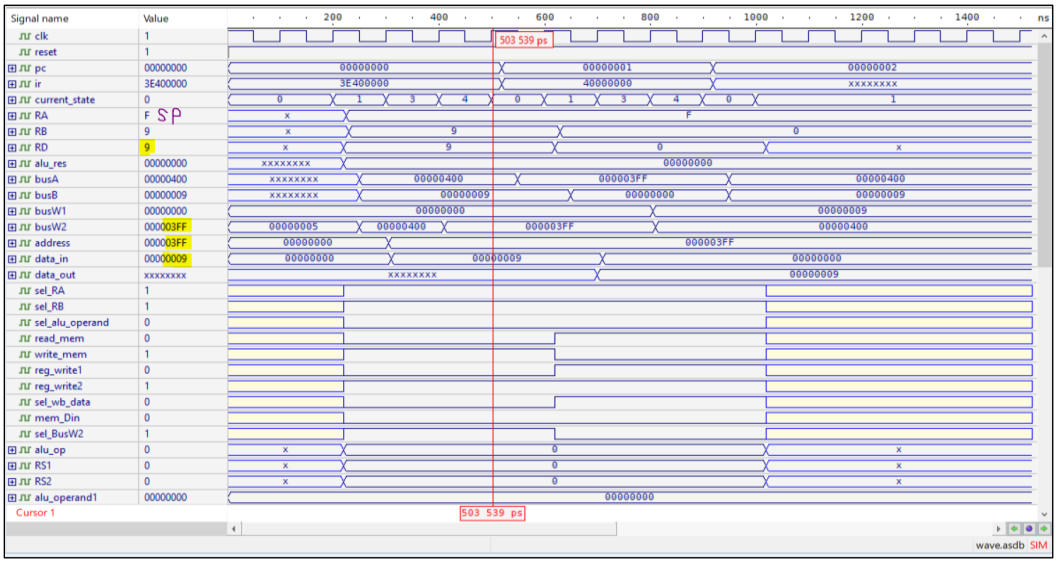

Figure 29 push

As we can see, busW2 holds the new value that will be written back to the stack-pointer register.

And data_in holds the value of the register we pushed to the stack (R9 = 9).

POP Instruction

Instruction: POP R0

After Pushing R9 to the stack, we popped the value on top of the stack to R0 to see if the push instruction saved the value of R9 into the memory.

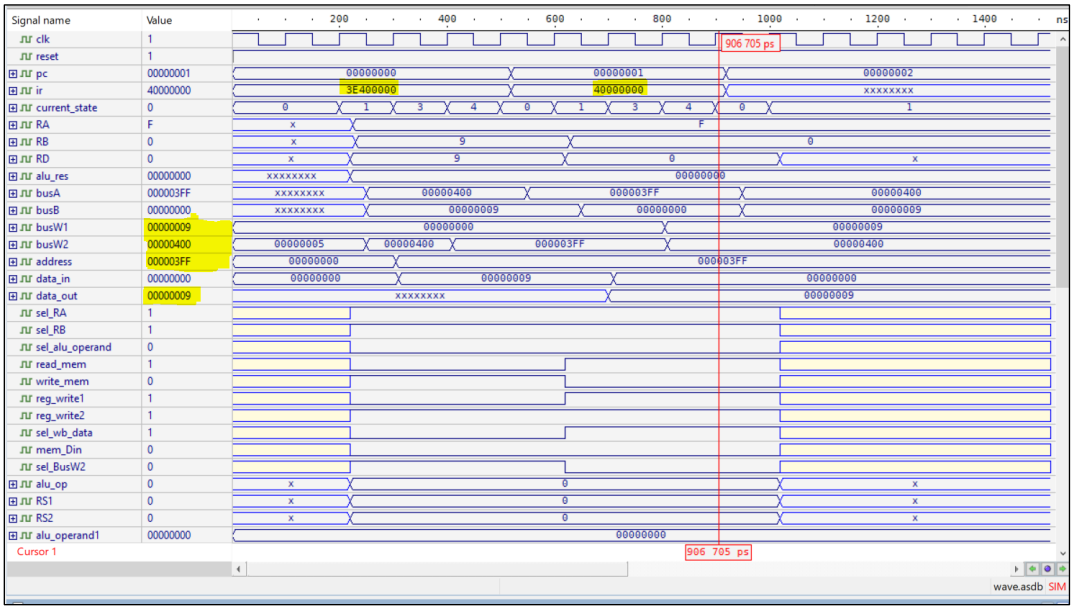

Figure 30 POP

We can see the data_out is 9, which is the value we popped from the stack (same value we pushed in the previous instruction).

