Department of Electrical & Computer Engineering ENCS4370 - Computer Architecture

Multi Cycle Processor Implementation

Prepared By:

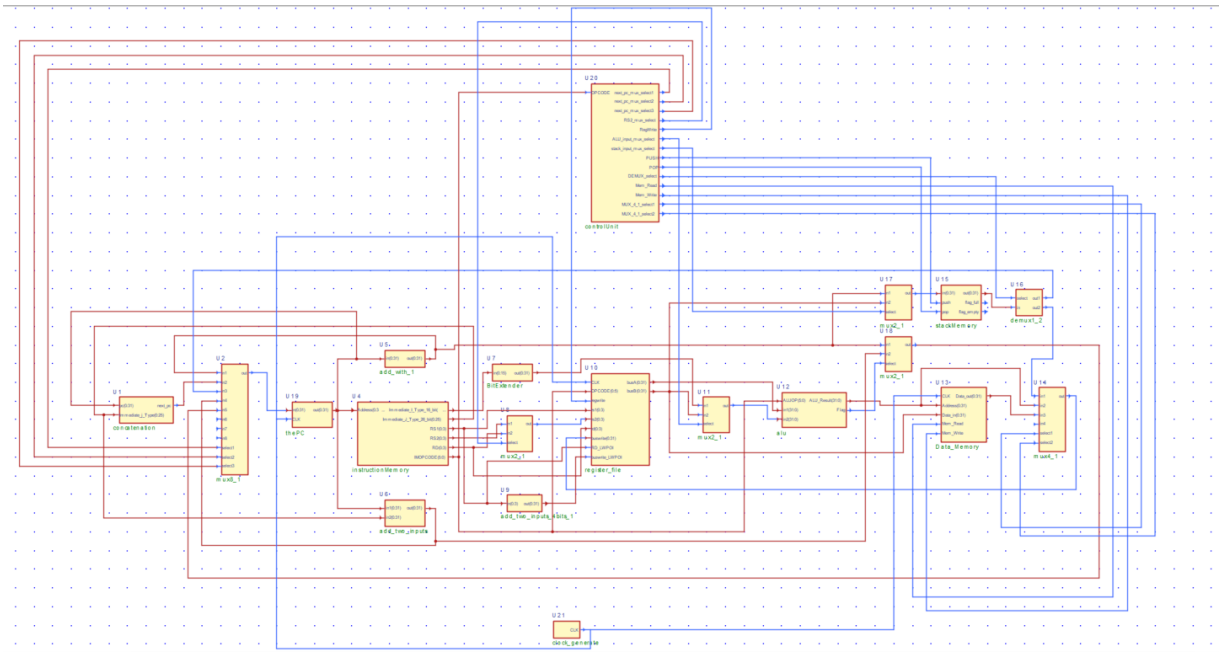
Ayman Qashoo	1200312
Ayman Salamah	1200488
Anas Sarabta	1200242

Instructor: Dr. Ayman Hroub

Date: 1/29/2024

Table of Contents


```
1  module concatenation(input [0:31] pc, input [0:25] Immediate_j_Type, output reg next_pc);
2
3    wire [0:5] last_6_bits_pc; // Wire to store the last 6 bits of pc
4
5    // Extract the last 6 bits from pc
6    assign last_6_bits_pc = pc[26:31];
7
8    // Concatenate the last 6 bits of pc with immediate_j_type
9    assign next_pc = {last_6_bits_pc, Immediate_j_Type};
10
11 endmodule
```



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 and the words size is 32bits
2. 16 32-bit general-purpose registers: from R0 to R15.
3. 32-bit special purpose register for the program counter (PC)
4. 32-bit special purpose register for the stack pointer (SP), which points to the topmost empty element of the stack. This register is visible to the programmer.
5. The program memory layout comprises the following three segments: Static data segment, Code segment, Stack segment. It is a LIFO (Last in First out) data structure. This machine has explicit instructions that enables the programmer to push/pop elements on/from the stack. The stack stores the return address, registers' values upon function calls, etc.
6. The processor has two separate physical memories, one for instructions and the other one for data. The data memory stores both the static data segment and the stack segment.
7. Four instruction types (R-type, I-type, J-type, and S-type).
8. Separate data and instructions memories
9. Word-addressable memory
10. The ALU to calculate the condition branch outcome (taken/ not taken). These signals might include zero, carry, overflow.

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

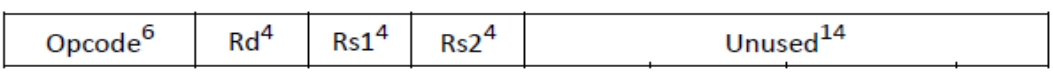


Figure 1: R-Type Format

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

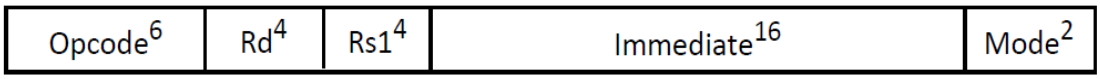


Figure 2: I-Type Format

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

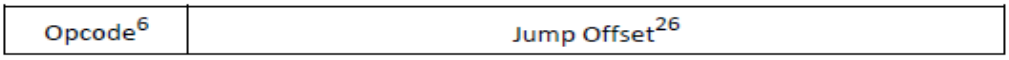


Figure 3: J-Type Format

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

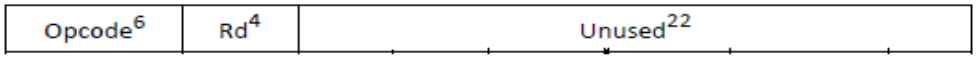


Figure 4: S-Type Format

Instruction Set

- The table below shows the different instructions you are required to implement. It shows their type, the opcode value, and their meaning in RTN (Register Transfer Notation).

