Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Networks– ENCS3320

Packet Tracer Project Report

Prepared by:

Nada AbuSaif 1200202 Section 2

Dana Ghnimat 1200031 Section 1

Shahd Ali 1200183 Section 1

Instructor: Dr. Ayman Hroub

Dr. Aziz Qaroush

Section: 2 and 1

Date: 29/1/2024

Abstract

The project aims to design and verify a pipelined RISC processor using Verilog The program contains Control signal generator, Datapath design, RTL implementation and verification through test bench that insure us that the projects runs correctly.

Table of Contents

Truth Table:

instruction	function	PCSrs1	PCsrs0	push	Reg W	Ext 16	Ext26	ALU SR0	ALU OP	Rmem	Wmem	WB
AND	000000	1	0	0	1	x	x	0	10	0	0	1
ADD	000001	1	0	0	1	x	x	0	00	0	0	1
SUB	000010	1	0	0	1	x	x	0	01	0	0	1
ANDI	000011	1	0	0	1	1	x	1	10	0	0	1
ADDI	000100	1	0	0	1	1	x	1	00	0	0	1
LW	000101	1	0	0	1	1	x	1	xx	1	0	0
LWPOI	000110	1	0	0	1	1	x	1	xx	1	0	x
SW	000111	1	0	0	0	1	x	1	xx	0	1	x
BGT	001000	0	0	0	0	1	x	1	xx	0	0	x
BLT	001001	0	0	0	0	1	x	1	xx	0	0	x
BEQ	001010	0	0	0	0	1	x	1	xx	0	0	x
BNE	001011	0	0	0	0	1	x	1	xx	0	0	x
JMP	001100	0	1	0	0	x	x	0	xx	0	0	x
CALL	001101	0	1	1	0	x	x	0	xx	0	0	x
RET	001110	0	1	0	0	0	0	x	x	x	x	x
PUSH	001111	x	x	1	x	x	x	x	x	x	x	x
POP	010000	x	x	0	1	x	x	x	x	x	x	x

Boolean Equations and signals:

PCcont1, PCcont2: when this control signal is equal to

• 00 then PC = Imm_B + PC

• 01 then PC = Imm_j + PC

• 10 then PC = PC + 1

Push: when this control signal is equal to

• 0 then stack[top] = PC + 4

RegW: when this signal is:

• 0 then writing on register will be performed else not.

Ext14: when this control signal is equal to:

• 0 then it extends the immediate number with zero bits. When it is equal to 1, it extends the number with ones. Otherwise (means X) it will do nothing.

Ext24: when this control signal is equal to:

• 0 then it extends the immediate number with zero bits. When it is equal to 1, it extends the number with ones. Otherwise (means X) it will do nothing

ALUSrc1,ALUSrc2: when the bits are

• 00: The source is Rs2

• 01: The source is the extended immediate value

• 11: The source is Rd

ALUSrc1,ALUSrc2: when the bits are

• 000: The operation is ‘ADD’

• 001: The operation is ‘SUB’

• 010: The operation is ‘AND’

• xxx: it does not do any operation

Rmem, Wmem: when this control signal is equal to

• 00: No read, no write on data memory

• 01: write on the data memory

• 10: read on the data memory

Wdata: when 0 the output of the data memory will be written back while 1 is for the ALU operations

PCcont1: (instruction == R-type) | (instruction == S-type)

PCcont2: instruction == J | instruction == JAL

Push: Instruction == JAL

RegW: instruction == AND | instruction == ADD | instruction == Sub | Instruction == ADDI |

Instruction = ANdI | Instruction == LW | (Instruction == S-typr)

Ext14: (Instruction == I-type)

Ext24: (Instruction == J-type)

ALUsrc1: Instruction == BEQ | Instruction == SLL | Instruction == SLR

ALUsrc2: (Instruction == I-type)

ALU_OP[0]: (Instruction == S-type)

ALU_OP[1]: Instruction == AND | Instruction == ANDI | Instruction == SLR | Instruction ==

SLRV

ALU_OP[2]: Instruction == SUB | Instruction == CMP | Intruction == BEG | Instruction ==

SLL | Instruction == SLLV

Rmem: Instruction == LW

Wmem: Instruction == SW

Wdata: Instruction == AND | Instruction == ADD | Instruction == SUB | Instruction == ANDI |

Instruction == ADDI | (Instruction == S-type)

Procedure and explanation:

In short here are the stages are the states in which:
Instruction Fetch: The processor fetches the instruction from memory and loads it into the instruction register.