Notice how busW2 is now 0x400 (1024), this value is written as the new value of the stack pointer (it was 1023 before the pop instruction).

Also look at busW1, it holds the value 9, which is the value we pushed to the stack in the previous instruction, this value is written into register R0.

ADD Instruction

Instruction: ADD R1, R0, R2

R0 holds the value 9, because we popped 9 from top of the stack to R0 – is added with R2 (which holds the value 2).

The result is shown as alu_res = 0x0B (11 in decimal).

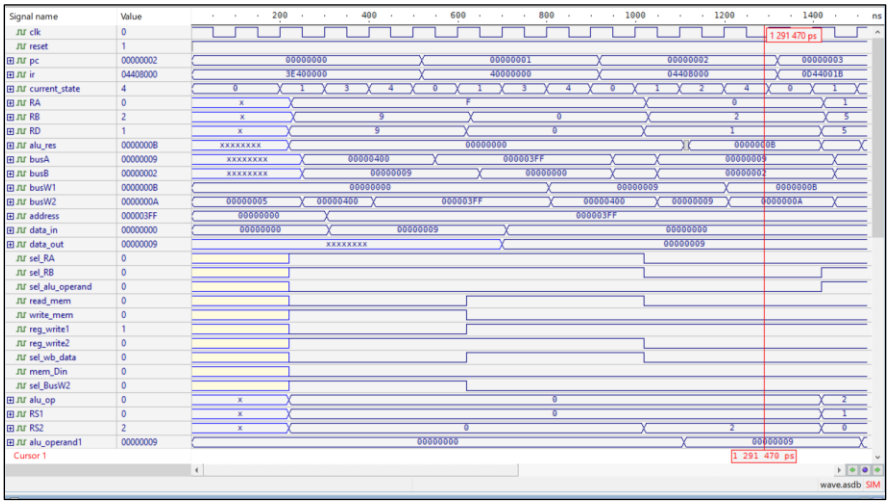

Figure 31 ADD

Notice busW1 which holds the ALU result that will be written to the register R1.

Also look at the signal's values.

ANDI Instruction

Instruction: ANDI R5, R1, 6

The value of R1 (which is 0xB, the result of the previous instruction) is &-ed with the immediate value 0x06 (alu_operand2).

See the result in alu_res = 0x02.