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

Table 1: Instruction Set

Data Path Design:

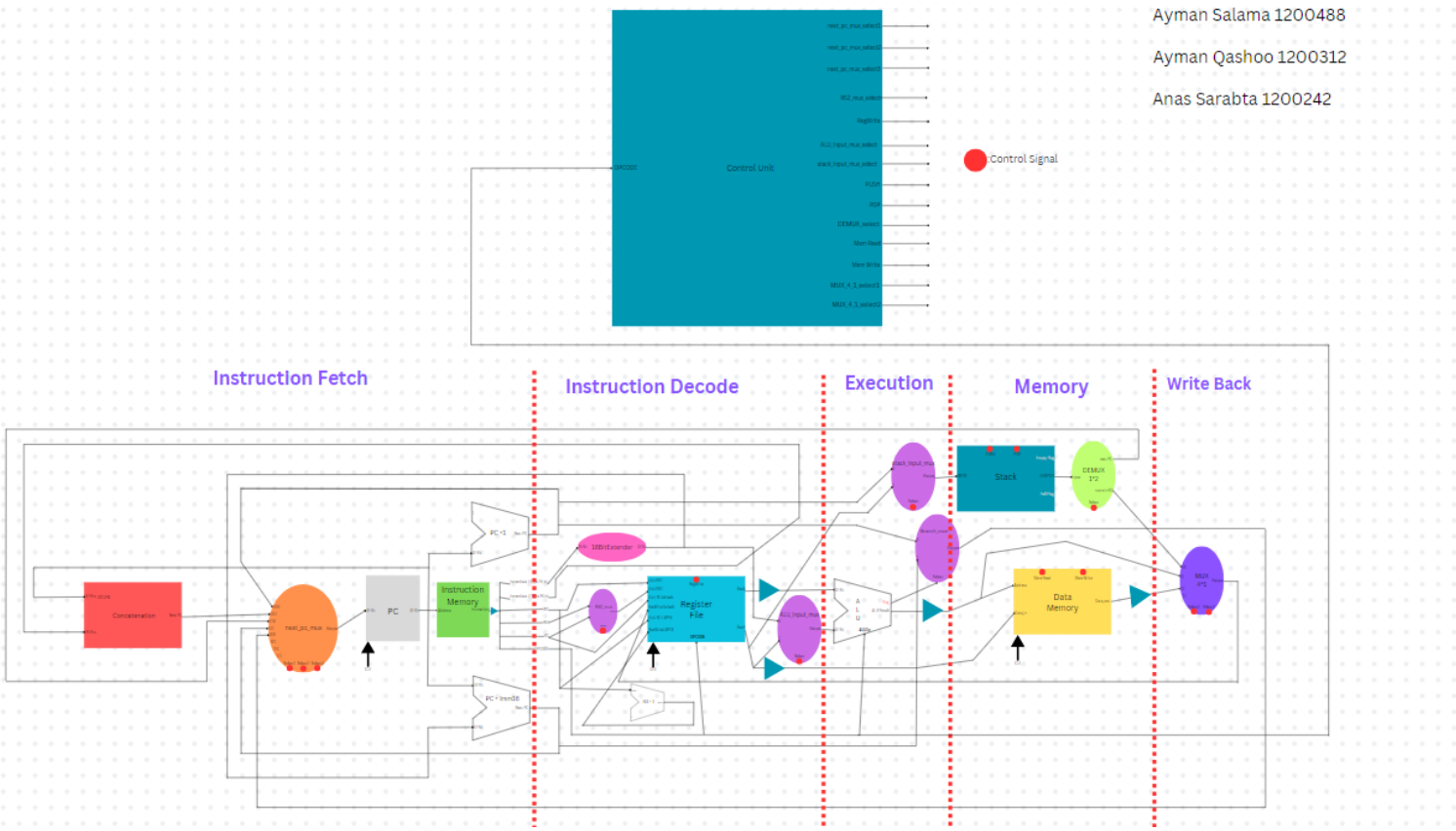


Figure 6: Data Path Design

Control Unit Block

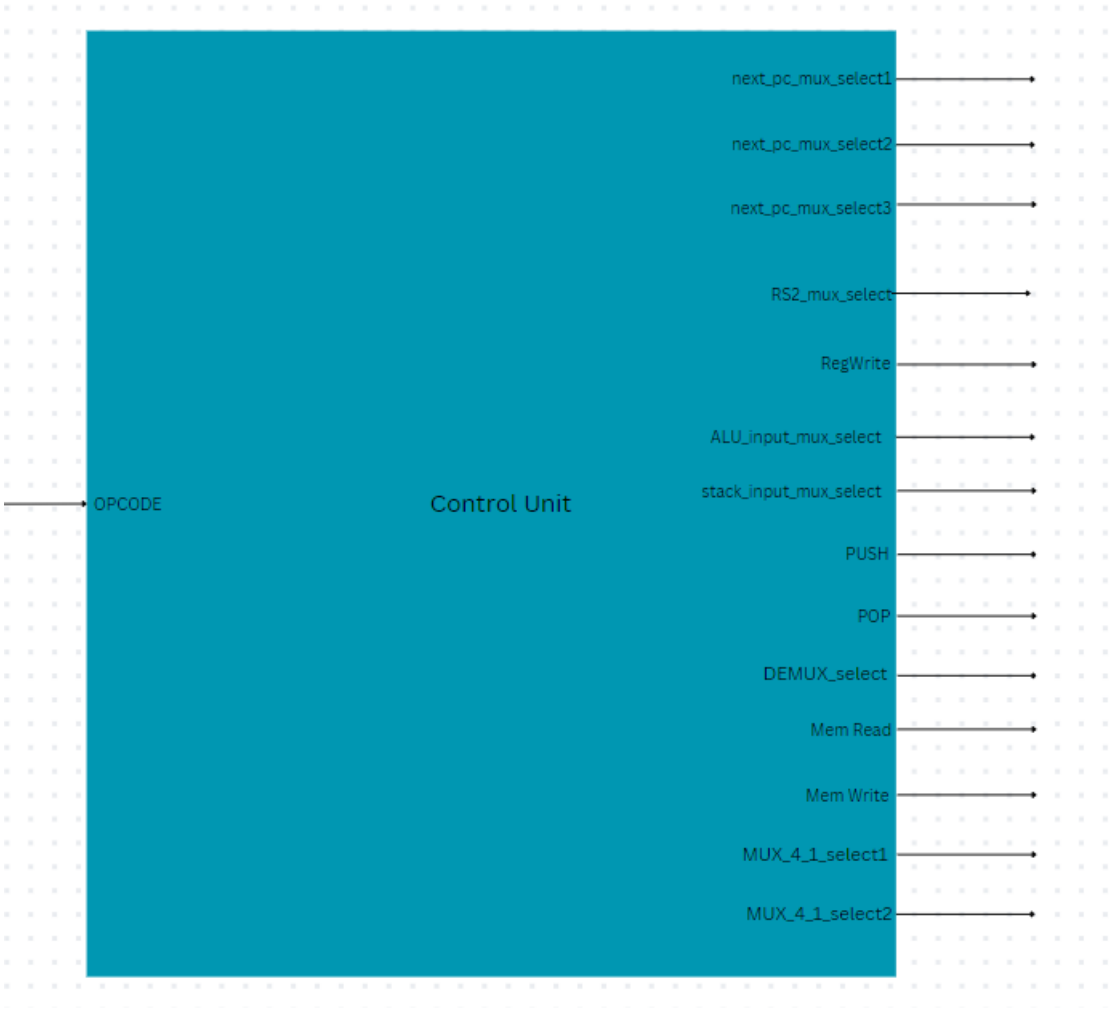


Figure 7: Control Unit

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.

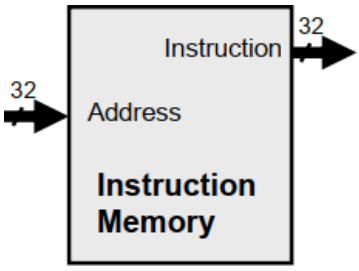


Figure 8: Instruction Memory Module

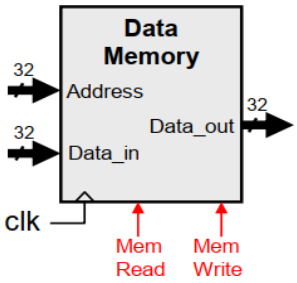


Figure 9: Data Memory Module

Code of instruction memory and data memory

```
module instructionMemory(
    input [0:31] Address,
    output reg [0:15] Immediate_I_Type_16_bit,
    output reg [0:25] Immediate_J_Type_26_bit,
    output reg [0:3] RS1,
    output reg [0:3] RS2,
    output reg [0:3] RD,
    output reg [5:0] IMOPCODE
);
    reg [0:5] opcode;

    initial begin
        Immediate_I_Type_16_bit = 16'b0;
        Immediate_J_Type_26_bit = 26'b0;
        RS1 = 4'b0;
        RS2 = 4'b0;
        RD = 4'b0;
        opcode = 6'b0;
    end

    always @(Address) begin
        opcode = Address[0:5];
       IMOPCODE = opcode;

        case(opcode)
            // R-Type Instructions (0 to 2)
            6'b000000, 6'b000001, 6'b000010: begin
                RD  = Address[6:9]; // 4-bit RD
                RS1 = Address[10:13]; // 4-bit RS1
                RS2 = Address[14:17]; // 4-bit RS2
                // Clear other fields as they are not used by R-type instructions