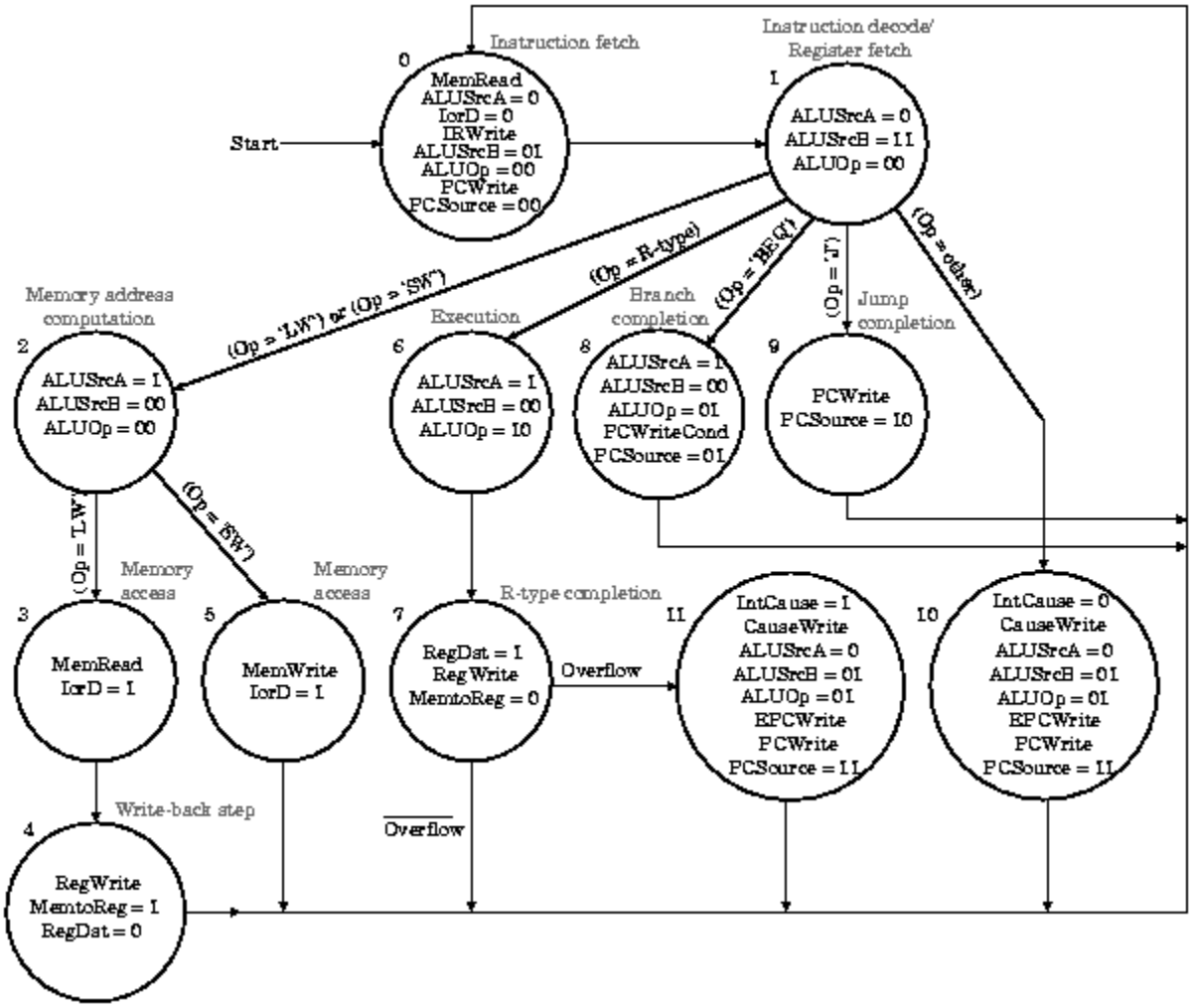
Instruction Decode: The processor decodes the instruction, determines the type of

instruction
and the operands needed, and stores the relevant information in internal registers.

Execution: The processor performs the necessary operations, which may involve multiple clock
cycles, to complete the instruction.

Memory Access: If the instruction requires data to be read from or written to memory, the processor accesses the memory to perform the operation
Write Back: The processor writes the result of the operation back to the appropriate internal
register or memory location.

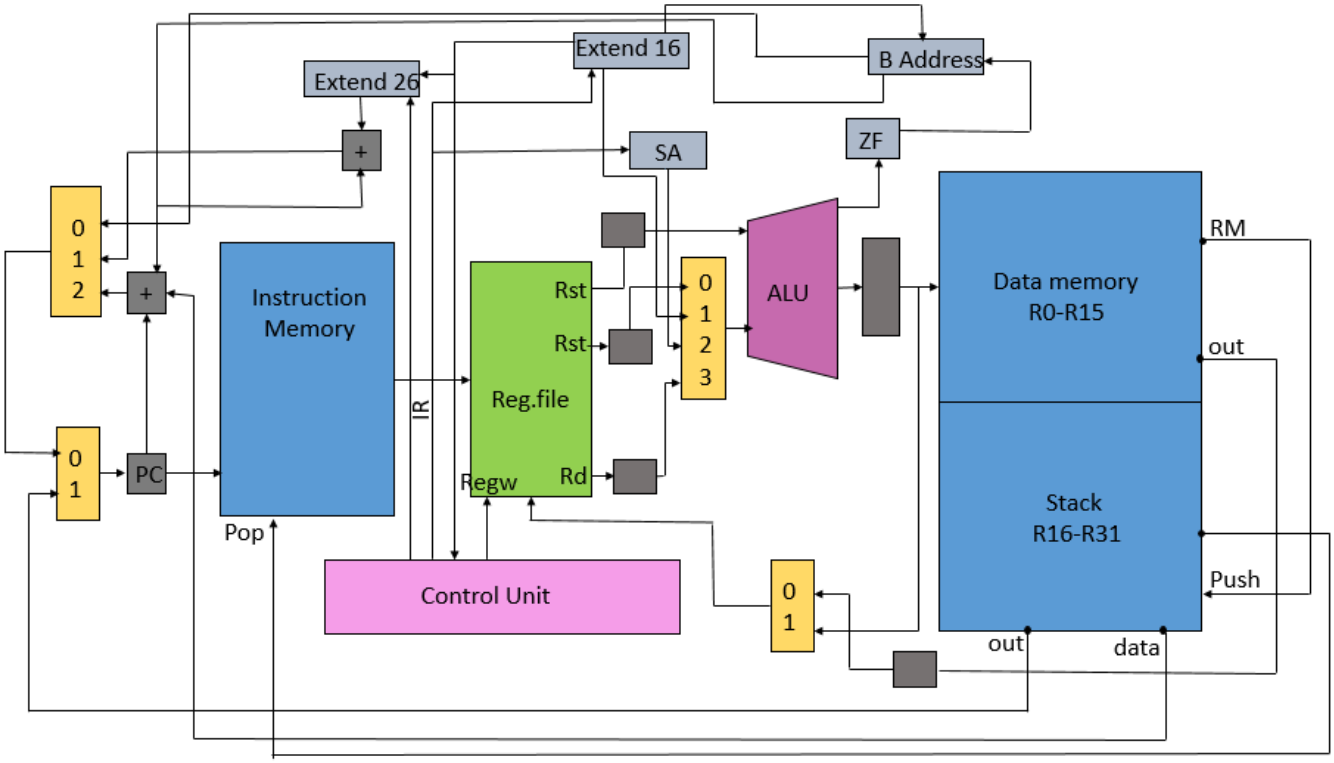
The finite state machine below shows the transition between stages from fetch to write back and
the behavior of each instruction respectively



