Department of Electrical & Computer Engineering
ENCS4370 - Computer Architecture

Multi Cycle Processor Implementation

Prepared By:
Amr Halahla 1200902
Manar Shawahni 1201086

Instructor: Dr. Ayman Hroub
Date: July 12, 2023

Table of Contents

List Of Figures

List Of Tables

Abstract

This report discusses the design, and implementation of a Multi Cycle Processor according to a given RISC instruction set. The design was done by analyzing each instruction, and notice main components it needs. The implementation was done by building the components, then derive the control signals for them. Testing was done for each instruction, and then by applying complete scenarios to the processor and validate the result.

Introduction

Objective

The objective of this project is to successfully design, model and simulate a MIPS Multi-Cycle Processor using Verilog HDL. The design approach was used where each sub-module of the processor was first designed, coded, and tested. Once all sub-modules were designed and determined to be fully functional, they were instantiated into a structural module to form our processor.

Introduction to RISC Machines

RISC is a type of CPU design in which it is believed that a simplified instruction set will enhance performance of the processor. It uses a small, highly-optimized set of instructions rather than a more complex set as found in other processors. The word "reduced" in the name refers to the amount of work for any single instruction set.
Typical features of RISC architecture include: fixed length, standard format, identical general-purpose registers, simple addressing modes, one cycle execution time, and pipelining.
Aside from quicker performance, RISC processor components are generally cheaper to design and produce as they utilize less transistors. RISC is the newer technology and is widely used in the industry compared to other processor types.

Multi Cycle Processors

A multi-cycle processor is a type of processor architecture that divides the execution of instructions into multiple stages or cycles. Each cycle performs a specific operation, allowing instructions to be executed efficiently and enabling more complex instructions to be processed. Using the Multi-cycle approach, different instruction may take different amounts of time to process unlike in the

Single-cycle approach where instruction processing is as fast as the slowest instruction.

In a multi-cycle processor, each instruction goes through several stages, with each stage completing within a single clock cycle. The stages typically include:

1. Instruction Fetch (IF): The processor fetches the instruction from memory using the program counter (PC) and increments the PC to point to the next instruction.

2. Instruction Decode (ID): The fetched instruction is decoded to determine the operation to be performed. This stage also involves fetching any necessary operands or data from registers.

3. Execution (EX): The actual operation specified by the instruction is performed in this stage. It may involve arithmetic calculations, logical operations, or address computations.

4. Memory Access (MEM): If the instruction requires accessing memory, such as loading or storing data, it is performed in this stage. Data is read from or written to memory.

5. Write Back (WB): The results of the previous stage are written back to the appropriate register(s). This stage updates the register file with the computed values.

Design Specifications and Implementation

Processor Properties

1. The instruction size is 32 bits.
2. 32 32-bit general-purpose registers: from R0 to R31.
3. A special purpose register for the program counter (PC).
4. It has a stack called control stack which saves the return addresses
5. Stack pointer (SP), another special purpose register to point to the top of the control stack. SP holds the address of the empty element on the top of the stack. For simplicity, you can assume a separate on-chip memory for the stack, and the initial value of SP is zero.
6. Four instruction types (R-type, I-type, J-type, and S-type).
7. The processor's ALU has an output signal called "zero" signal, which is asserted when the result of the last ALU operation is zero.
8. Separate data and instructions memories

Instruction Types and Formats

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

a) **2-bit instruction type** (00: R-Type, 01: J-Type, 10: I-type, 11: S-type)
b) **5-bit function,** to determine the specific operation of the instruction
c) **Stop bit,** which is the least significant bit of each instruction binary format, and it is used to mark the end of a function code block. In other words, if the value of this stop bit is "1", this means that this instruction is the last instruction of the function, and hence the execution control should return to the return address which is stored on the top of the control stack.

1. R-Type (Register Type) Formats
   - **5-bit Rs1:** first source register
   - **5-bit Rd:** destination register

- **5-bit Rs2:** second source register
- 9-bit unused

Function^5	Rs1^5	Rd^5	Rs2^5	Unused^9	Type^2	Stop^1

Figure 1: R-Type Format

2. I-Type (Immediate Type) Format
   - **5-bit Rs1:** first source register
   - **5-bit Rd:** destination register
   - **14-bit immediate:** unsigned for logic instructions, and signed otherwise

Function^5	Rs1^5	Rd^5	Immediate^{14}	Type^2	Stop^1

Figure 2: I-Type Format

3. J-Type (Jump Type) Format
   - 24-bit signed immediate: jump offset

Function^5	Signed Immediate^{24}	Type^2	Stop^1

Figure 3: J-Type Format

4. S-Type (Shift Type) Format
   - **5-bit Rs1:** first source register
   - **5-bit Rd:** destination register
   - **5-bit Rs2:** second source register. This register stores the shift amount in case the shift amount is variable and it is calculated at runtime
   - **5-bit SA:** the constant shift amount.
   - 4-bit unused

Function^5	Rs1^5	Rd^5	Rs2^5	SA^5	Unused^4	Type^2	Stop^1

Figure 4: S-Type Format

Instruction Set

- Table 1 shows the instructions supported by this instruction set, with their meaning and decoding.

No.	Instr	Meaning	Function Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	00000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	00001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	00010
4	CMP	zero-signal = Reg(Rs) < Reg(Rs2)	00011
I-Type Instructions			
5	ANDI	Reg(Rd) = Reg(Rs1) & Immediate^{14}	00000
6	ADDI	Reg(Rd) = Reg(Rs1) + Immediate^{14}	00001
7	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{14})	00010
8	SW	Mem(Reg(Rs1) + Imm^{14}) = Reg(Rd)	00011
9	BEQ	Branch if (Reg(Rs1) == Reg(Rd))	00100
J-Type Instructions			
10	J	PC = PC + Immediate^{24}	00000
11	JAL	PC = PC + Immediate^{24} Stack.Push (PC + 4)	00001
S-Type Instructions			
12	SLL	Reg(Rd) = Reg(Rs1) << SA^{5}	00000
13	SLR	Reg(Rd) = Reg(Rs1) >> SA^{5}	00001
14	SLLV	Reg(Rd) = Reg(Rs1) << Reg(Rs2)	00010
15	SLRV	Reg(Rd) = Reg(Rs1) >> Reg(Rs2)	00011

Table 1: Instruction Set

Finite State Machine

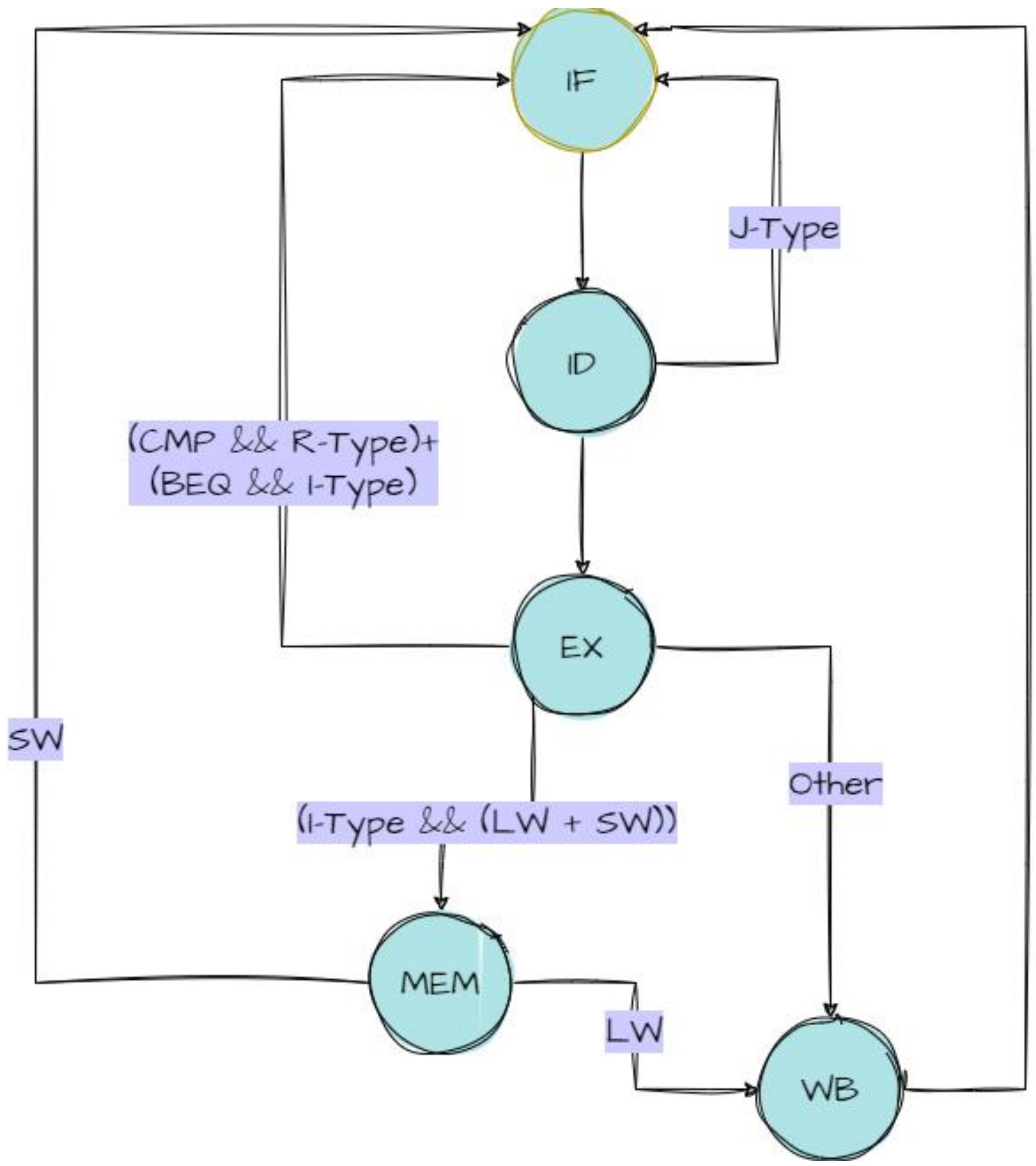


Figure 5: Finite State Machine

Data Path Design and details and description

Control Signals needed

1- PC_Src: since there is multiple possible values for the next pc value, a signal is needed to choose the value. The possible options are PC+1, BTA, JA, RA.
   - PC+1: because that the memory cell in our implementation is 32-bits wide, so each instruction is stored in 1 memory cell, so after fetch it, the pc value must be incremented by 1.
   - BTA: it will be used as a next pc value for BEQ instruction if the branch is taken.
   - JA: it will be the next pc value in J-Type instructions.
   - RA: will be used as next pc value if stop bit is 1.