                Immediate_I_Type_16_bit = 16'b0;
                Immediate_J_Type_26_bit = 26'b0;
            end

            // I-Type Instructions (3 to 11)
            6'b000011, 6'b000100, 6'b000101, 6'b000110,
            6'b000111, 6'b001000, 6'b001001, 6'b001010,
            6'b001011: begin
                RD  = Address[6:9]; // 4-bit RD
                RS1 = Address[10:13]; // 4-bit RS1
                Immediate_I_Type_16_bit = Address[14:29]; // 16-bit immediate
                // Clear other fields as they are not used by I-type instructions
                RS2 = 4'b0;
                Immediate_J_Type_26_bit = 26'b0;
            end
```

Figure 11: Instruction Memory Code

```
module Data_Memory(input CLK,
                   input [0:31] Address,
                   input [0:31] Data_in,
                   input Mem_Read,
                   input Mem_Write,
                   output reg [0:31] Data_out);

    reg [0:31] memory [0:255];

    always  @(posedge CLK)
    begin
        if (Mem_Read)
            begin
            Data_out <= memory[Address];
            end
        else if (Mem_Write)
            begin
            memory[Address] <= Data_in;
            end
    end

    integer i;
    initial
    begin
        for (i = 0; i < 32; i = i + 1)
        begin
            memory[i] = 32'b0;
        end
    end
endmodule: Data_Memory
```