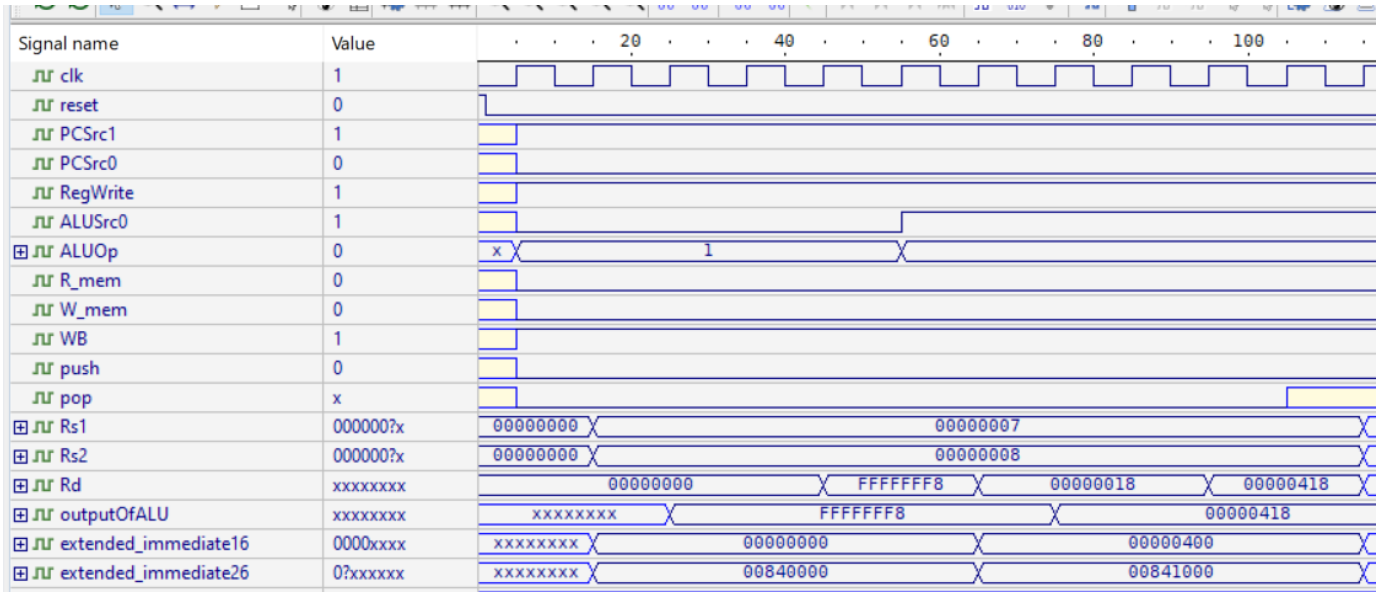
Stages:

The table of control unit appears the comparing esteem to control flag and instruction actualized. Whereas the figure over appears the move per instruction, for case 'and, include, sub' informational being get decoded executed and compose back to enroll record at that point return to bring state for the following instruction, whereas 'j' is being brought decoded and after that return to bring once more.

Datapath:

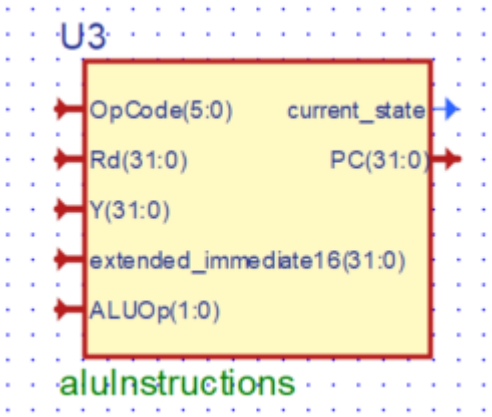

Extend 16
B Address
Extend 26
SA
ZF
+
0
1
2
+
Instruction
Memory
Rst
Rst
Reg.file
Regw
Rd
0
1
2
3
ALU
Data memory
R0-R15
RM
out
Stack
R16-R31
Push
out
data
IR
0
1
PC
Pop
Control Unit
0
1

Final execution for the system:



Instruction memory:

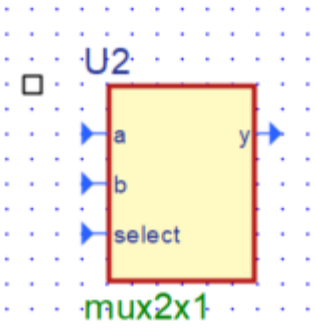
An essential part of a computer system's data path is the instruction memory, also known as the instruction cache. It ensures quick access and effective processing by storing and supplying the instructions for execution. It is essential to the operation of the program.