2- Reg_Src: Since that CMP instruction is using the Rd as second register operand, then a signal is needed to choose between Rd and Rs2 as a second register to be read from the register file.

3- Reg_Write: to prevent and manage the write on register operation, since there are instructions that prevented from write to the register. In opposite, there is other instructions that will write on the register file.

4- ALU_Src: The ALU has two inputs, first input is BusA, and the second input has different possible values, so ALU_Src signal will be used to choose the second ALU input between these options: extended-immediate, BusB and Extended-Shift Amount.

5- ALU_Op: it will be used to manage the ALU and choose the specific operation to be performed.

6- Mem_Read: only LW instruction can read from memory, so this signal will manage the read operation and to prevents the other instructions from read from memory.

7- Mem_Write: only SW instruction can Write to memory, so this signal will manage the write operation and to prevents the other instructions from Write to memory.

8- WB_Src: to choose the suitable data to be written into the register file, two possible values are ALU_Result and Data_out_of_Memory.

9- ExtOp: to decide if the extension operation will be zero or sign extension.

Data Path Design

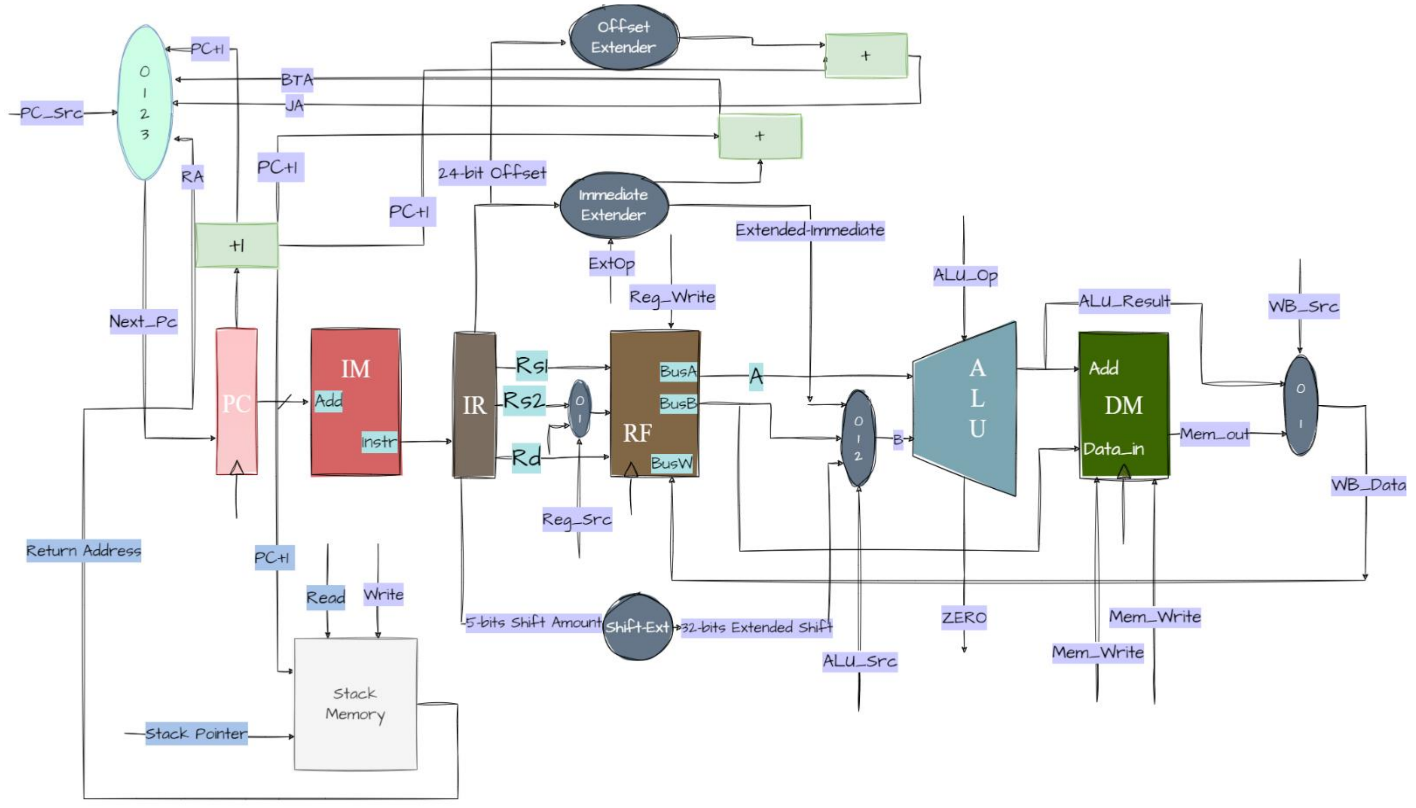


Figure 6: Data Path Design

Control Unit Block

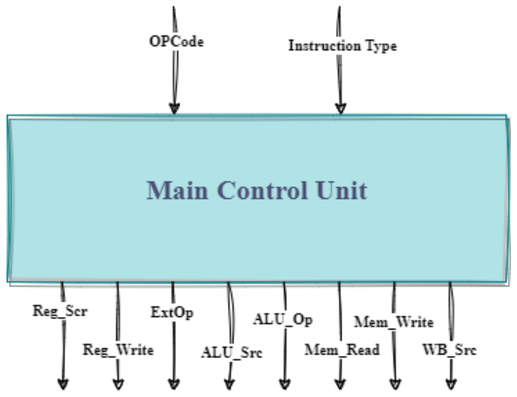


Figure 7: Control Unit

Components

After analyzing the instruction set, we found that the following components are needed.

Instruction Memory & 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.

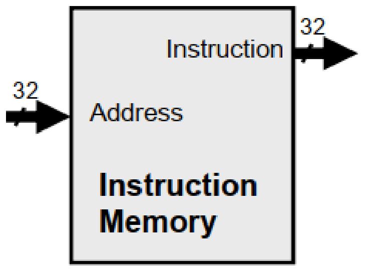


Figure 8: Instruction Memory Module

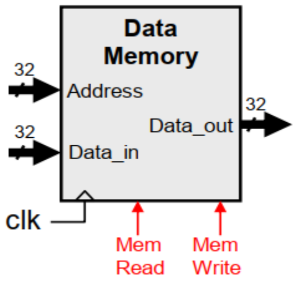
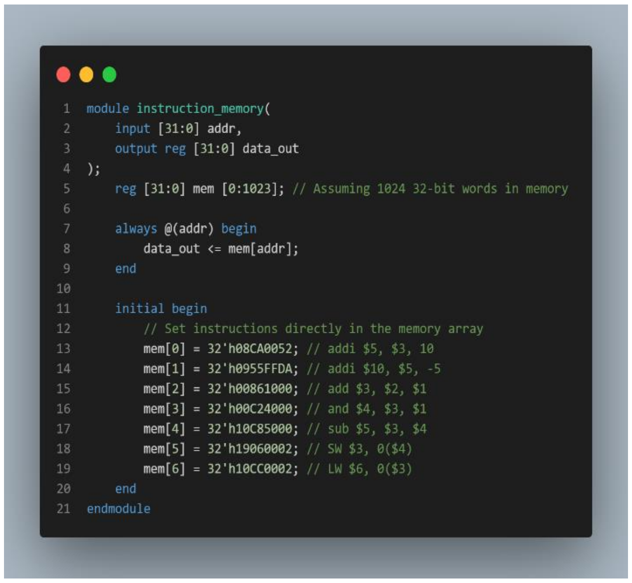


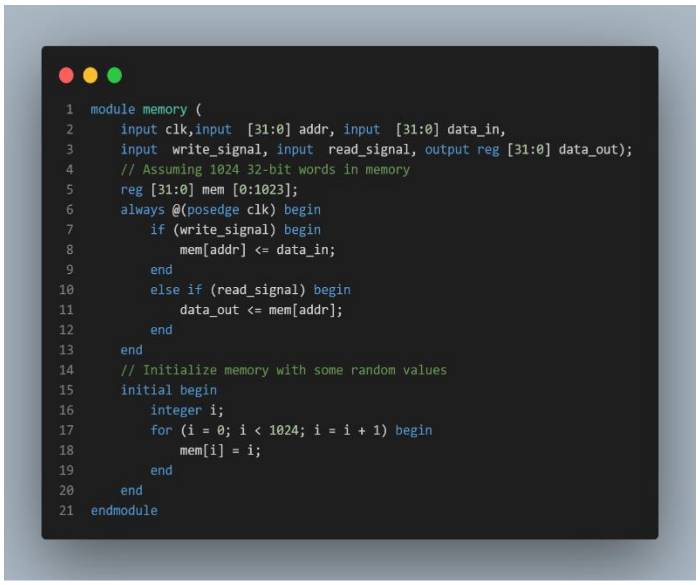
Figure 9: Data Memory Module

Code of instruction memory and data memory



```
module instruction_memory(
    input [31:0] addr,
    output reg [31:0] data_out
);
    reg [31:0] mem [0:1023]; // Assuming 1024 32-bit words in memory

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

    initial begin
        // Set instructions directly in the memory array
        mem[0] = 32'h08CA0052; // addi $5, $3, 10
        mem[1] = 32'h0955FFDA; // addi $10, $5, -5
        mem[2] = 32'h00861000; // add $3, $2, $1
        mem[3] = 32'h00C24000; // and $4, $3, $1
        mem[4] = 32'h10C85000; // sub $5, $3, $4
        mem[5] = 32'h19060002; // SW $3, 0($4)
        mem[6] = 32'h10CC0002; // LW $6, 0($3)
    end
endmodule
```

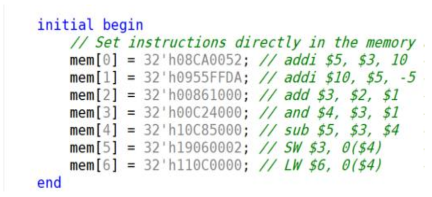
Figure 11: Instruction Memory Code



```
module memory (
    input clk,input  [31:0] addr, input  [31:0] data_in,
    input  write_signal, input  read_signal, output reg [31:0] data_out);
    // Assuming 1024 32-bit words in memory
    reg [31:0] mem [0:1023];
    always @(posedge clk) 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
```