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

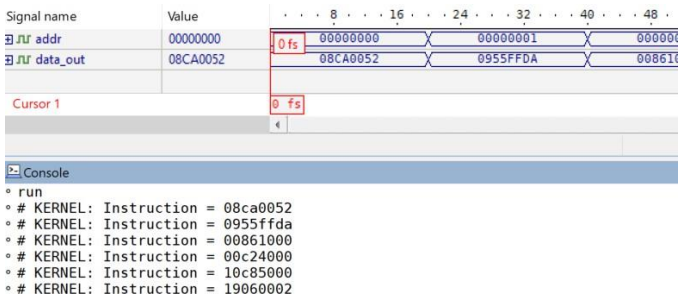


Figure 13: Instruction Memory Test

Test for data memory module

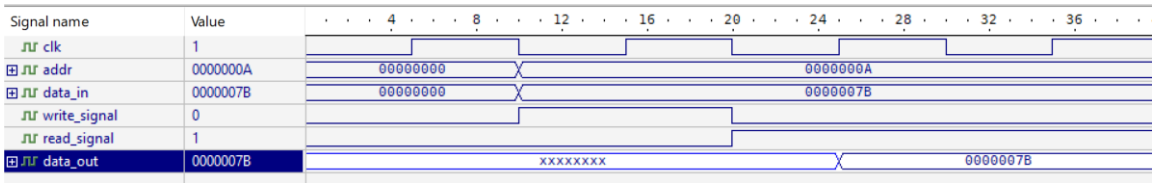


Figure 14: Data Memory Test

Register File

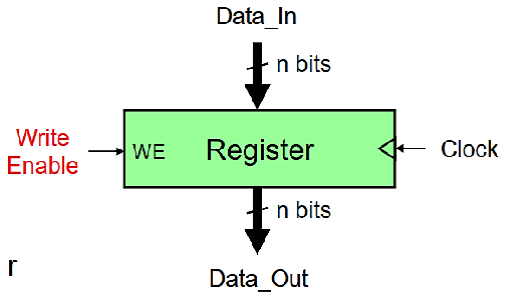


Figure 16: Register Element

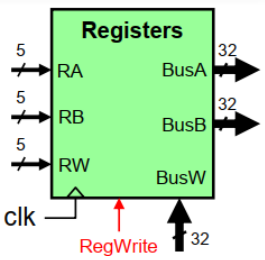


Figure 15: Register File Module

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

```
module register_file(
    input CLK, // Clock signal
    input [0:5] OPCODE,
    input regwrite, // Register Write enable
    input [0:3] rs1, // Read address for BusA
    input [0:3] rs2, // Read address for BusB
    input [0:3] rd, // Write address
    input [0:31] buswrite, // Write data
    input RD_LWPOI,
    input buswrite_LWPOI,
    output [0:31] busA, // Read data for BusA
    output [0:31] busB  // Read data for BusB
);

    // Register array of 16 registers, each 32 bits wide
    reg [0:31] registers[0:15];

    // Read operations (combinational logic)
    assign busA = (rs1 != 4'b0) ? registers[rs1] : 32'b0; // Read register RS1 if not 0
    assign busB = (rs2 != 4'b0) ? registers[rs2] : 32'b0; // Read register RS2 if not 0

    // Write operation (sequential logic)
    always @(posedge CLK) begin
        if (regwrite && (rd != 4'b0)) begin // Write enable and RD is not 0
            registers[rd] <= buswrite;
        end
        if(OPCODE == 5'b000110) begin
            registers[RD_LWPOI] <= buswrite_LWPOI;
        end
    end

    //Initialize the register file to 0 (optional)
    integer i;
    initial begin
        for (i = 0; i < 16; i = i + 1) begin
            registers[i] = 32'b0;
        end
    end

endmodule
```

Figure 17: Register File Code

Test Register File

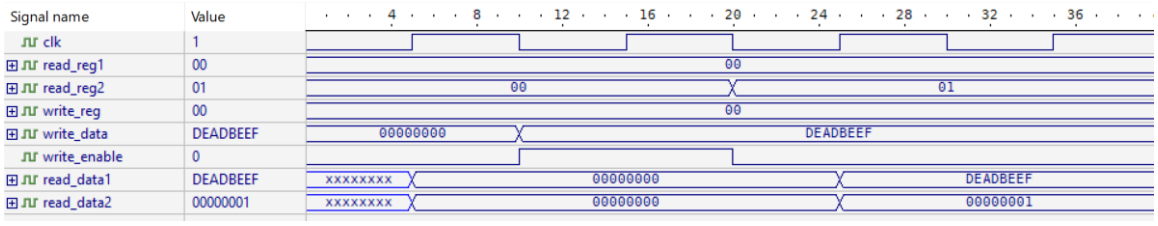


Figure 18: Test Register File

Arithmetic Logical Unit

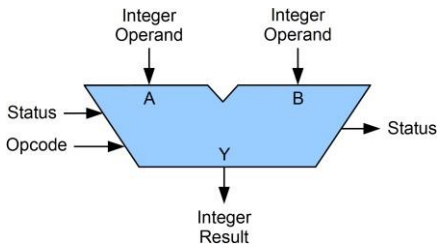


Figure 19: ALU bloc

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 [5:0] ALUOP, // ALU operation code
    input [31:0] in1,      // First operand
    input [31:0] in2,      // Second operand
    output reg [31:0] ALU_Result, // ALU result
    output reg Flag    // Flag for BEQ, BNE, BGT, BLT conditions
);

    // Temporary variables for the subtraction result
    reg [31:0] subtraction_result;
    reg negative_flag;

    always @(*) begin
        Flag = 1'b0;
        ALU_Result = 32'b0; // Default result

        case (ALUOP)
            6'b000000: begin // AND
                ALU_Result = in1 & in2;
            end
            6'b000001: begin // ADD
                ALU_Result = in1 + in2;
            end
            6'b000010: begin // SUB
                ALU_Result = in1 - in2;
            end
            6'b000011: begin // AND
                ALU_Result = in1 & in2;
            end
            6'b000100: begin // ADD
                ALU_Result = in1 + in2;
            end
            6'b000101: begin // ADD
                ALU_Result = in1 + in2;
            end
            6'b000110: begin // ADD
                ALU_Result = in1 + in2;
            end
            6'b000111: begin // ADD
                ALU_Result = in1 + in2;
            end
            // For BEQ, BNE, BGT, BLT: perform subtraction and set the Flag
            6'b001000: begin // BGT
                subtraction_result = in1 - in2;
                negative_flag = subtraction_result[31];
                Flag = !negative_flag && (subtraction_result != 0);
            end
            6'b001001: begin // BLT (Branch if Less Than)
                subtraction_result = in1 - in2;
                negative_flag = subtraction_result[31];
                Flag = negative_flag && (subtraction_result != 0);
            end
            6'b001010: begin // BEQ (Branch if Equal)
                subtraction_result = in1 - in2;
                Flag = (subtraction_result == 32'b0);
            end
            6'b001011: begin // BNE (Branch if Not Equal)
                subtraction_result = in1 - in2;
                Flag = (subtraction_result != 32'b0);
            end
            default: begin
                ALU_Result = 32'b0; // Default result is 0 for unknown opcode
            end
        endcase
    end