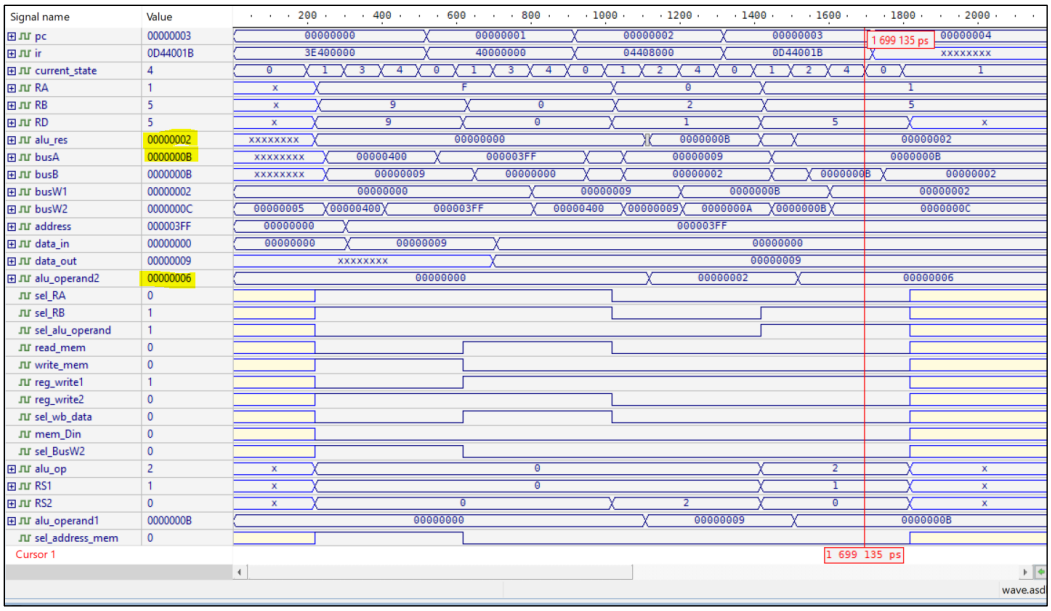

Figure 32 ANDI

SW Instruction

Instruction: SW R5, R1, 16

Stores the value in R5 into the memory address (R1 + sign_extend(16))

busA = 0x0B which is the value in R1 (RA in the diagram), is added with 16 (0x10), the result is 0x1B,

which is the store address. The data_in is the value of RD, which is 0x02.

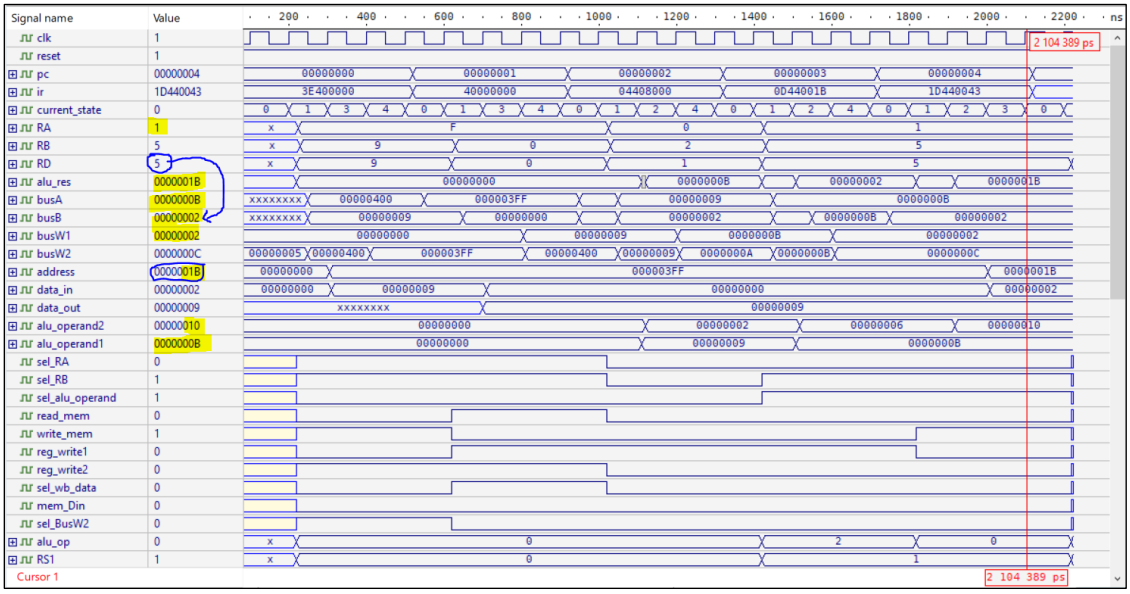

Figure 33 SW

LW Instruction

Instruction: LW R6, R3, 0x018

The value stored in the memory address (0x018 + R3 = 0x1B) is 2 (from the previous instruction).

This value is loaded into register R6. Notice the values of busW1 and address in the figure below.