Mux2x1:

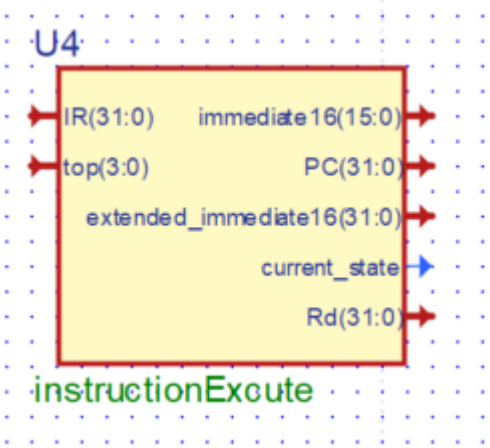
A crucial part of a computer system's data path is the MUX 2x1, or 2-to-1 multiplexer.

Based on a select signal, it chooses one input signal from a pair. It is frequently utilized in the data path for operations like routing and data selection. By enabling the selection of various data sources according to control signals, it improves the data path's functionality and flexibility.

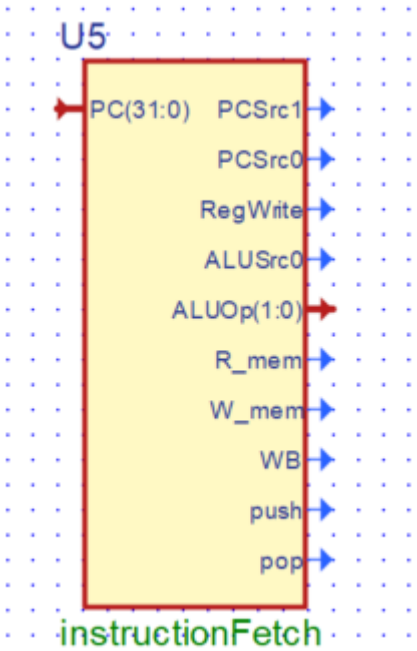


Instruction executes:

To execute the instruction, after fetching it and setting the data signals:

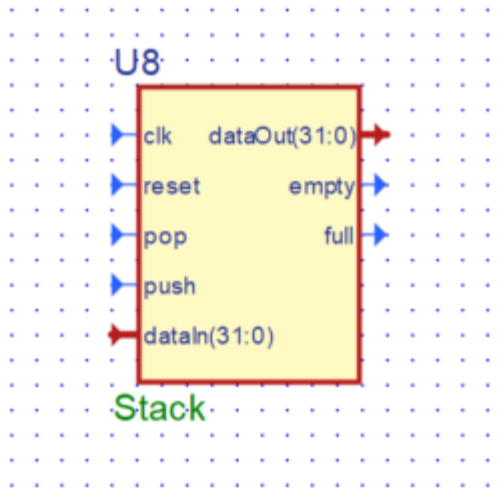


Instruction fetch:



Fetch the instruction from PC address

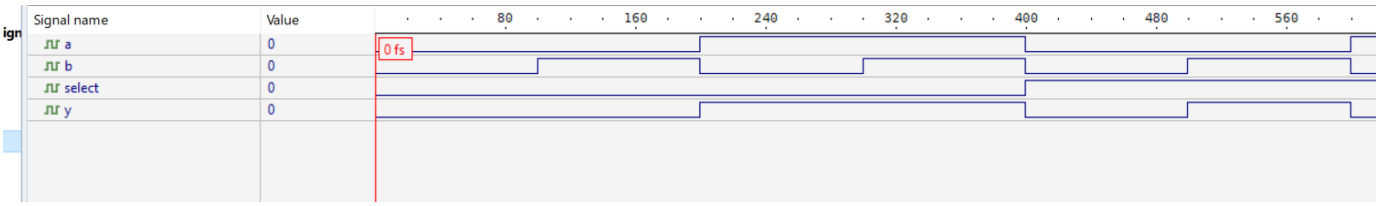
Stack:



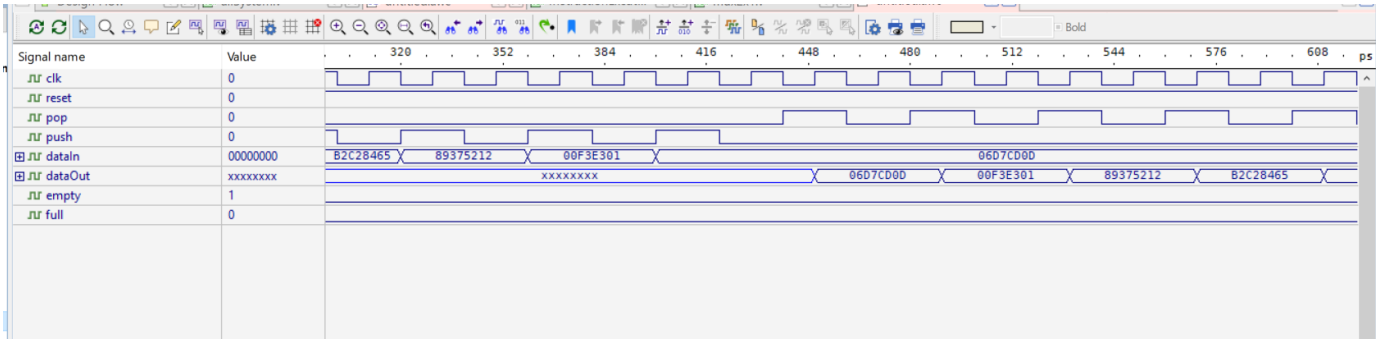
Used push and pop modes to get the address of a specific data from the top of the stack.