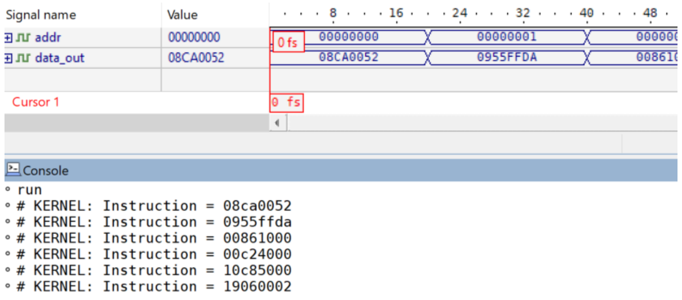
Figure 10: Data Memory Code

Test for instruction memory module



```
initial begin
    // Set instructions directly in the memory
    mem[0] = 32'h08CA0052; // addi $5, $3, 10
    mem[1] = 32'h0955FFDA; // addi $10, $5, -5
    mem[2] = 32'h00861000; // add $3, $2, $1
    mem[3] = 32'h00C24000; // and $4, $3, $1
    mem[4] = 32'h10C85000; // sub $5, $3, $4
    mem[5] = 32'h19060002; // SW $3, 0($4)
    mem[6] = 32'h110C0000; // LW $6, 0($4)
end
```

Figure 12: Sample Instructions

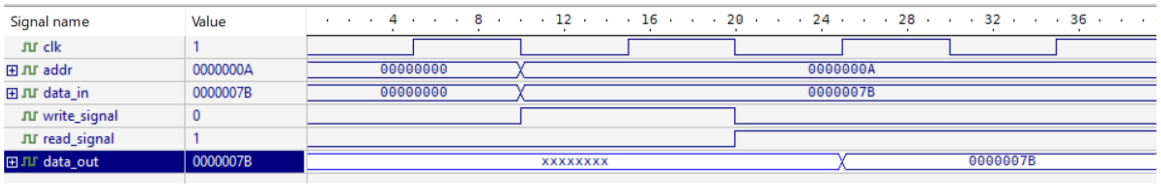


Signal name	Value
addr	00000000
data_out	08CA0052

```
◦ run
◦ # KERNEL: Instruction = 08ca0052
◦ # KERNEL: Instruction = 0955ffda
◦ # KERNEL: Instruction = 00861000
◦ # KERNEL: Instruction = 00c24000
◦ # KERNEL: Instruction = 10c85000
◦ # KERNEL: Instruction = 19060002
```

Figure 13: Instruction Memory Test

Test for data memory module



Signal name	Value
clk	1
addr	0000000A
data_in	0000007B
write_signal	0
read_signal	1
data_out	0000007B

Figure 14: Data Memory Test

Register File

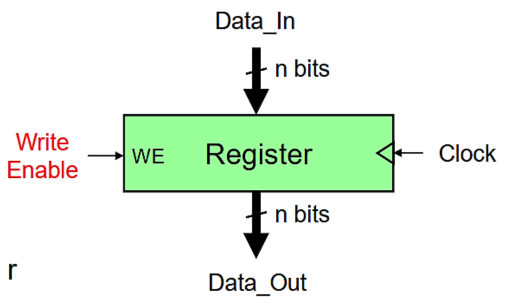


Figure 16: Register Element

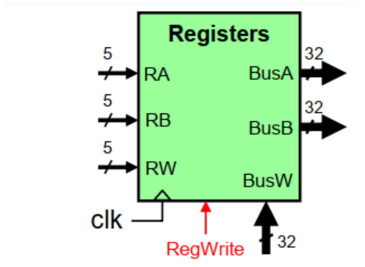
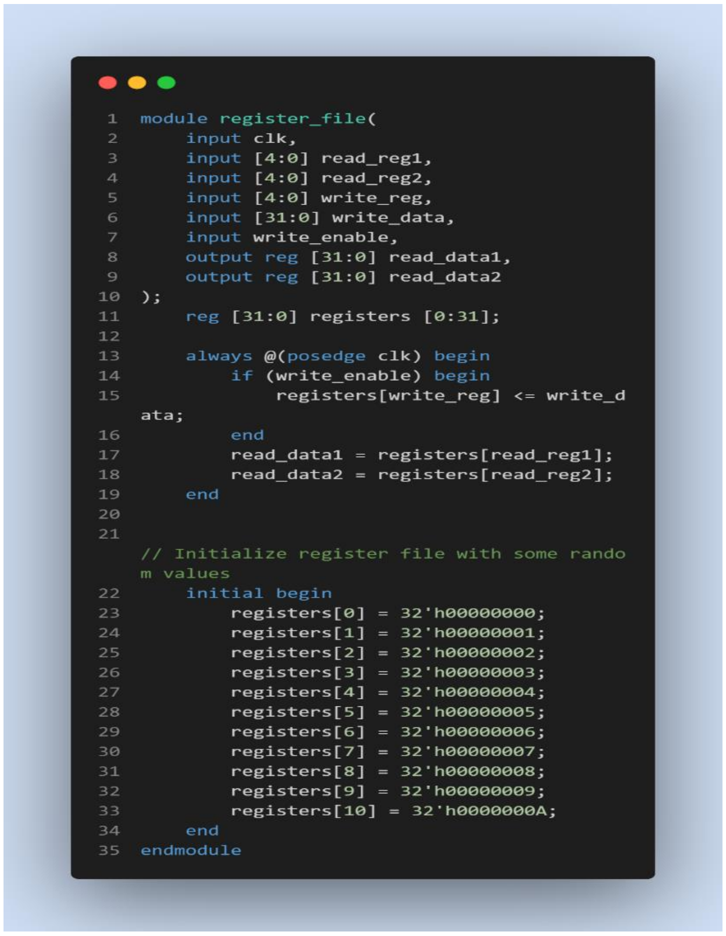


Figure 15: Register File Module

The register file is an essential component in the CPU, providing high-speed storage and quick access to registers for various operations. Its efficient design and close proximity to the execution units contribute to the overall performance and functionality of the computer system.

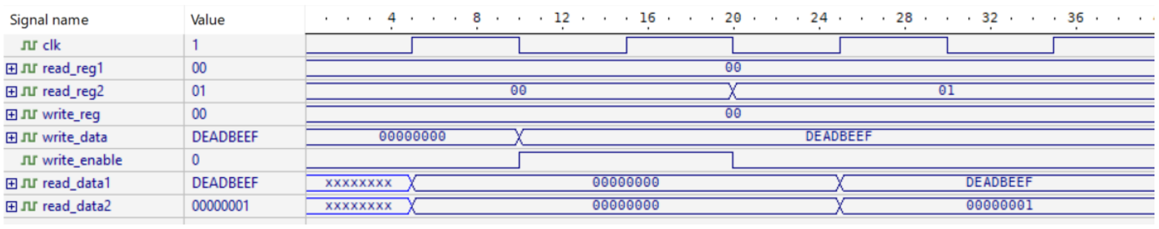
Register File Code



```verilog
module register_file(
    input clk,
    input [4:0] read_reg1,
    input [4:0] read_reg2,
    input [4:0] write_reg,
    input [31:0] write_data,
    input write_enable,
    output reg [31:0] read_data1,
    output reg [31:0] read_data2
);
    reg [31:0] registers [0:31];

    always @(posedge clk) begin
        if (write_enable) begin
            registers[write_reg] <= write_data;
        end
        read_data1 = registers[read_reg1];
        read_data2 = registers[read_reg2];
    end


// Initialize register file with some random values
    initial begin
        registers[0] = 32'h00000000;
        registers[1] = 32'h00000001;
        registers[2] = 32'h00000002;
        registers[3] = 32'h00000003;
        registers[4] = 32'h00000004;
        registers[5] = 32'h00000005;
        registers[6] = 32'h00000006;
        registers[7] = 32'h00000007;
        registers[8] = 32'h00000008;
        registers[9] = 32'h00000009;
        registers[10] = 32'h0000000A;
    end
endmodule
```

Figure 17: Register File Code

Test Register File



Signal name	Value
clk	1
read_reg1	00
read_reg2	01
write_reg	00
write_data	DEADBEEF
write_enable	0
read_data1	DEADBEEF
read_data2	00000001

Figure 18: Test Register File

Arithmetic Logical Unit

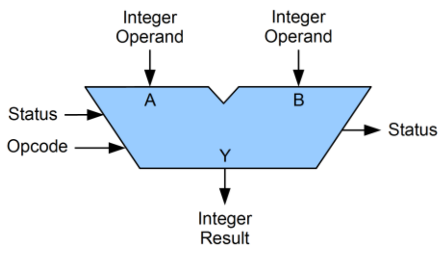
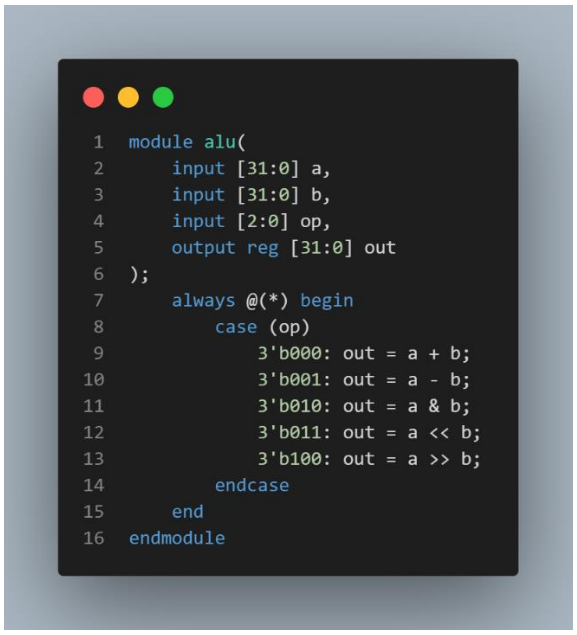


Figure 19: ALU block

The ALU is a digital circuit within the CPU that executes arithmetic and logical operations on binary data. It takes two input operands, operates on them according to the specified operation, and produces a result. The ALU can perform a wide range of operations, including addition, subtraction, multiplication, division, bitwise logical operations (AND, OR, XOR), and comparisons (such as greater than, less than, equal to).

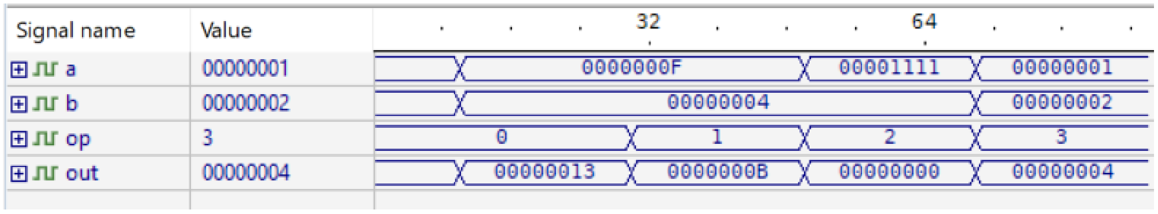
ALU code



```
module alu(
    input [31:0] a,
    input [31:0] b,
    input [2:0] op,
    output reg [31:0] out
);
    always @(*) begin
        case (op)
            3'b000: out = a + b;
            3'b001: out = a - b;
            3'b010: out = a & b;
            3'b011: out = a << b;
            3'b100: out = a >> b;
        endcase
    end
endmodule
```

Figure 20: ALU Code

ALU Test



Signal name	Value					
a	00000001		0000000F	00001111	00000001
b	00000002		00000004		00000002
op	3	0	1	2	3
out	00000004	00000013	0000000B	00000000	00000004

Figure 21: ALU test

Extenders

We used 3 extenders in our data path in order to support all instruction types, for immediate and offset and shift.