endmodule
```

Figure 20: ALU Code

ALU Test

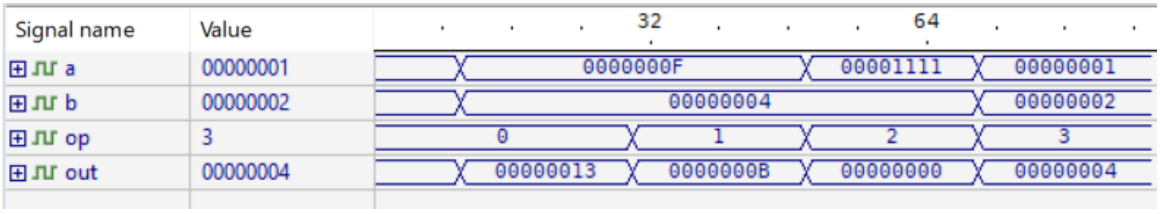


Figure 21: ALU test

Extender:

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

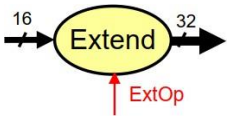


Figure 22: Extender

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 16 to 32-bits Code

```
1  module BitExtender(
2      input [0:15] in,  // 16-bit input
3      output [0:31] out // 32-bit sign-extended output
4  );
5
6      // Sign-extend the input by replicating the sign bit (bit 15) to fill the upper 16 bits
7      assign out = {{16{in[0]}}, in};
8  endmodule
9
```

Figure 23: Signed Extender Code

Signed Extender Test

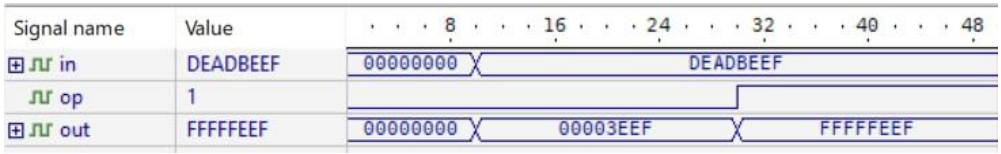


Figure 24: Signed Extender Test

Concatenation:

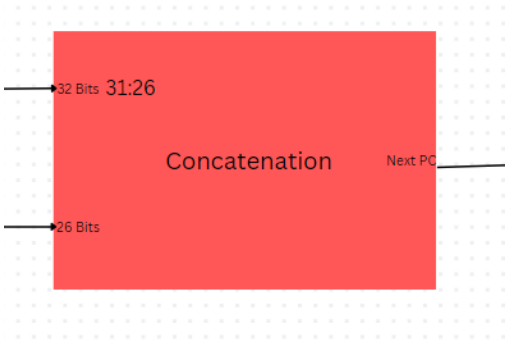


Figure 20: Concatenation

Concatenation Operation { , }: The curly braces {} indicate a concatenation operation. Concatenation in this context means combining two binary numbers end-to-end to form a new binary number. Here, the 6 bits from PC[31:26] are combined with the 26 bits from Immediate26 to form a new 32-bit value.

The purpose of such an operation is usually related to instruction decoding and execution in a processor. In many instruction set architectures, certain instructions require the construction of a target address for jumps or branches. This concatenation could be a part of calculating such a target address.

----> Next PC = {PC[31:26], Immediate26 }

PC[31:26]: This refers to the most significant 6 bits of the Program Counter (PC). The Program Counter is a special register in the CPU that holds the address of the next instruction to be executed. By specifying PC[31:26], we are extracting the upper 6 bits of the PC.

Immediate26: This is a 26-bit immediate value, which typically comes from an instruction. Immediate values are used to encode operands directly within the instruction itself, rather than referring to memory or a register.

Code Concatenation:

```
1  module concatenation(input [0:31] pc, input [0:25] Immediate_j_Type, output reg next_pc);
2
3    wire [0:5] last_6_bits_pc; // Wire to store the last 6 bits of pc
4
5    // Extract the last 6 bits from pc
6    assign last_6_bits_pc = pc[26:31];
7
8    // Concatenate the last 6 bits of pc with immediate_j_type
9    assign next_pc = {last_6_bits_pc, Immediate_j_Type};
10
11 endmodule
```

Figure21: code concatenation

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

```
// a stack conating 32 cells, each is 32 bit
module stackMemory(input [0:31]in, input push, input pop, output reg [0:31]out, output flag_full, output flag_empty);

  reg [4:0] stack_elements; // pointers tracking the stack
  reg [31:0] memory [31:0]; // the stack is 32 bit wide and 32 locations in size

  assign flag_full = (stack_elements == 5'd31) ? 1 : 0;
  assign flag_empty = (stack_elements == 5'd0) ? 1 : 0;