Verification The Component of Data path:

2x1Mux:



For stack:



Conclusion

The project involves demonstrating the construction of a complete data pipeline based on specified instructions.

Each stage has been developed with the necessary control signals to regulate their operation.

Simulations are performed for each step as well as design work to create the necessary data paths and final state of the machine, all of which are detailed in the report.

Teamwork We worked together to complete this project, first we started by thinking about the design for each part, sharing ideas and correcting each other's mistakes, then Then we agreed on the design and put the signal into the controller board.

Then one of us built and tested the PC, IF/ID and register file, one built and tested the decode condition, sub condition, add condition and ALU controller and the latest set of ALUs, controllers and extensions built.

Finally, when all the devices are ready and the is working properly, we all put in the effort and build the pipeline together data, then test to make sure it works as required.

Appendix:

Instruction fetch stage:

```
module instructionFetch(
    input wire [31:0] PC,
    output reg PCSrc1,
    output reg PCSrc0,
    output reg RegWrite,
    output reg ALUSrc0,
    output reg [1:0] ALUOp,
    output reg R_mem,
    output reg W_mem,
    output reg WB,
    output reg push,
```

```
    output reg pop
);

reg [31:0] memory[0:15];
reg [31:0] IR;
reg [5:0] OpCode;

always @(PC) begin
    IR = memory[PC]; // Fetch instruction from memory using PC

    // Extract the opcode from the instruction
    OpCode = IR[31:26];

                //$display ("%b", OpCode);



                //pop signal
                pop = IR[0];



                /// R-type
                if (OpCode == 6'b000000 || OpCode == 6'b000001 || OpCode == 6'b000010)
begin ///ADD        || AND || SUB
```

```
                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
            RegWrite = 1'b1;
                ALUSrc0 = 1'b0;

                if (OpCode == 6'b000000)
                    ALUOp = 2'b10;
                else if (OpCode == 6'b000001)
                    ALUOp = 2'b00;
                else
                    ALUOp = 2'b01;

        R_mem = 1'b0;
        W_mem = 1'b0;
        WB = 1'b1;
                push = 1'b0;
    end

    /// I-Type
    else if (OpCode == 6'b000011 || OpCode == 6'b000100) begin      ///ANDI ||
ADDI

                PCSrc1 = 1'b1;
```

```
            PCSrc0 = 1'b0;
        RegWrite = 1'b1;
            ALUSrc0 = 1'b1;

            if (OpCode == 6'b000011)   // ANDI
                ALUOp = 2'b10;
            else
                ALUOp = 2'b00;


    R_mem = 1'b0;
    W_mem = 1'b0;
    WB = 1'b1;
            push = 1'b0;
end

else if (OpCode == 6'b000101) begin //LW

            PCSrc1 = 1'b1;
            PCSrc0 = 1'b0;
        RegWrite = 1'b1;
            ALUSrc0 = 1'b1;
            ALUOp = 2'b00;
    R_mem = 1'b1;
```

```
        W_mem = 1'b0;
                push = 1'b0;
                WB = 1'b0;
                push = 1'b0;
end

else if (OpCode == 6'b000111) begin //SW

                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b00;
        R_mem = 1'b0;
        W_mem = 1'b1;
                push = 1'b0;
end

else if (OpCode == 6'b001010) begin //BEQ

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
```

```
                ALUOp = 2'b01;
        R_mem = 1'b0;
        W_mem = 1'b0;
                push = 1'b0;
end

else if (OpCode == 6'b001000) begin //BGT
                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b01;    /// We make new ALU op for Greater than
        R_mem = 1'b0;
        W_mem = 1'b0;
                push = 1'b0;

end


else if (OpCode == 6'b001001) begin //BLT

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
```

```
            ALUSrc0 = 1'b1;
            ALUOp = 2'b01;          // We make new ALU for less than
        R_mem = 1'b0;
        W_mem = 1'b0;
            push = 1'b0;

end

else if (OpCode == 6'b001011) begin //BNE

            PCSrc1 = 1'b0;
            PCSrc0 = 1'b0;
          RegWrite = 1'b0;
            ALUSrc0 = 1'b1;
            ALUOp = 2'b01;
        R_mem = 1'b0;
        W_mem = 1'b0;
            push = 1'b0;


end
```

```
/// J-Type
else if (OpCode == 6'b001100) begin   // J

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b1;
            RegWrite = 1'b0;
  R_mem = 1'b0;
        W_mem = 1'b0;
            push = 1'b0;
          end

else if(OpCode == 6'b001101) begin    // Call

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b1;
            RegWrite = 1'b0;
        R_mem = 1'b0;
        W_mem = 1'b0;
                push = 1'b1;
          end
else if (OpCode == 6'b001110) begin   // RET

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b1;
```

```
            RegWrite = 1'b0;
        R_mem = 1'b0;
        W_mem = 1'b0;



          end


else if (OpCode == 6'b001111) begin/// S-Type for stack    push

                push = 1'b1;

end



else if (OpCode == 6'b010000) begin          //pop

                pop = 1'b1;
                RegWrite = 1'b1;
end
```

```
end


endmodule
```