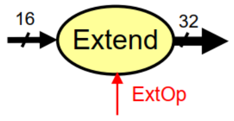
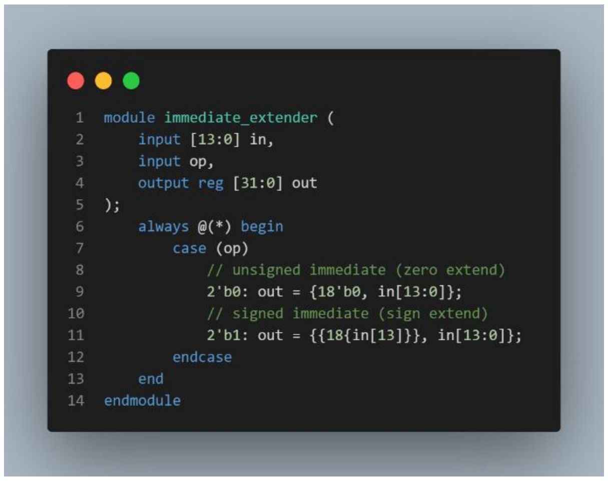


Figure 22: Extender

Signed Extender

A signed extender is a component or circuit that extends the bit width of a binary number while preserving its interpretation as either a signed or unsigned value.
When extending a binary number, the most significant bit (MSB) is usually replicated to fill the additional bits. This replication is called sign extension or zero extension, depending on whether the original number is interpreted as signed or unsigned.
In our data path, we used the extender to extended the immediate 14-bit to 32-bit, It can be either signed or zero extension, depends on the control signal ExtOp.

Signed Extender 14 to 32-bits Code



```
module immediate_extender (
    input [13:0] in,
    input op,
    output reg [31:0] out
);
    always @(*) begin
        case (op)
            // unsigned immediate (zero extend)
            2'b0: out = {18'b0, in[13:0]};
            // signed immediate (sign extend)
            2'b1: out = {{18{in[13]}}, in[13:0]};
        endcase
    end
endmodule
```

Figure 23: Signed Extender Code

Signed Extender Test

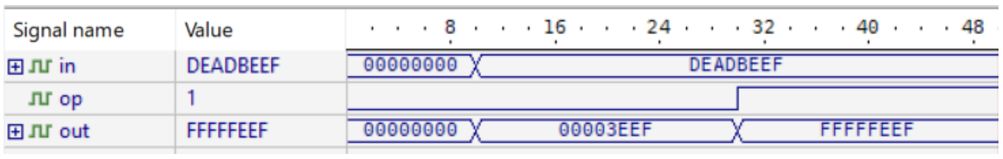
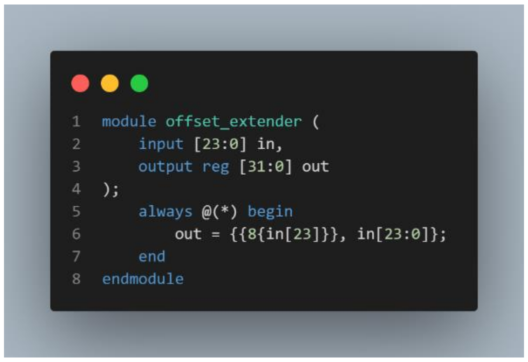


Figure 24: Signed Extender Test

Offset Extender

used to extend the offset field from 24-bits to 32-bits so that it can be used to calculate the jump address. The input if the offset-24bits field and the output is a 32-bits offset register.

Offset Extender Code



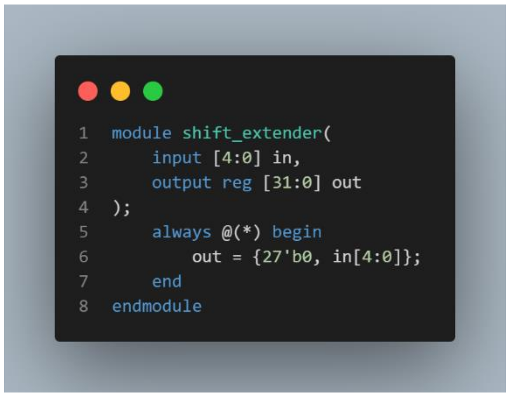
```verilog
module offset_extender (
    input [23:0] in,
    output reg [31:0] out
);
    always @(*) begin
        out = {{8{in[23]}}, in[23:0]};
    end
endmodule
```

Figure 25: Offset Extender Code

Shift Extender

We have user a 5 to 32-bits extender to extend the shift amount field so that it can be used as a second ALU operand in shift operations. The input is the Shift Amount 5-bits field, and the output is a shift amount zero extended register.

Shift Extender Code



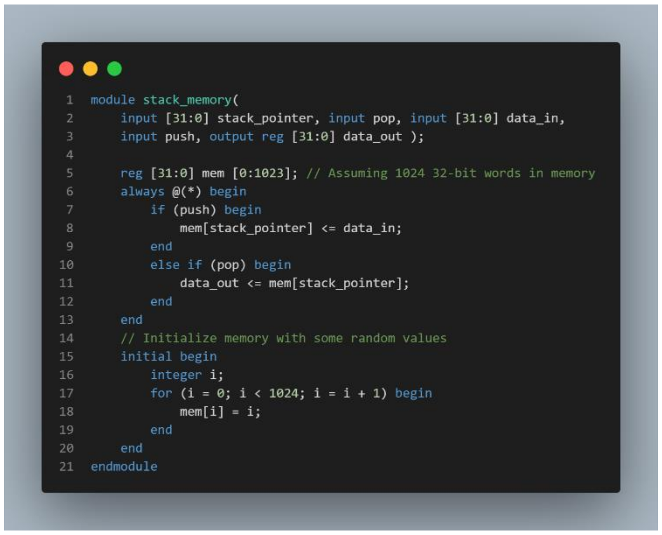
```verilog
module shift_extender(
    input [4:0] in,
    output reg [31:0] out
);
    always @(*) begin
        out = {27'b0, in[4:0]};
    end
endmodule
```

Figure 26: Shift Extender Code

Stack Memory

Stack memory, often referred to simply as the stack, is a region of a computer's memory used for dynamic storage allocation and management. It is a fundamental data structure in most programming languages and is crucial for managing function calls, local variables, and supporting nested function execution. We have used the stack memory to store and retrieve the return address and variables with functions, since its efficient for managing function calls and local variables due to its simple and fast LIFO structure. It provides a convenient way to organize and access data within the scope of functions and is an integral part of program execution in most programming languages.

Stack Memory Code



```
module stack_memory(
    input [31:0] stack_pointer, input pop, input [31:0] data_in,
    input push, output reg [31:0] data_out );

    reg [31:0] mem [0:1023]; // Assuming 1024 32-bit words in memory
    always @(*) begin
        if (push) begin
            mem[stack_pointer] <= data_in;
        end
        else if (pop) begin
            data_out <= mem[stack_pointer];
        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 27: Stack Memory Code

Stack Memory Test

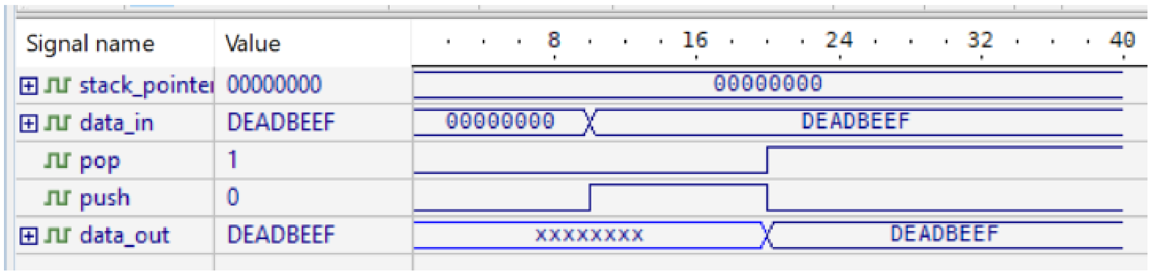


Figure 28: Stack Memory Test

Multiplexers

It is a combinational circuit which have many data inputs and single output depending on control or select inputs. For N input lines, log n (base2) selection lines, or we can say that for 2^n input lines, n selection lines are required. Multiplexers are also known as **“Data n selector, parallel to serial convertor, many to one circuit, universal logic circuit”**. Multiplexers are mainly used to increase amount of the data that can be sent over the network within certain amount of time and bandwidth.

2x1 Mux

Two 2x1 mux were used in our design of the multicycle processor as shown below:

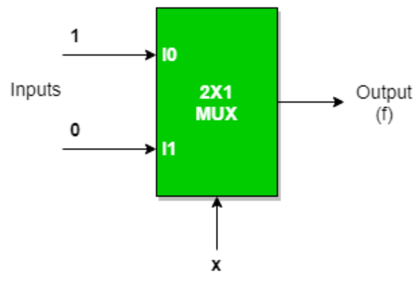


Figure 29: 2x1 Mux Block

1- Based on the value of the write back source signal, which will be used as the selection line, determine which data will be written into the register file—either the memory output or ALU result.
2- Based on the value of the Register Source signal, which will be used as a selection line, determine which register will be read as a second operand from the register file—either Rs2 or Rd.

4x1 Mux

Two 4x1 mux were used in our design of the multicycle processor as shown below:

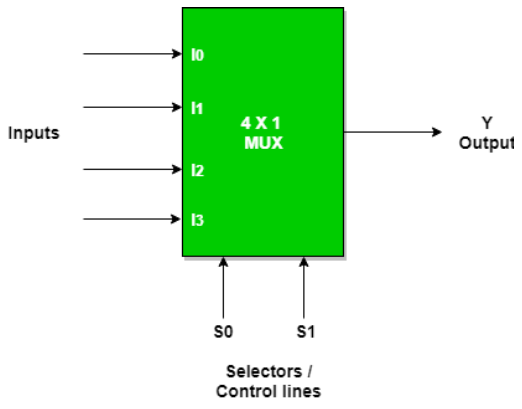


Figure 30: 4x1 Multiplexer Block

1- A 4x1 mux has been used to determine the next PC value based on the value of the PC source signal, which has been used as a selection line. The inputs of the mux are [PC+1, BTA, JA, RA], and the next PC will be calculated as a result of the mux operation.
2- The other 4x1 Mux has been used to determine the ALU second operand, based on the value the ALU source signal, which has been used as a selection line of the mux. The inputs of the mux are Second Operand Register Rb, Extended immediate, Extended Shift Amount.

8x1 Mux

One 8x1 Mux has been used in our design of the multi cycle processor, it used to determine the ALU operation to be performed in the ALU, it has 5 inputs are (ADD, AND, SUB, SLL, SLR). the ALU_Op signal has been used as a selection line of 3-bits to determine which operation will be chosen.

Instruction Register

The instruction being executed at the moment is stored in the instruction register (IR), a part of the control unit of a multi-cycle CPU. The decoding of the instruction and selection of the proper control signals for the remainder of the CPU are tasks performed by the IR.

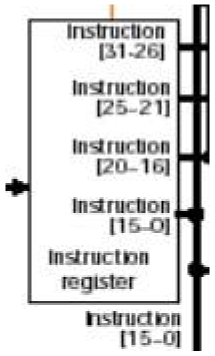
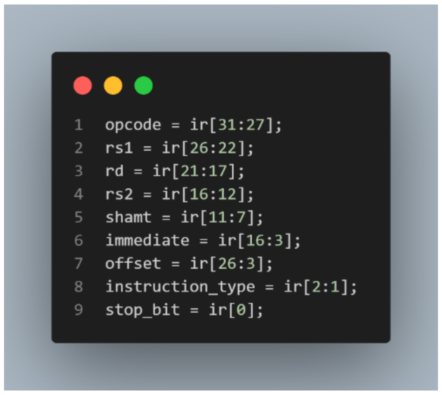


Figure 31: Instruction Register Block

Instruction Register Field