  initial begin
    stack_elements = 5'd0;
  end

  always @(*)
  begin
    if (push & !flag_full)
      begin
        memory[stack_elements] <= in;
        stack_elements <= stack_elements + 1;
      end
    else if (pop & !flag_empty)
      begin
        out <= memory[stack_elements];
        stack_elements = stack_elements - 1;
      end
  end
endmodule
```

Figure 27: Stack Memory Code

Stack Memory Test

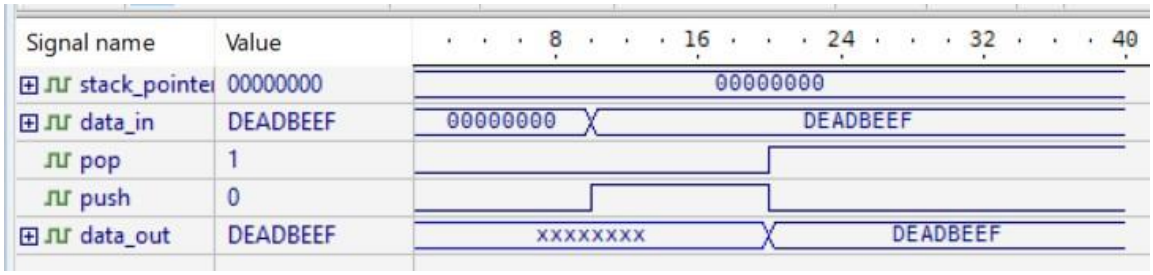


Figure 28: Stack Memory Test

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:

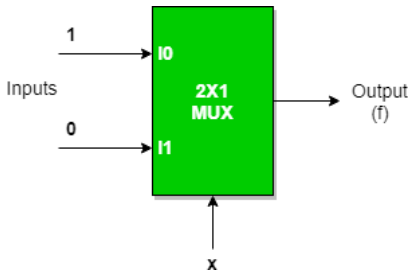


Figure 29: 2x1 Mux Block

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:

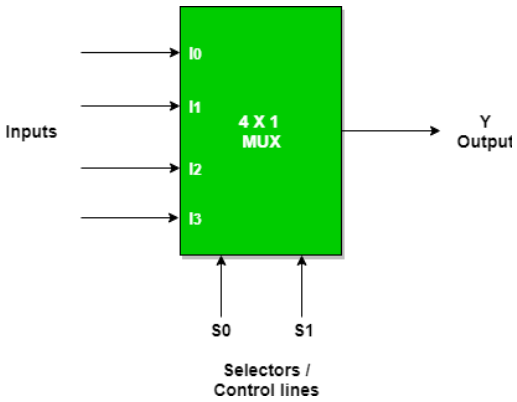


Figure 30: 4x1 Multiplexer Block

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.

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.

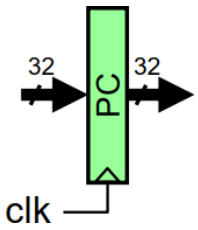


Figure 33: PC Register Block

Control Unit

Main Control Input

- 6-bit opcode field

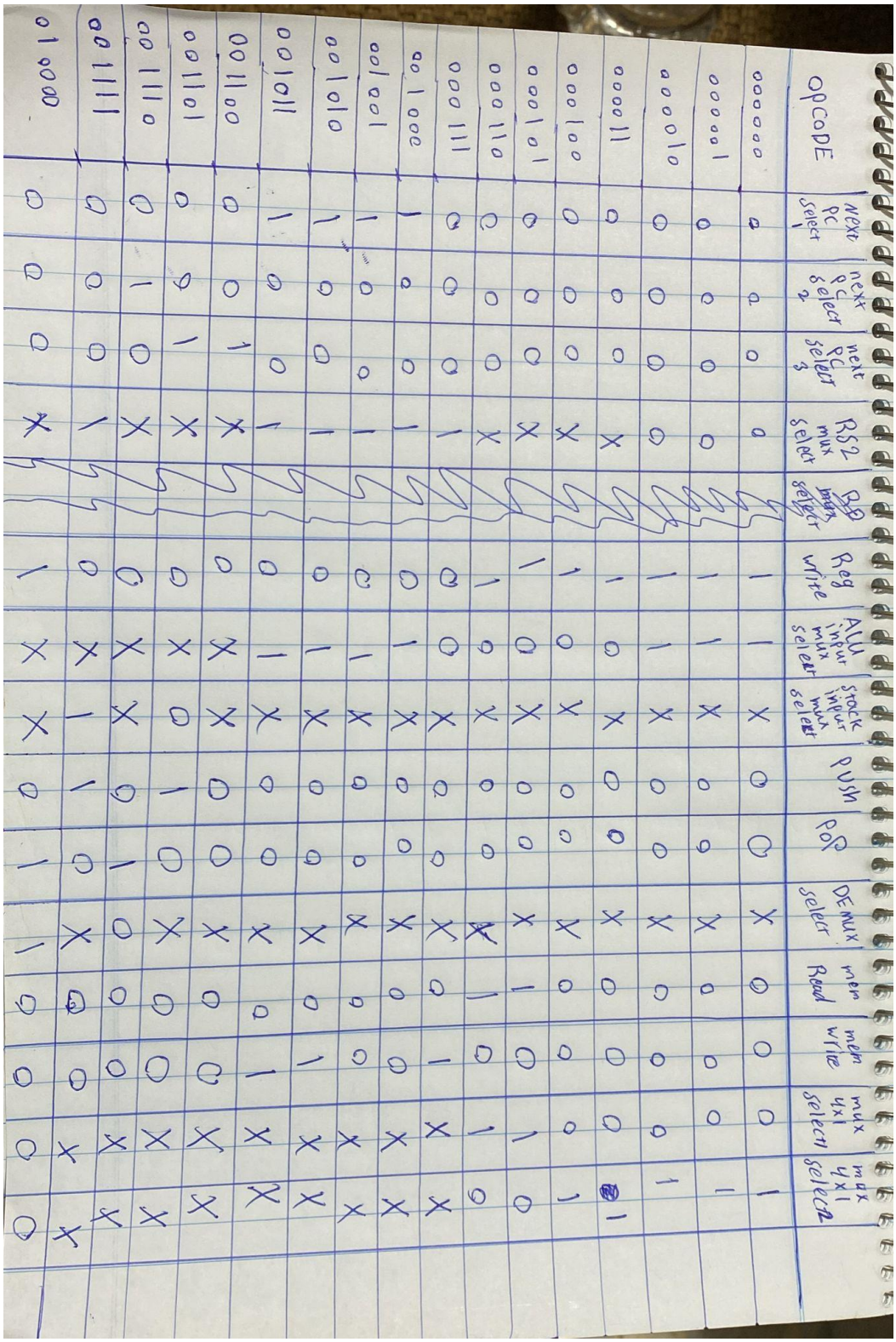
Main Control Output

- Main control signals

Main Control Signal:

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



OPCODE	next PC select 1	next PC select 2	next PC select 3	RS2 mux select	RD input select	Reg write	ALU input mux select	stack input mux select	PUSH	POP	DEMUX select	mem Read	mem write	mux 4x1 select1	mux 4x1 select2