instruction execute:

```
module instructionExcute(
    input [31:0] IR,
    input [3:0] top,  // Assuming top is 4 bits


    output reg [15:0] immediate16,
    output reg [31:0] PC,
    output reg [31:0] extended_immediate16,
    output reg current_state,
    output reg [31:0] Rd
);
    // Registers
    reg [31:0] Rs1;
    reg [31:0] Rs2;
    reg [31:0] AdderRd;
    reg [31:0] AdderRs1;
    reg [31:0] AdderRs2;
    reg [31:0] AdderResult;
    reg [31:0] outputOfUpAdder;
```

```
reg [31:0] stack [0:15]; // Assuming STACK_DEPTH is defined
reg empty, full;
reg push, pop; // Added these signals


// Extenders
reg [31:0] immediate26;
reg [31:0] extended_immediate26;

// Mux output
reg [31:0] y;

// Extract fields from IR
assign immediate16 = IR[17:2];
assign extended_immediate16 = {16'b0000000000000000, immediate16};
assign immediate26 = IR[25:0];
assign extended_immediate26 = {6'b000000, immediate26};

// Assign addresses for registers
assign AdderRd = IR[25:22];
assign AdderRs1 = IR[21:18];
assign AdderRs2 = IR[17:14];

// Choose a value for the mux
```

```
mux2x1 m(Rs2, immediate16, ALUSrc0, y);

// Check the current instruction
always @* begin
    case (IR[31:26])
        6'b001100: begin // Jump (J)
            current_state = 4;
            outputOfUpAdder = PC + extended_immediate26;
            PC = outputOfUpAdder;
        end
        6'b001101: begin // Call
            current_state = 4;
            empty = (top == 0);
            full = (top == 15);

            // Push operation
            if (push && !pop && !full) begin
                stack[top] = PC;
          //      top = top + 1;
            end

            PC = PC + extended_immediate26;
        end
        6'b001110: begin // Return
```

```
            current_state = 4;
            PC = stack[top];
        end
        6'b001111: begin // Push Rd
            current_state = 4;
            empty = (top == 0);
            full = (top == 15);

            // Push Rd
            if (push && !pop && !full) begin
                stack[top] = Rd;
             //   top = top + 1;
            end
        end
        6'b010000: begin // Pop Rd
            current_state = 4;
            Rd = stack[top]; // Store the top of the stack in Rd
        //    top = top - 1;
        end
        default: begin
            // Handle other instructions if needed
        end
    endcase
end
```

```
endmodule
```