```
opcode = ir[31:27];
rs1 = ir[26:22];
rd = ir[21:17];
rs2 = ir[16:12];
shamt = ir[11:7];
immediate = ir[16:3];
offset = ir[26:3];
instruction_type = ir[2:1];
stop_bit = ir[0];
```

Figure 32: Instruction Register Fields

PC Register

The PC (Program Counter) register, also known as the instruction pointer, is a special-purpose register in a CPU (Central Processing Unit). It is used to store the memory address of the next instruction to be fetched and executed by the processor. When the instruction is being executed, the PC register is used to determine the address of the subsequent instruction to be fetched. It allows the processor to sequentially fetch and execute instructions in the correct order, advancing the program execution.

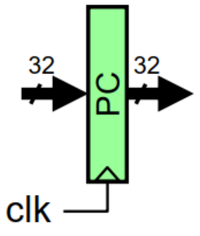


Figure 33: PC Register Block

Control Units – Main, PC, and Stack Memory Control

Main Control Unit

Main Control Input

- 5-bit opcode field and 2-bit instruction type field

Main Control Output

- Main control signals

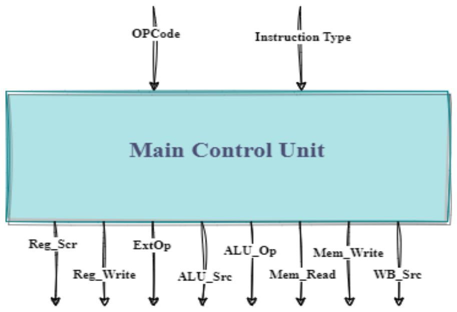


Figure 34: Main Control Unit Block

Main Control Signals

Signal	Effect
Reg_Src	When = 0, Second operand is Rs2 (to be read from RF) When = 1, Second Operand is Rd (to be read from RF)
Reg_Write	When =0, No register is written When = 1, Destination Register Rd is written with the data on BusW

ExtOp	When = 0, 14-bit immediate is Zero-Extended
	When = 1, 14-bit immediate is Sign-Extended

ALU_Src	When = 00, Second ALU operand is the value of the extended 14-bit immediate When = 01, Second ALU operand is the value of register (Rs2/Rd) that appears on BusB When = 10, Second ALU operand is the value of the extended 5-bit shift amount

Mem_Read	When = 0, Data Memory is NOT read
	When = 1, Data Memory is read: Mem_out 🡐 Memory [Address]

Mem_Write	When = 0, Data Memory is NOT written
	When = 1, Data Memory is written: Mem [Address] 🡐 Data_in

WB_Src	When = 0, ALU result will be written on the register: BusW = ALU result When = 1, Data from memory will be written on the register: BusW= Mem_out
ALU_Op	When = 000 => ADD operation will be performed in the ALU When = 001 => SUB operation will be performed in the ALU When = 010 => AND operation will be performed in the ALU When = 011 => Shift Logical Left operation will be performed in the ALU When = 100 => Shift Logical Right operation will be performed in the ALU

Table 2: Main Control Truth Table

Instr_type	Opcode	Reg_Src	Reg_Write	ExtOp	ALU_Src	Mem_Read	Mem_Write	WB_Src	ALU_Op
R	AND	0 = Rs2	1	X	01 = (BusB)	0	0	0 = ALU	010 = AND
R	ADD	0 = Rs2	1	X	01 = (BusB)	0	0	0 = ALU	000 = ADD
R	SUB	0 = Rs2	1	X	01 = (BusB)	0	0	0 = ALU	001 = SUB
R	CMP	0 = Rs2	0	X	01 = (BusB)	0	0	X	001 = SUB
I	ANDI	X	1	0 = Zero	00 = immediate	0	0	0 = ALU	010 = AND
I	ADDI	X	1	1 = Sign	00 = immediate	0	0	0 = ALU	000 = ADD
I	LW	X	1	1 = Sign	00 = immediate	1	0	1 = Mem	000 = ADD
I	SW	X	0	1 = Sign	00 = immediate	0	1	X	000 = ADD
I	BEQ	1 = Rd	0	1 = Sign	01 = BusB	0	0	X	001 = SUB
J	J	X	0	X	X	0	0	X	X
J	JAL	X	0	X	X	0	0	X	X
S	SLL	X	1	X	10 = Shift Amount	0	0	0 = ALU	011 = SLL
S	SLR	X	1	X	10 = Shift Amount	0	0	0 = ALU	100 = SLR
S	SLLV	0 = Rs2	1	X	01 = (BusB)	0	0	0 = ALU	011 =SLL
S	SLRV	0 = Rs2	1	X	01 = (BusB)	0	0	0 = ALU	100 = SLR

Table 3: Main Control Truth Table

Logic Equation for Main Control Signals

- Reg_Src = BEQ && I-type
- Reg_Write = (R-Type && ~CMP) || (I-type && ~SW) || (I-type && ~BEQ) || (S-Type)
- ExtOp = ~ (I-Type && ANDI)
- WB_Src = SW && I-Type
- Mem_Read = I-Type && LW
- Mem_Write = I-Type && SW
- ALU_Src = {S1, S0}

  Where:
  - S1 = (S-Type && (SLL || SLR))
  - S0 = (R-Type || (I-Type && BEQ) || (S-Type && (SLLV || SLRV)))
- ALU_Op = {S2, S1, S0}

  Where:
  - S2 = (S-Type && (SLR || SLRV))
  - S1 = (R-Type && AND) || (I-Type && ANDI) || (S-Type && (SLL || SLLV))
  - S0 = (R-Type && (SUB || CMP)) || (I-Type && BEQ) || (S-Type && (SLL || SLLV))

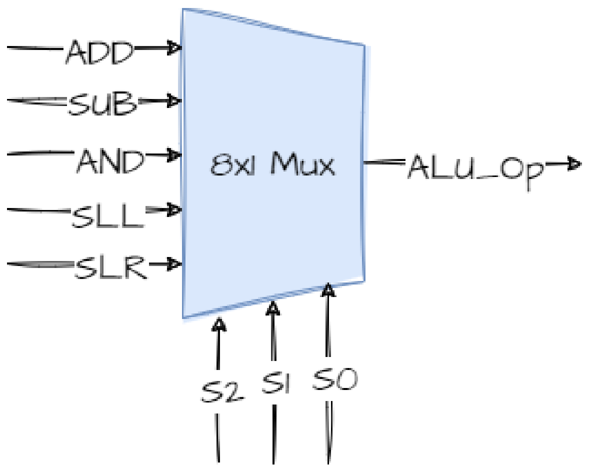


Figure 35: ALU Operation Signal

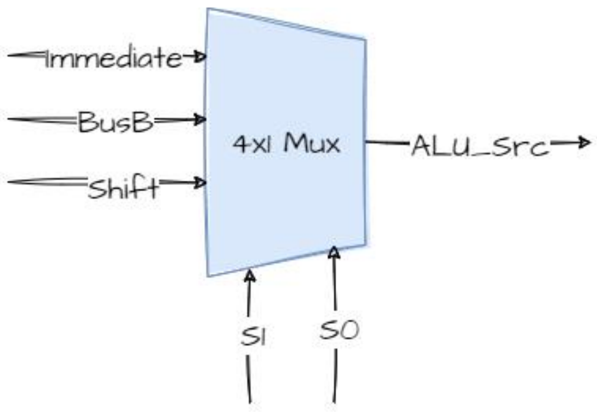
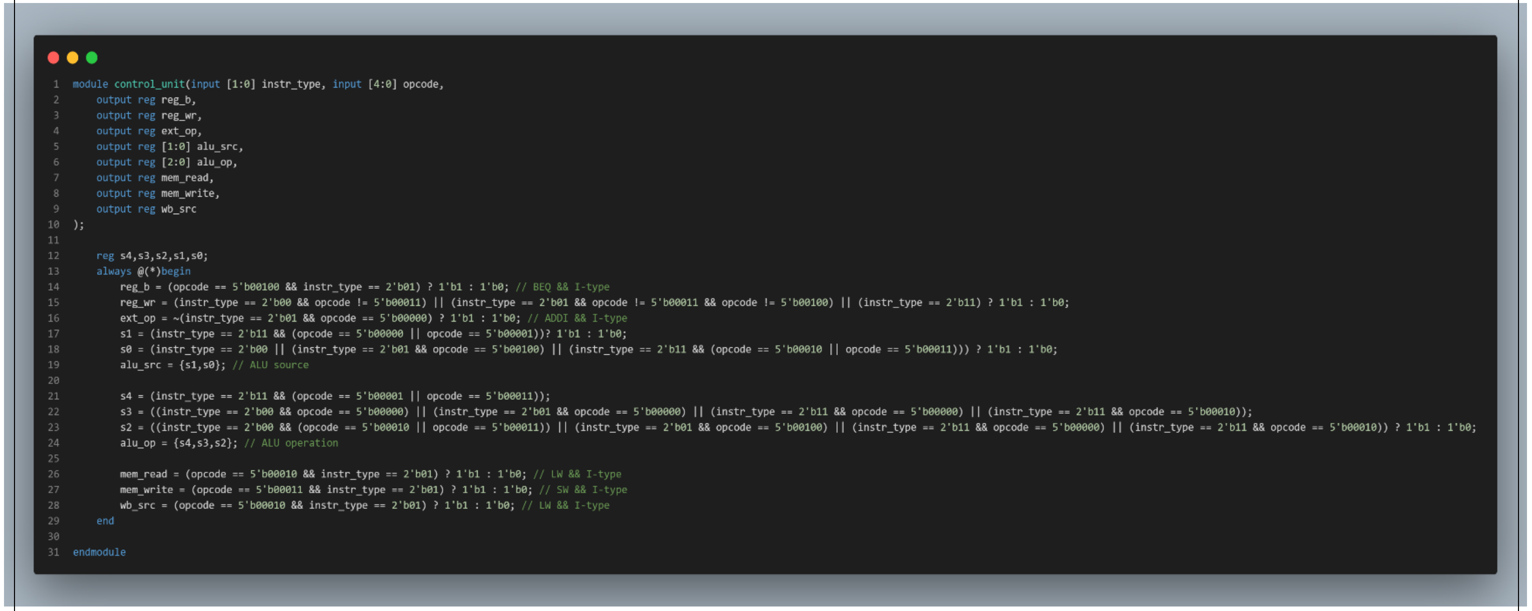


Figure 36: ALU_Src Signal

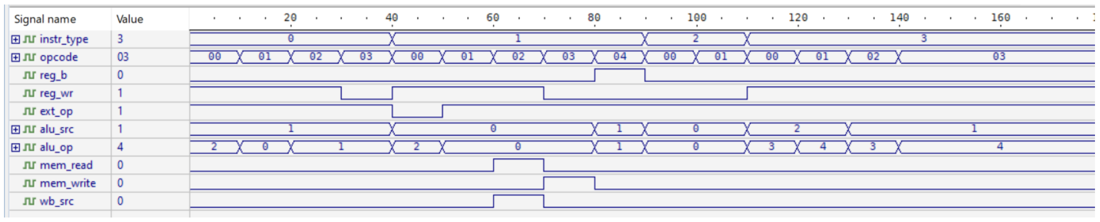
Main Control Unit Code