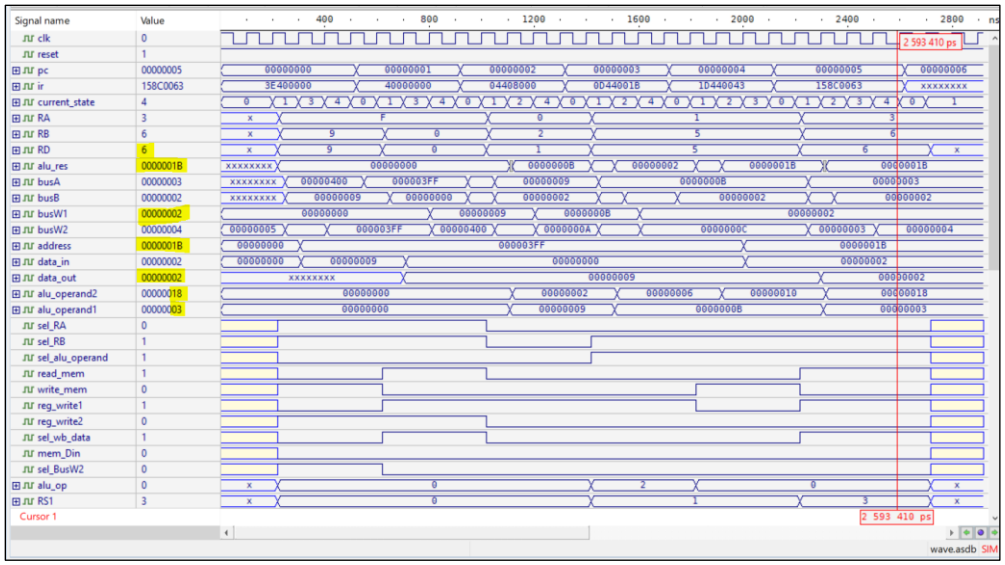

Figure 34 LW

LW.POI

Instruction: LW.POI R6, R3, 0x018

The value stored in the memory address (0x018 + R3 = 0x01B) is 2 (from the previous store instruction).

The value is loaded into register R6, and the value of R3 is then incremented by 1. Look at busW1 and busW2 (busW1 writes to RD->R6, and busW2 writes to RA->R3).

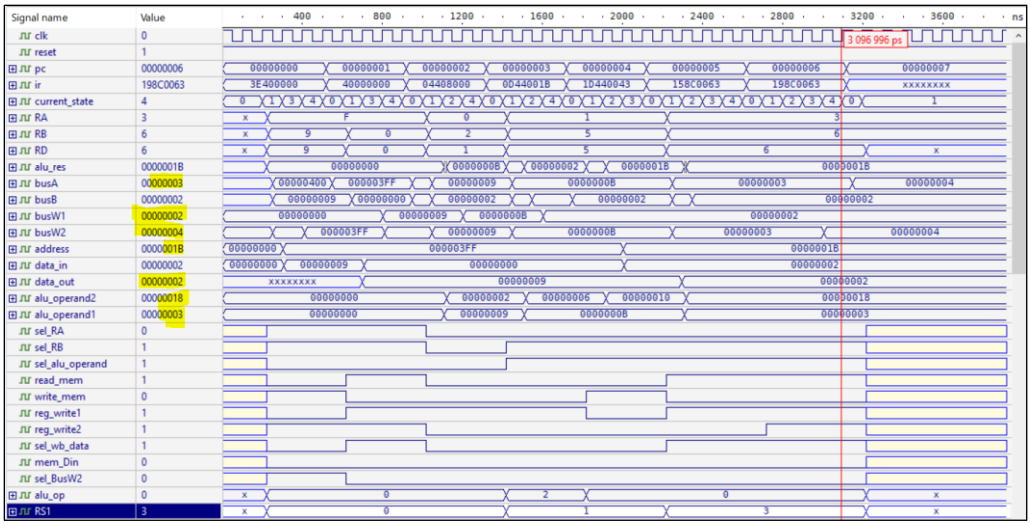

Figure 35 LW.POI

BGT Instruction

Instruction: BGT R4, R2, -1 (0xFFFF)

Notice that busA holds the value of R2, busA holds the value of R4.

R4 is greater than R2, and we subtracted R2 – R4, and the result is negative, so the branch is taken.

Note the value on BTA (Branch Target Address), this value will be the next PC value.