Alu operation:

```
module aluInstructions(
    input [5:0] OpCode,
        input [31:0] Rd,
        input [31:0] Y,
    input [31:0] extended_immediate16,
    input [1:0] ALUOp,
    output reg current_state,
    output reg [31:0] PC
);

    reg [31:0] operandA;
    reg [31:0] operandB;
    reg [31:0] outputOfALU;
    reg zero;
    reg greater_than;
    reg less_than;
    reg [31:0] outputOfBTA;
    // Assign operands
```

```
assign operandA = Rd;
assign operandB = Y;

// Perform ALU operation based on ALUOp
always @* begin
    case (ALUOp)
        2'b00: outputOfALU = operandA + operandB;   // Addition
        2'b01: outputOfALU = operandA - operandB;   // Subtraction
        2'b10: outputOfALU = operandA & operandB;   // Bitwise AND
        default: outputOfALU = 32'b0;               // Default case: result is 0
    endcase
end

// Set flags
always @* begin
    zero = (outputOfALU == 32'b0);                // Set zero flag if the result is zero
    greater_than = (outputOfALU > 32'h0);         // Set greater than flag
    less_than = (outputOfALU < 32'h0);            // Set less than flag
end

// Check instructions
always @* begin
    outputOfBTA = 32'b0;                          // Initialize outputOfBTA
```

```
// Check BGT instruction
if (OpCode == 6'b001000) begin
    if (greater_than) begin
        current_state = 4;
        outputOfBTA <= PC + extended_immediate16;
        PC <= outputOfBTA;
    end else begin
        PC <= PC + 1;
        current_state = 4;
    end
end

// Check BLT instruction
if (OpCode == 6'b001001) begin
    if (less_than) begin
        current_state = 4;
        outputOfBTA <= PC + extended_immediate16;
        PC <= outputOfBTA;
    end else begin
        PC <= PC + 1;
        current_state = 4;
    end
end
```

```
    // Check BEQ instruction
    if (OpCode == 6'b001010) begin
        if (zero) begin
            current_state = 4;
            outputOfBTA <= PC + extended_immediate16;
            PC <= outputOfBTA;
        end else begin
            PC <= PC + 1;
            current_state = 4;
        end
    end

    // Check BNE instruction
    if (OpCode == 6'b001011) begin
        if (!zero) begin
            current_state = 4;
            outputOfBTA <= PC + extended_immediate16;
            PC <= outputOfBTA;
        end else begin
            PC <= PC + 1;
            current_state = 4;
        end
    end
end
```

```
endmodule


module aluInstructions_tb();

    // Inputs
    reg [5:0] OpCode;
    reg [31:0] Rd;
    reg [31:0] Y;
    reg [15:0] extended_immediate16;
    reg [1:0] ALUOp;

    // Outputs
    reg current_state;
    reg [31:0] PC;

    // Instantiate the ALU instructions module
    aluInstructions dut (
        .OpCode(OpCode),
        .Rd(Rd),
        .Y(Y),
        .extended_immediate16(extended_immediate16),
        .ALUOp(ALUOp),
        .current_state(current_state),
```

```
    .PC(PC)
);

// Initialize inputs
initial begin

    // Test Case 1: BGT with greater_than flag set
    OpCode = 6'b001000;
    Rd = 32'h0000000A;    // Example value for Rd
    Y = 32'h00000005;     // Example value for Y
    extended_immediate16 = 16'h0002;  // Example immediate value
    ALUOp = 2'b00;         // Addition operation
    #10;

    // Test Case 2: BLT with less_than flag set
    OpCode = 6'b001001;
    Rd = 32'hFFFFFFFF;    // Example value for Rd (negative)
    Y = 32'h0000000A;     // Example value for Y
    extended_immediate16 = 16'h0002;  // Example immediate value
    ALUOp = 2'b01;         // Subtraction operation
    #10;

    // Test Case 3: BEQ with zero flag set
    OpCode = 6'b001010;
```

```
Rd = 32'h00000000;    // Rd is set to zero
Y = 32'h00000000;     // Y is set to zero
extended_immediate16 = 16'h0003;  // Example immediate value
ALUOp = 2'b00;         // Addition operation
#10;

// Test Case 4: BNE with non-zero result
OpCode = 6'b001011;
Rd = 32'h00000001;     // Example value for Rd
Y = 32'h00000002;     // Example value for Y
extended_immediate16 = 16'h0003;  // Example immediate value
ALUOp = 2'b00;          // Addition operation
#10;

        // Test Case 1: BGT with greater_than flag set
    OpCode = 6'b001000;
    Rd = 32'h0000000A;    // Example value for Rd
    Y = 32'h00000005;     // Example value for Y
    extended_immediate16 = 16'h0002;  // Example immediate value
    ALUOp = 2'b00;         // Addition operation
    #10;

    // Test Case 2: BLT with less_than flag set
    OpCode = 6'b001001;
```

```
Rd = 32'hFFFFFFFF;    // Example value for Rd (negative)
Y = 32'h0000000A;     // Example value for Y
extended_immediate16 = 16'h0002;  // Example immediate value
ALUOp = 2'b01;          // Subtraction operation
#10;

// Test Case 3: BEQ with zero flag set
OpCode = 6'b001010;
Rd = 32'h00000000;    // Rd is set to zero
Y = 32'h00000000;     // Y is set to zero
extended_immediate16 = 16'h0003;  // Example immediate value
ALUOp = 2'b00;          // Addition operation
#10;

// Test Case 4: BNE with non-zero result
OpCode = 6'b001011;
Rd = 32'h00000001;     // Example value for Rd
Y = 32'h00000002;     // Example value for Y
extended_immediate16 = 16'h0003;  // Example immediate value
ALUOp = 2'b00;          // Addition operation
#10;
```

```
        // Finish simulation
        $finish;
    end

    // Display PC value during simulation
    always @* begin
        $display("PC = %h", PC);
    end
endmodule
```