```verilog
module control_unit(input [1:0] instr_type, input [4:0] opcode,
    output reg reg_b,
    output reg reg_wr,
    output reg ext_op,
    output reg [1:0] alu_src,
    output reg [2:0] alu_op,
    output reg mem_read,
    output reg mem_write,
    output reg wb_src
);

    reg s4,s3,s2,s1,s0;
    always @(*)begin
        reg_b = (opcode == 5'b00100 && instr_type == 2'b01) ? 1'b1 : 1'b0; // BEQ && I-type
        reg_wr = (instr_type == 2'b00 && opcode != 5'b00011) || (instr_type == 2'b01 && opcode != 5'b00011 && opcode != 5'b00100) || (instr_type == 2'b11) ? 1'b1 : 1'b0;
        ext_op = ~(instr_type == 2'b01 && opcode == 5'b00000) ? 1'b1 : 1'b0; // ADDI && I-type
        s1 = (instr_type == 2'b11 && (opcode == 5'b00000 || opcode == 5'b00001))? 1'b1 : 1'b0;
        s0 = (instr_type == 2'b00 || (instr_type == 2'b01 && opcode == 5'b00100) || (instr_type == 2'b11 && (opcode == 5'b00010 || opcode == 5'b00011))) ? 1'b1 : 1'b0;
        alu_src = {s1,s0}; // ALU source

        s4 = (instr_type == 2'b11 && (opcode == 5'b00001 || opcode == 5'b00011));
        s3 = ((instr_type == 2'b00 && opcode == 5'b00000) || (instr_type == 2'b01 && opcode == 5'b00000) || (instr_type == 2'b11 && opcode == 5'b00000) || (instr_type == 2'b11 && opcode == 5'b00010));
        s2 = ((instr_type == 2'b00 && (opcode == 5'b00010 || opcode == 5'b00011)) || (instr_type == 2'b01 && opcode == 5'b00100) || (instr_type == 2'b11 && opcode == 5'b00000) || (instr_type == 2'b11 && opcode == 5'b00010)) ? 1'b1 : 1'b0;
        alu_op = {s4,s3,s2}; // ALU operation

        mem_read = (opcode == 5'b00010 && instr_type == 2'b01) ? 1'b1 : 1'b0; // LW && I-type
        mem_write = (opcode == 5'b00011 && instr_type == 2'b01) ? 1'b1 : 1'b0; // SW && I-type
        wb_src = (opcode == 5'b00010 && instr_type == 2'b01) ? 1'b1 : 1'b0; // LW && I-type
    end

endmodule
```

Figure 37: Main Control Unit Code

Main Control Unit Test



Signal name	Value
instr_type	3
opcode	03
reg_b	0
reg_wr	1
ext_op	1
alu_src	1
alu_op	4
mem_read	0
mem_write	0
wb_src	0

Figure 38: Main Control Unit Test

PC Control Unit

PC Control Input

- 5-bit opcode field, 2-bit instruction type field, 1-bit Zero Signal and Stop bit.

PC Control Output

- PC Source signal

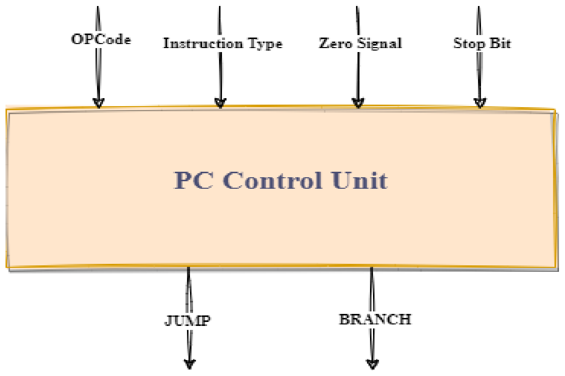


Figure 39: PC Control Block

PC Control Truth Table

OPCode	Type	Zero Signal	Stop bit	JUMP	BRANCH
X	X	X	1	1	1
X	R	X	0	0	0
X	S	X	0	0	0
X	J	X	0	1	0
BEQ	I	0	0	0	0
BEQ	I	1	0	0	1

Table 4: PC Control Truth Table

Logic Equations for PC Source Signal

- JUMP = J-Type || Stop-bit
- BRANCH = (BEQ && Zero_Signal && I-Type) || Stop-bit

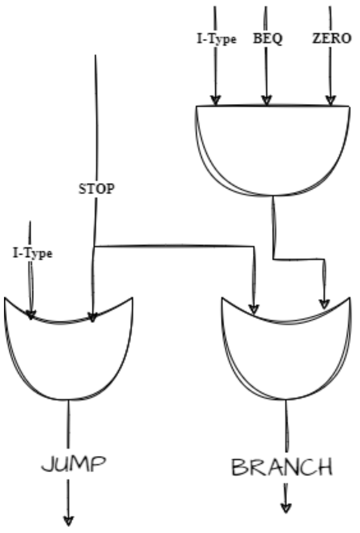


Figure 40: PC Source Block Diagram

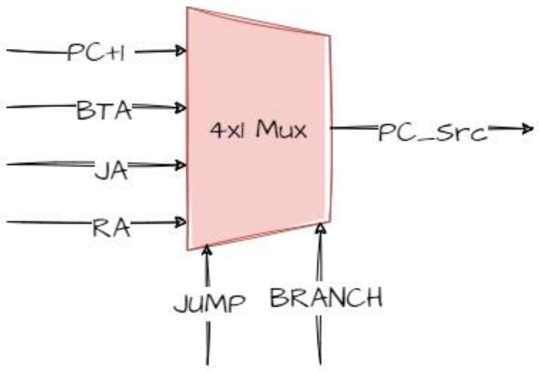
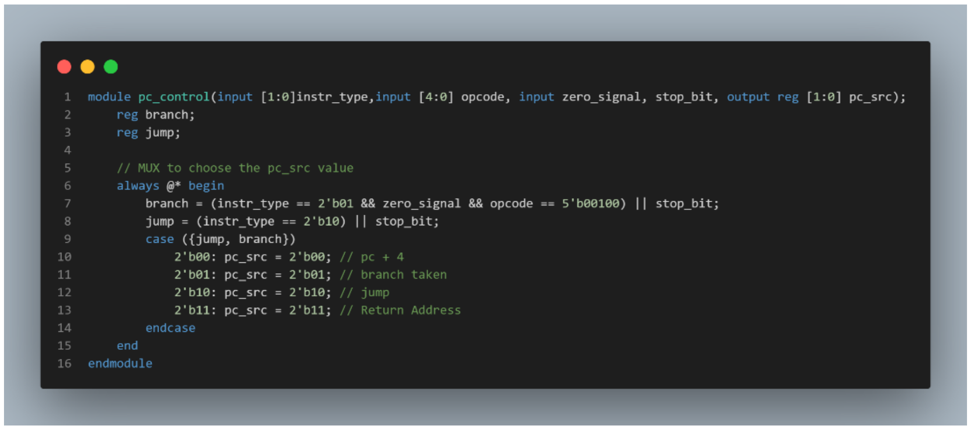


Figure 41: PC Source Signal Block

PC Control Code



```verilog
module pc_control(input [1:0]instr_type,input [4:0] opcode, input zero_signal, stop_bit, output reg [1:0] pc_src);
    reg branch;
    reg jump;