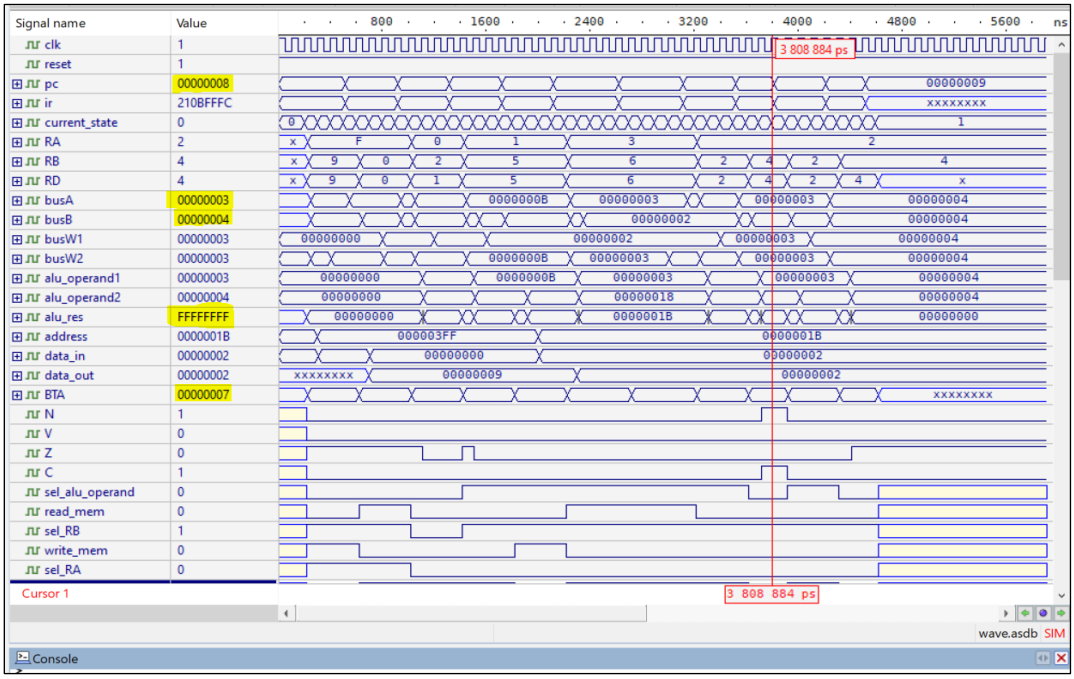

Figure 36 BGT

CALL Instruction

Instruction: CALL 7 (will go into address 7 in memory)

Notice the value of the next PC in the waveform in the figure below.

The value of PC+1 is shown in data_in, and it is pushed to the stack in memory address 0x3FF, and the SP is updated from 0x400 (1024) into 0x3FF (1023).