000000	0	0	0	0		1	1	X	0	0	X	0	0	0	1
000001	0	0	0	0		1	1	X	0	0	X	0	0	0	1
000010	0	0	0	0		1	1	X	0	0	X	0	0	0	1
000011	0	0	0	X		1	0	X	0	0	X	0	0	0	1
000100	0	0	0	X		1	0	X	0	0	X	0	0	0	1
000101	0	0	0	X		1	0	X	0	0	X	0	0	0	0
000110	0	0	0	X		1	0	X	0	0	X	1	0	1	0
000111	0	0	0	1		0	0	X	0	0	X	0	0	X	X
001000	1	0	0	1		0	1	X	0	0	X	0	0	X	X
001001	1	0	0	1		0	1	X	0	0	X	0	0	X	X
001010	1	0	0	1		0	1	X	0	0	X	0	0	X	X
001011	1	0	0	1		0	1	X	0	0	X	0	0	X	X
001100	0	0	1	X		0	X	X	0	0	X	0	0	X	X
001101	0	0	1	X		0	X	X	0	0	X	0	0	X	X
001110	0	0	0	X		0	X	X	0	0	X	0	0	X	X
001111	0	0	0	X		0	X	X	0	0	X	0	0	X	X
010000	0	0	0	X		0	X	X	0	0	X	0	0	X	X

Table 3: Main Control Truth Table

equations control unit

push = call || push
pop = ret || pop
memory read = load word || load word.poi
memory write = store word
register write = and || add || sub || andi || addi || lw || lw.poi
rs mux select = bgt || blt || beq || bne || j | push
alu input mux seect = and || add || sub ||Andi ||Addi|| lw|| lw.poi ||pop
demux select = pop
stack input mux select = push
next pc select1 = bgt || blt || beq || bne
next pc select2 = ret
next pc select3 = j || call
mux 4*1 select 1 = lw || lw.poi
mux 4 * 1 select 2 = and || add || sub || andi || addi

Control Unit Code:

```
module controlUnit(input OPCODE, output reg next_pc_mux_select1,
                                 output reg next_pc_mux_select2,
                                 output reg next_pc_mux_select3,
                                 output reg RS2_mux_select,
                                 output reg RegWrite,
                                 output reg ALU_input_mux_select,
                                 output reg stack_input_mux_select,
                                 output reg PUSH,
                                 output reg POP,
                                 output reg DEMUX_select,
                                 output reg Mem_Read,
                                 output reg Mem_Write,
                                 output reg MUX_4_1_select1,
                                 output reg MUX_4_1_select2);

always @(*) begin
            next_pc_mux_select1 = 0;
            next_pc_mux_select2 = 0;
            next_pc_mux_select3 = 0;
            RS2_mux_select = 0;
            RegWrite = 0;
            ALU_input_mux_select = 0;
            stack_input_mux_select = 0;
            PUSH = 0;
            POP = 0;
            DEMUX_select = 0;
            Mem_Read = 0;
            Mem_Write = 0;
            MUX_4_1_select1 = 0;
            MUX_4_1_select2 = 0;
            case (OPCODE)
            6'b000000: begin // AND ------------------------------------
                 next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 0;
                RegWrite = 1;
                ALU_input_mux_select = 1;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
                MUX_4_1_select1 = 0;
                MUX_4_1_select2 = 1;
            end
            6'b000001: begin // ADD ------------------------------------
                 next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 0;
                RegWrite = 1;
                ALU_input_mux_select = 1;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
                MUX_4_1_select1 = 0;
                MUX_4_1_select2 = 1;
            end
            6'b000010: begin // SUB ------------------------------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 0;
                RegWrite = 1;
                ALU_input_mux_select = 1;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
                MUX_4_1_select1 = 0;
                MUX_4_1_select2 = 1;
            end
            6'b000011: begin // ANDI -----------------------------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RegWrite = 1;
                ALU_input_mux_select = 0;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
                MUX_4_1_select1 = 0;
                MUX_4_1_select2 = 1;
            end
            6'b000100: begin // ADDI -----------------------------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RegWrite = 1;
                ALU_input_mux_select = 0;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
                MUX_4_1_select1 = 0;
                MUX_4_1_select2 = 1;
            end
            6'b000101: begin // LW -------------------------------------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RegWrite = 1;
                ALU_input_mux_select = 0;
                PUSH = 0;
                POP = 0;
                Mem_Read = 1;
                Mem_Write = 0;
                MUX_4_1_select1 = 1;
                MUX_4_1_select2 = 0;
            end
            6'b000110: begin // LW.POI ---------------------------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RegWrite = 1;
                ALU_input_mux_select = 0;
                PUSH = 0;
                POP = 0;
                Mem_Read = 1;
                Mem_Write = 0;
                MUX_4_1_select1 = 1;
                MUX_4_1_select2 = 0;
            end
            6'b000111: begin // SW ---------------------------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 1;
                RegWrite = 0;
                ALU_input_mux_select = 0;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 1;
            end
            // For BEQ, BNE, BGT, BLT: perform subtraction and set the Flag
            6'b001000: begin // BGT  -------------------------------------------
                next_pc_mux_select1 = 1;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 1;
                RegWrite = 0;
                ALU_input_mux_select = 1;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b001001: begin // BLT (Branch if Less Than) ------------
                next_pc_mux_select1 = 1;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 1;
                RegWrite = 0;
                ALU_input_mux_select = 1;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b001010: begin // BEQ (Branch if Equal) -----------------
                next_pc_mux_select1 = 1;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 1;
                RegWrite = 0;
                ALU_input_mux_select = 1;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b001011: begin // BNE (Branch if Not Equal) -------------
                next_pc_mux_select1 = 1;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 1;
                RegWrite = 0;
                ALU_input_mux_select = 1;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b001100: begin // JMP (Branch if Not Equal) -------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 1;
                RegWrite = 0;
                PUSH = 0;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b001101: begin // CALL (Branch if Not Equal) ------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 1;
                RegWrite = 0;
                stack_input_mux_select = 0;
                PUSH = 1;
                Mem_Write = 0;
            end
            6'b001101: begin // CALL (Branch if Not Equal) ------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 1;
                RegWrite = 0;
                stack_input_mux_select = 0;
                PUSH = 1;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b001110: begin // RET (Branch if Not Equal)  ------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 1;
                next_pc_mux_select3 = 0;
                RegWrite = 0;
                PUSH = 0;
                POP = 1;
                DEMUX_select = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b001111: begin // PUSH (Branch if Not Equal)-------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RS2_mux_select = 1;
                RegWrite = 0;
                stack_input_mux_select = 1;
                PUSH = 1;
                POP = 0;
                Mem_Read = 0;
                Mem_Write = 0;
            end
            6'b010000: begin // POP (Branch if Not Equal)--------------
                next_pc_mux_select1 = 0;
                next_pc_mux_select2 = 0;
                next_pc_mux_select3 = 0;
                RegWrite = 1;
                PUSH = 0;
                POP = 1;
                DEMUX_select = 1;
                Mem_Read = 0;
                Mem_Write = 0;
                MUX_4_1_select1 = 0;
                MUX_4_1_select2 = 0;
            end
    endcase
    end
    endmodule
```

Figure 37: Main Control Unit Code

Main Control Unit Test:

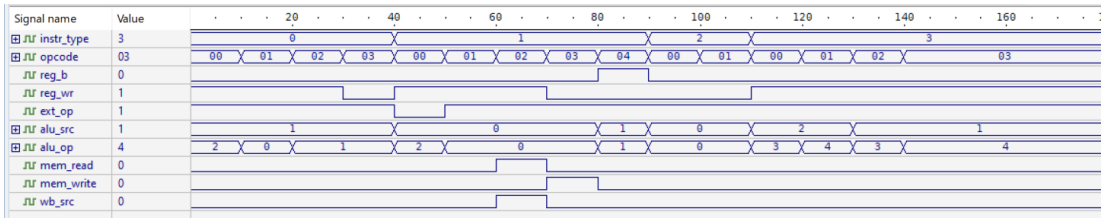
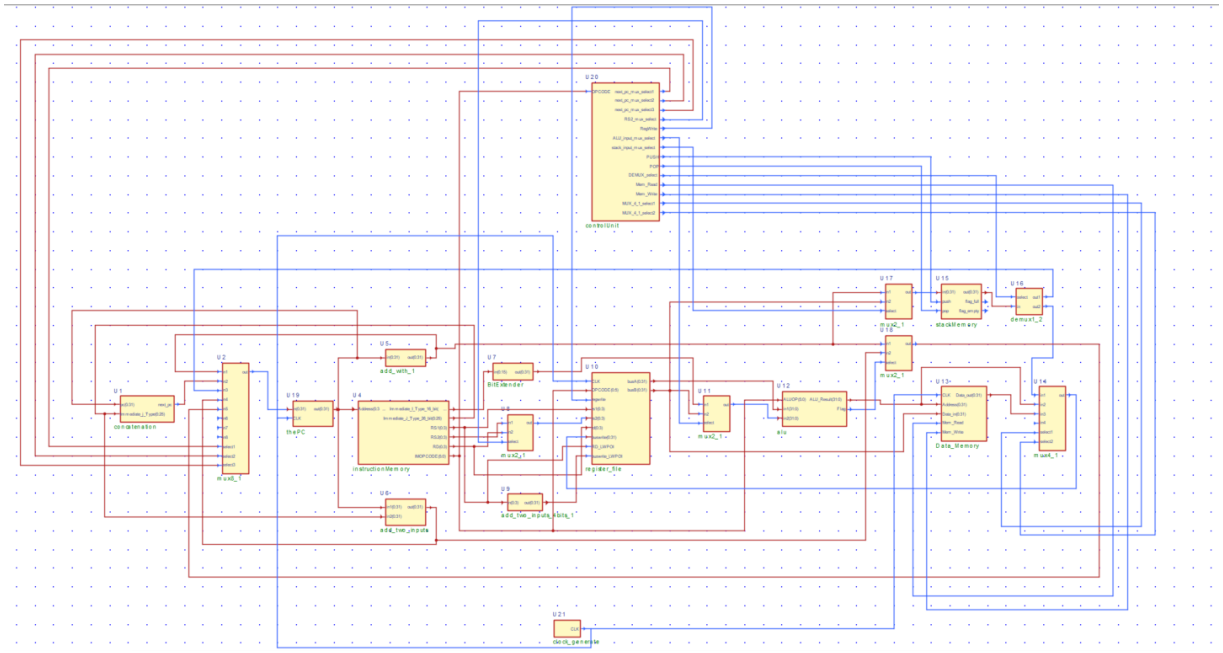


Figure 38: Main Control Unit Test

Figure 40: PC Source Block Diagram

Testing

Connection of components for test:



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.