    // MUX to choose the pc_src value
    always @* begin
        branch = (instr_type == 2'b01 && zero_signal && opcode == 5'b00100) || stop_bit;
        jump = (instr_type == 2'b10) || stop_bit;
        case ({jump, branch})
            2'b00: pc_src = 2'b00; // pc + 4
            2'b01: pc_src = 2'b01; // branch taken
            2'b10: pc_src = 2'b10; // jump
            2'b11: pc_src = 2'b11; // Return Address
        endcase
    end
endmodule
```

Figure 42: PC Control Code

PC Control Test

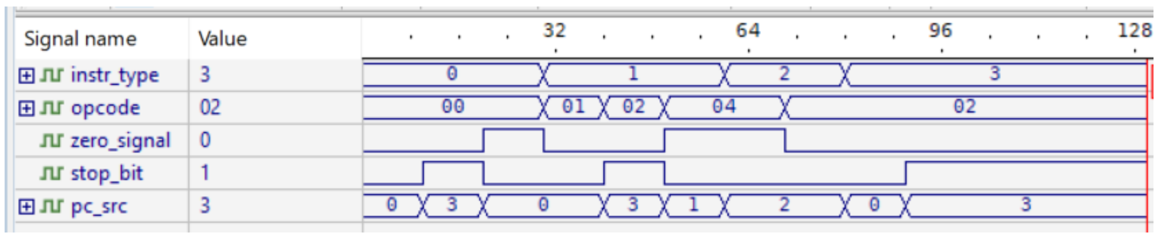


Figure 43: PC Control Test

Stack Memory Control Unit

Stack Control Input

- 5-bit opcode field, 2-bit instruction type field, 1-bit Zero Signal and 32-bits SP.

Stack Control Output

- Push, Pop and new value of SP.

Stack Control Code

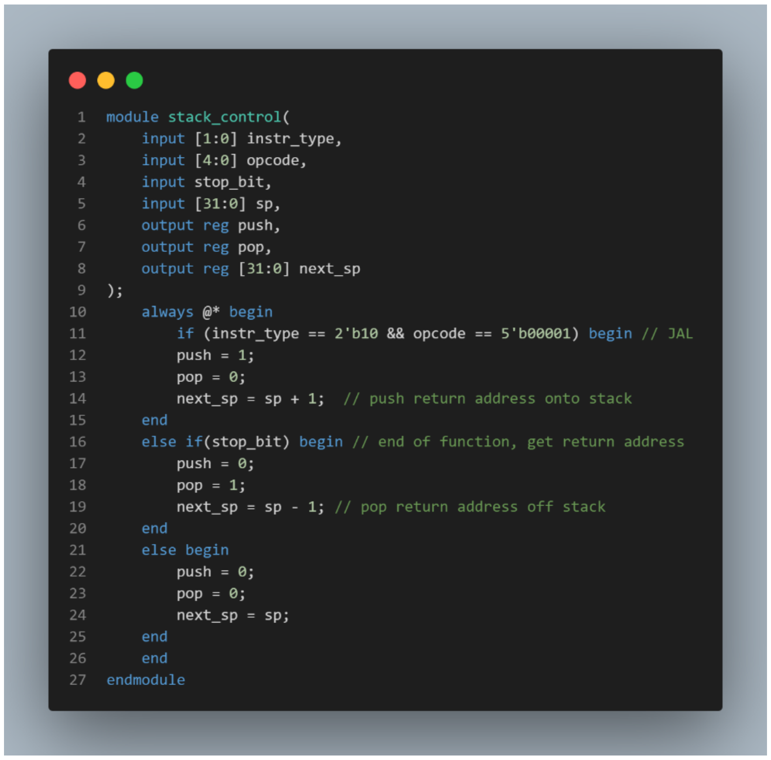

```verilog
module stack_control(
    input [1:0] instr_type,
    input [4:0] opcode,
    input stop_bit,
    input [31:0] sp,
    output reg push,
    output reg pop,
    output reg [31:0] next_sp
);
    always @* begin
        if (instr_type == 2'b10 && opcode == 5'b00001) begin // JAL
            push = 1;
            pop = 0;
            next_sp = sp + 1;  // push return address onto stack
        end
        else if(stop_bit) begin // end of function, get return address
            push = 0;
            pop = 1;
            next_sp = sp - 1; // pop return address off stack
        end
        else begin
            push = 0;
            pop = 0;
            next_sp = sp;
        end
    end
endmodule
```

Figure 44: Stack Control Code

Stack Control Test

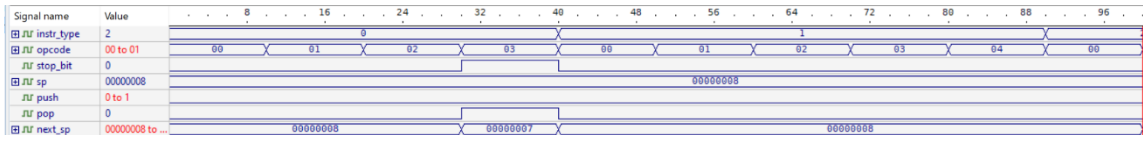


Figure 45: Stack Control Test 1

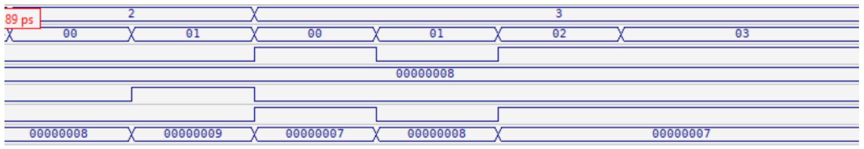
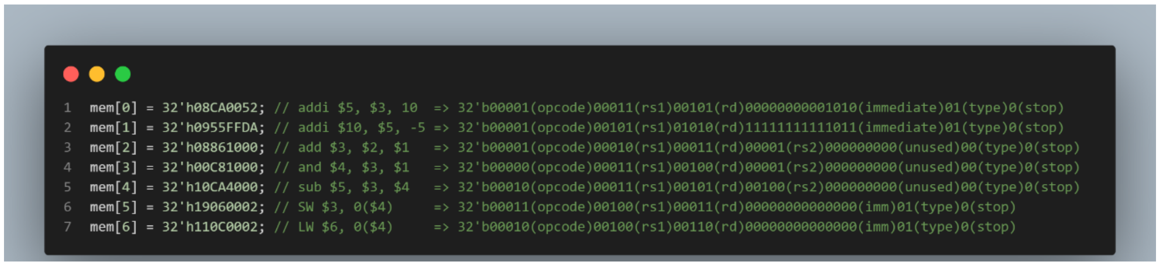


Figure 47: Stack Control Test 2

Testing

Sample instructions for test:



```
mem[0] = 32'h08CA0052; // addi $5, $3, 10  => 32'b00001(opcode)00011(rs1)00101(rd)00000000001010(immediate)01(type)0(stop)
mem[1] = 32'h0955FFDA; // addi $10, $5, -5 => 32'b00001(opcode)00101(rs1)01010(rd)11111111111011(immediate)01(type)0(stop)
mem[2] = 32'h08861000; // add $3, $2, $1   => 32'b00001(opcode)00010(rs1)00011(rd)00001(rs2)000000000(unused)00(type)0(stop)
mem[3] = 32'h00C81000; // and $4, $3, $1   => 32'b00000(opcode)00011(rs1)00100(rd)00001(rs2)000000000(unused)00(type)0(stop)
mem[4] = 32'h10CA4000; // sub $5, $3, $4   => 32'b00010(opcode)00011(rs1)00101(rd)00100(rs2)000000000(unused)00(type)0(stop)
mem[5] = 32'h19060002; // SW $3, 0($4)     => 32'b00011(opcode)00100(rs1)00011(rd)00000000000000(imm)01(type)0(stop)
mem[6] = 32'h110C0002; // LW $6, 0($4)     => 32'b00010(opcode)00100(rs1)00110(rd)00000000000000(imm)01(type)0(stop)
```

Figure 48: Sample Instructions to Test

Note: Data memory and registers are all initialized with value as same as its address:
Mem[i] = i, Register[i] = i

Test Result for first and second instructions:

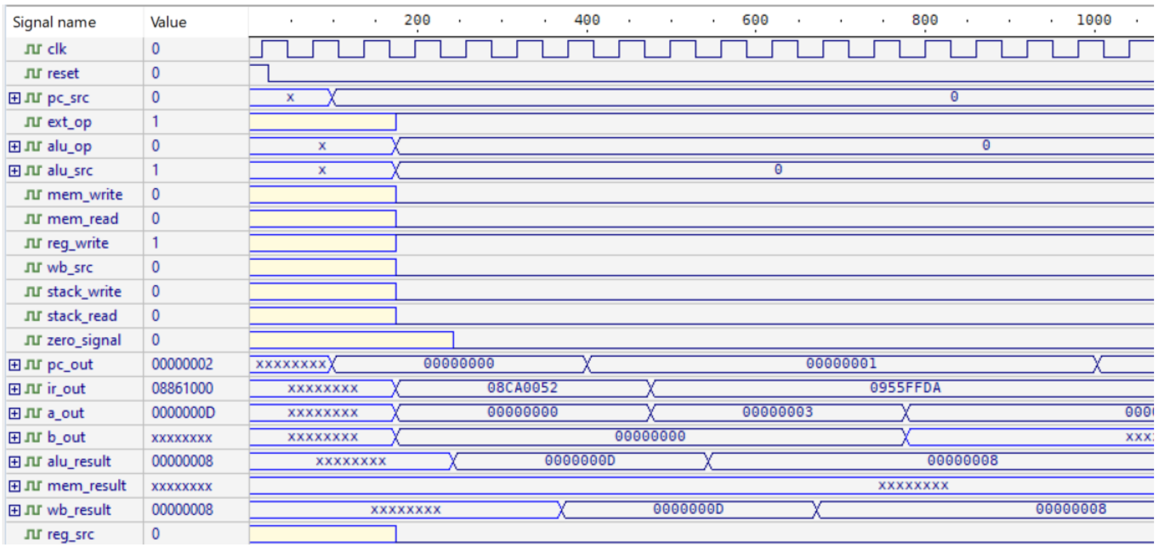


Figure 49: Test Result for first and second instructions

1- After execute first instruction: addi $5, $3, 10:
   - PC_out = Current PC = 0
   - ExtOp = 1 (Add operation => sign extension for immediate field)
   - ALU_Op = 0 => Add operation
   - ALU_Src = 0 => Extended Immediate
   - Mem_Read = Mem_Write = 0 => No memory access
   - Reg_Write = 1 => instruction writes on register file
   - Reg_Src = 0 => has no effect on this instruction
   - WB_Src = 0 => the data to be write on the register is ALU result
   - Stack (Read and Write) = 0 => No stack operations
   - Zero_signal = 0 => it has no effect in the instruction

Result of the instruction:

- As shown in the wave form, ALU Result = 0xD, as follow:
- $S3 = 0x3
- Addi $S5, $S3, 10 => $S5 = 3 + 10 = 13 => 0xD
- WB_Data = 0xD

2- After execute second instruction: addi $10, $5, -5:

a. PC_out = Current PC = 1
b. ExtOp = 1 (Add operation => sign extension for immediate field)
c. ALU_Op = 0 => Add operation
d. ALU_Src = 0 => Extended Immediate
e. Mem_Read = Mem_Write = 0 => No memory access
f. Reg_Write = 1 => instruction writes on register file
g. Reg_Src = 0 => has no effect on this instruction
h. WB_Src = 0 => the data to be write on the register is ALU result
i. Stack (Read and Write) = 0 => No stack operations
j. Zero_signal = 0 => it has no effect in the instruction

Result of the instruction:

- As shown in the wave form, ALU Result = 0xD, as follow:
- $S5 = 0xD => result of previous instruction
- Addi $S10, $S5, 5 => $S10 = 13 - 5 = 8 => 0x8
- WB_Data = 0x8

Test Result for Third and Fourth instructions:

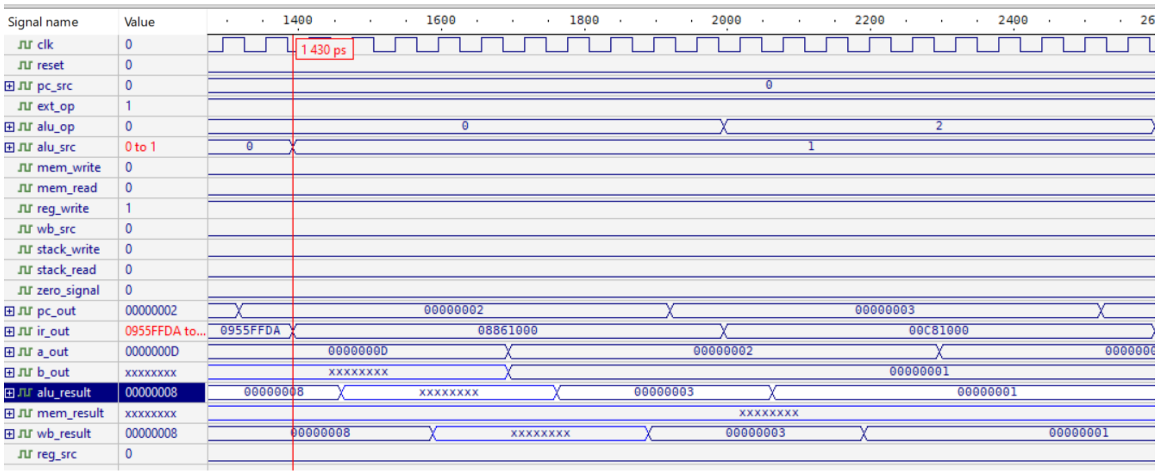


Figure 50: Result of Third and Fourth instructions

1- After execute Third instruction: add \$3, \$2, \$1:
   - PC_out = Current PC = 2
   - ExtOp = 1 has no effect (didn't changed from previous instruction)
   - ALU_Op = 0 => Add operation
   - ALU_Src = 1 => BusB (register \$S1)
   - Mem_Read = Mem_Write = 0 => No memory access
   - Reg_Write = 1 => instruction writes on register file
   - Reg_Src = 0 => Rs2
   - WB_Src = 0 => the data to be write on the register is ALU result
   - Stack (Read and Write) = 0 => No stack operations
   - Zero_signal = 0 => it has no effect in the instruction

Result of the instruction:

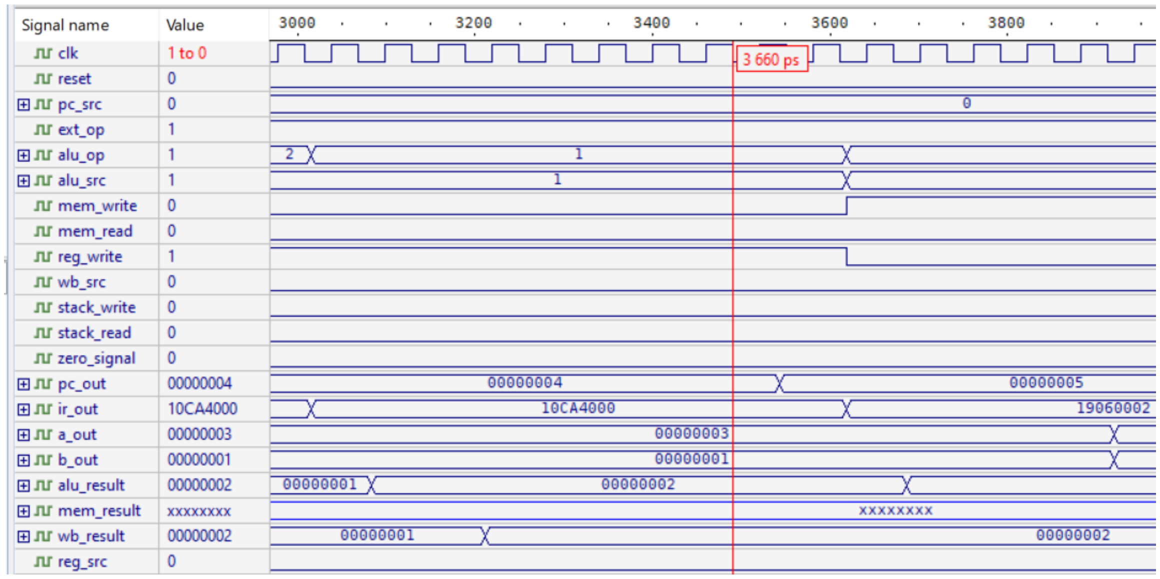
- As shown in the wave form, ALU Result = 0x3, as follow:
- \$S2 = 0x2, \$S1 = 0x1
- Add \$S3, \$S2, \$S1 => \$S3 = 1 + 2 = 3 => 0x3
- WB_Data = 0x3

2- After execute Forth instruction: and $4, $3, $1:
   - PC_out = Current PC = 3
   - ExtOp = 1 has no effect (didn't changed from previous instruction)
   - ALU_Op = 2 => And operation
   - ALU_Src = 1 => BusB (register $S1)
   - Mem_Read = Mem_Write = 0 => No memory access
   - Reg_Write = 1 => instruction writes on register file
   - Reg_Src = 0 => Rs2
   - WB_Src = 0 => the data to be write on the register is ALU result
   - Stack (Read and Write) = 0 => No stack operations
   - Zero_signal = 0 => it has no effect in the instruction

Result of the instruction:

- As shown in the wave form, ALU Result = 0xD, as follow:
- $S3 = 0x3, $S1 = 0x1
- And $S4, $S3, $S1 => $S4 = 0x3 & 0x1 = 1 => 0x1
- WB_Data = 0x1

Result for Fifth instruction:



Signal name	Value
clk	1 to 0
reset	0
pc_src	0
ext_op	1
alu_op	1
alu_src	1
mem_write	0
mem_read	0
reg_write	1
wb_src	0
stack_write	0
stack_read	0
zero_signal	0
pc_out	00000004
ir_out	10CA4000
a_out	00000003
b_out	00000001
alu_result	00000002
mem_result	xxxxxxxx
wb_result	00000002
reg_src	0

Figure 52: Result for Fifth instruction

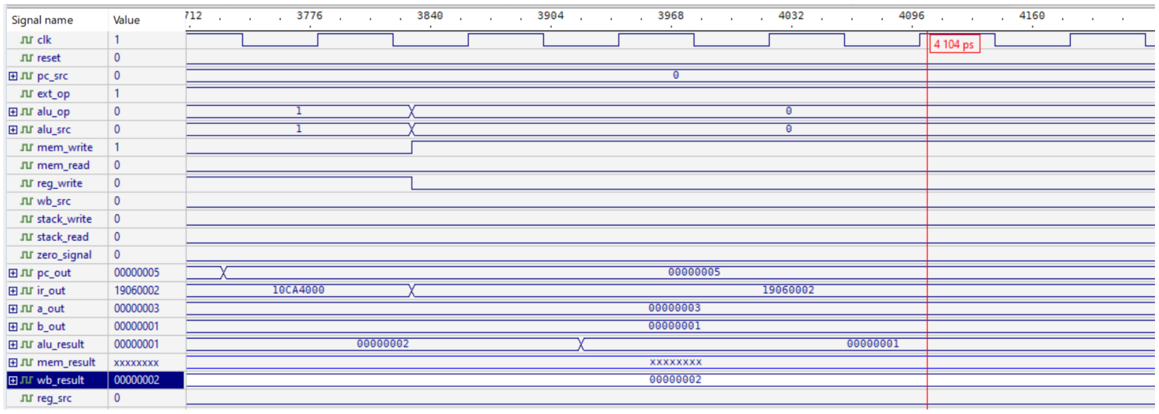
After execute Fifth instruction: sub $5, $3, $4:

- PC_out = Current PC = 0x4
- ExtOp = 1 has no effect (didn't changed from previous instruction)
- ALU_Op = 1 => Sub operation
- ALU_Src = 1 => BusB (register $S1)
- Mem_Read = Mem_Write = 0 => No memory access
- Reg_Write = 1 => instruction writes on register file
- Reg_Src = 0 => Rs2
- WB_Src = 0 => the data to be write on the register is ALU result
- Stack (Read and Write) = 0 => No stack operations
- Zero_signal = 0 => it has no effect in the instruction
- Mem_out = x

Result of the instruction:

- As shown in the wave form, ALU Result = 0x2, as follow:
- $S3 = 0x3, $S4 = 0x1
- Sub $S5, $S3, $S4 => $S5 = 0x3 – 0x1 = 2 => 0x2
- WB_Data = 0x2

Test Result for Sixth instruction:



Signal name	Value
clk	1
reset	0
pc_src	0
ext_op	1
alu_op	0
alu_src	0
mem_write	1
mem_read	0
reg_write	0
wb_src	0
stack_write	0
stack_read	0
zero_signal	0
pc_out	00000005
ir_out	19060002
a_out	00000003
b_out	00000001
alu_result	00000001
mem_result	xxxxxxxx
wb_result	00000002
reg_src	0

Figure 53: Test Result for Sixth instruction

After execute Sixth instruction: SW \$3, 0(\$4):

- PC_out = Current PC = 0x5
- ExtOp = 1 => Sign extension
- ALU_Op = 0 => Addition (calculate memory address)
- ALU_Src = 0 => extended immediate
- Mem_Read = 0 => prevent read from memory
- Mem_Write = 1 => Write on memory
- Reg_Write = 0 => No write on register file
- Reg_Src = 0 => has no effect (didn't changed)
- WB_Src = 0 => has no effect (didn't changed)
- Stack (Read and Write) = 0 => No stack operations
- Zero_signal = 0 => it has no effect in the instruction
- Mem_out = x

Result of the instruction:

- As shown in the wave form, ALU Result = 0x1, as follow:
- Extended immediate = 0, \$S4 = 1
- Memory address = ALU Result = 0 + \$S4 = 1
- WB_Data = 0x2 (didn't changed) => will not be written

Team Work

We have done all work together; it was all shared between us.

Conclusion

The design for a multi-cycle system must be precise and accurate, and it requires a large number of components.

NOTE: some delays has been added for each execution stage to make a correction of flow of the instruction execution and to prevent glitches, also to give enough time for read and write operations and prevent conflicts.