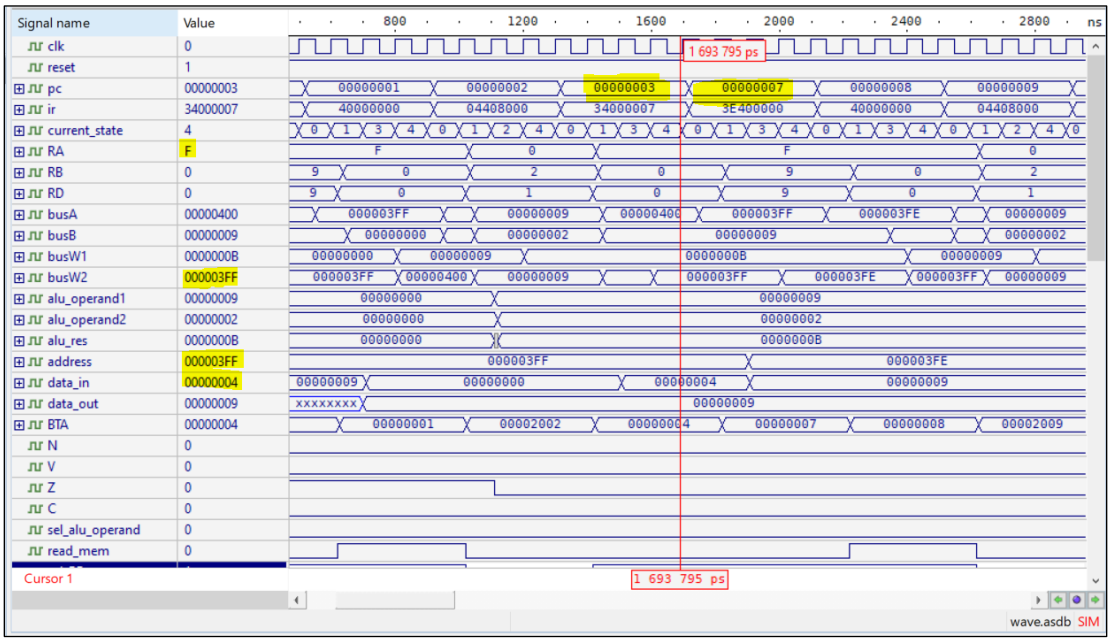

Figure 37 CALL

RET Instruction

Instruction: RET (set the next PC into the value on top of the stack).

The value on top of the stack is popped from the stack into the PC (notice the next instructions' PC value), and the value of the SP is then incremented by 1 (0x3FF into 0x400).

data_out = 4 is the value of the next PC, busW2 = 0x400 is the new SP value after pop.

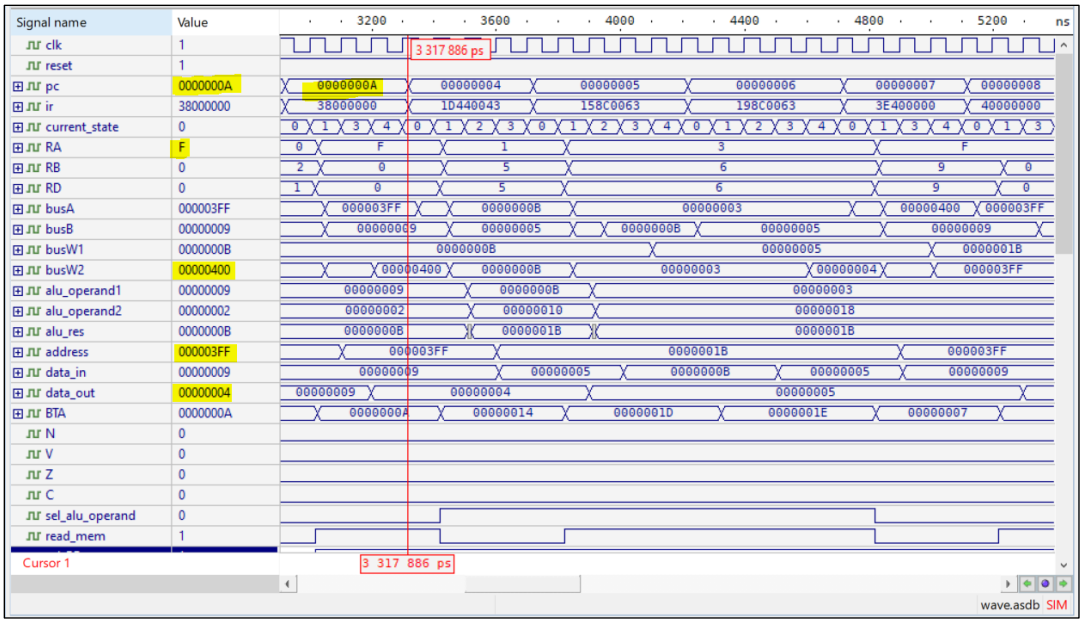

Figure 38 RET

Team Work

We worked as an integrated team on all parts of the project, from designing to writing the code and finally testing

Conclusion

The project's objective has been successfully achieved as we completed the design, implementation, and verification of a RISC processor using VERILOG HDL, following a multi-cycle CPU approach. Our methodology involved categorizing instructions that share the same Datapath and subsequently writing the RTL for each group. We carefully derived the components and data transfer from the RTL, specifying all necessary Datapath components. The design process included the development of both the Datapath and Control Unit to ensure the accurate execution of the given instructions. Following the design phase, we implemented each component in VERILOG, conducting individual testing before integrating all components. The final phase involved comprehensive testing of groups of instructions to verify the overall functionality of our design. Remarkably, all instruction groups performed correctly without errors. This project has provided valuable insights into the implementation of a multi-cycle CPU, emphasizing the intricacies of deriving and interconnecting components to realize a specific ISA.