All system code:

```
// Define global variables
reg [2:0]current_state;
reg [2:0]next_state;
reg [31:0]IR;


module allSystem(

        input clk,
        input reset,
```

```
// Observing the values of the generated signals
output reg PCSrc1,
output reg PCSrc0,
output reg RegWrite,
output reg ALUSrc0,
output reg [2:0] ALUOp,
output reg R_mem,
output reg W_mem,
output reg WB,
output reg push,
output reg pop,


///Instruction Decode Part
output reg [31:0] Rs1, // (Bus A )Output data from register 1
output reg [31:0] Rs2, // (Bus B) Output data from register 2
output reg [31:0] Rd, // (Bus W) Output data from register 3


///ALU Part
output reg [31:0] outputOfALU, // Just to verify that the ALU works properly

///immediate14 Part
output reg [31:0]extended_immediate16,
```

```
        ///immediate26 Part
        output reg [31:0]extended_immediate26,

        ///just for depugging the code for store
        output reg [31:0]outputOfTheStore,
);

        /// Instruction Fetch
        reg [31:0] memory [0:31];
        reg [31:0] PC;

        ///Instruction Decode
        reg [31:0] writeData;

    reg [31:0]Y;
        reg [4:0] Registers[0:31];


        reg [5:0]AddressOfRd;     // Address for Rd
        reg [5:0]AddressOfRs1;  // Address for RS1
        reg [5:0]AddressOfRs2;  // Address for RS2


        ///Control Unit Signals Generated
```

```
        reg [5:0]OpCode;

        ///ALU Part
         reg [31:0] operandA;    // Operand A
     reg [31:0] operandB;    // Operand B
         reg zero;
     reg greater_than;
     reg less_than;


        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        ///Data Memory
        reg [31:0] address,data_in;
    reg [31:0] data_out;

        ///Data Memory
//      reg [31:0] data_memory [31:0];
         reg [31:0] data_memory [31:0];
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        ///Write back stage
    reg [31:0]outOfDownMux;

        //Extenders 14 and 24
```

```
        reg [15:0] immediate16;  //16
        reg [25:0] immediate26;              // 26

        //Up adder
        reg [31:0] outputOfUpAdder;

        ///Stack Part
        reg [31:0] outputOfStack;
        reg empty,full;
        parameter STACK_DEPTH = 16;   // This is the maximum # of inner functions for
this stack
    reg [31:0] stack [STACK_DEPTH-1:0];
    reg [2:0] top;       //Stack Pointer

        //Branch Target Address
        reg [31:0] outputOfBTA;

        ///outputOfMuxAfterBTA
        reg [31:0]outputOfMuxAfterBTA;

        ///outputOfMuxAfterStack
        reg [31:0]outputOfMuxAfterStack;

        initial begin
```

```
$display("Test Case 1: Load-Store");
// Set memory values
memory[0] = 32'h10040034; // LW Rd = 9
memory[1] = 32'h18040004; // SW Rd
// Initialize registers
Registers[9] = 32'h00001234; // Set a value in register 9
// Apply clock cycles to execute instructions

// Verify the memory content and register value after execution
$display("Memory content at address 4: %h", data_memory[4]);
$display("Register 9 after load: %h", Registers[9]);


// Test case 2: Arithmetic Operations
$display("Test Case 2: Arithmetic Operations");
// Set memory values
memory[0] = 32'h08840000; // ADD Rd = Rd + 8 (Rd = 17)
memory[1] = 32'h10841000; // SUB Rd = Rd - 7 (Rd = 10)
// Initialize registers
Registers[15] = 32'h00000008; // Set initial value in register 17
// Apply clock cycles to execute instructions
```

```
// Verify the values in register 17 after arithmetic operations
$display("Register 17 after ADD operation: %h", Registers[15]);
$display("Register 17 after SUB operation: %h", Registers[15]);




  end

  initial top = 3'b000;


  integer i;
  always @(posedge clk)
        case (current_state)
              0:
                    next_state = 1;
              1:
                    next_state = 2;
              2:
                    next_state = 3;
              3:
                    next_state = 4;
              4:
```

```
                    next_state = 0;
        endcase


always @(posedge clk)
        current_state = next_state;


always @(posedge clk, posedge reset)

        if (reset)      begin

                current_state = 4;
                PC <= 32'h00000000;
                Rs1 <= 32'd0;
                Rs2 <= 32'd0;
                Rd <= 32'd0;
                Y <= 32'h00000000;


                for (i = 0; i < 31; i = i + 1)
                 Registers[i] <= 32'h00000000;
        end
```

```
else  if (current_state == 0) begin
        //instruction fetch and set signals:
                IR = memory[PC]; // Fetch instruction from memory using PC
                // Extract the opcode from the instruction
                OpCode = IR[31:26];

                        //$display ("%b", OpCode);



                        //pop signal
                        pop = IR[0];


                        /// R-type
                        if (OpCode == 6'b000000 || OpCode == 6'b000001 ||
OpCode == 6'b000010) begin ///ADD || AND || SUB

                                PCSrc1 = 1'b1;
                                PCSrc0 = 1'b0;
                              RegWrite = 1'b1;
                                ALUSrc0 = 1'b0;

                                if (OpCode == 6'b000000)
```

```
                    ALUOp = 2'b10;
                else if (OpCode == 6'b000001)
                    ALUOp = 2'b00;
                else
                    ALUOp = 2'b01;

            R_mem = 1'b0;
            W_mem = 1'b0;
            WB = 1'b1;
                push = 1'b0;
        end

        /// I-Type
        else if (OpCode == 6'b000011 || OpCode == 6'b000100)
begin    ///ANDI || ADDI

                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
            RegWrite = 1'b1;
                ALUSrc0 = 1'b1;

                if (OpCode == 6'b000011)   // ANDI
                    ALUOp = 2'b10;
                else
```

```
                ALUOp = 2'b00;


        R_mem = 1'b0;
        W_mem = 1'b0;
        WB = 1'b1;
                push = 1'b0;
end

else if (OpCode == 6'b000101) begin //LW

                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
            RegWrite = 1'b1;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b00;
        R_mem = 1'b1;
        W_mem = 1'b0;
                push = 1'b0;
                WB = 1'b0;
                push = 1'b0;
end

else if (OpCode == 6'b000111) begin //SW
```

```
                PCSrc1 = 1'b1;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b00;
        R_mem = 1'b0;
        W_mem = 1'b1;
                push = 1'b0;
    end

    else if (OpCode == 6'b001010) begin //BEQ

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b01;
        R_mem = 1'b0;
        W_mem = 1'b0;
                push = 1'b0;
    end

    else if (OpCode == 6'b001000) begin //BGT
```

```
                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b01;    /// We make new ALU op
for Greater than
            R_mem = 1'b0;
            W_mem = 1'b0;
                push = 1'b0;

            end

            else if (OpCode == 6'b001001) begin //BLT

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b01;          // We make new
ALU for less than
            R_mem = 1'b0;
            W_mem = 1'b0;
                push = 1'b0;
```

```
end

else if (OpCode == 6'b001011) begin //BNE

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b0;
            RegWrite = 1'b0;
                ALUSrc0 = 1'b1;
                ALUOp = 2'b01;
        R_mem = 1'b0;
        W_mem = 1'b0;
                push = 1'b0;


end



/// J-Type
else if (OpCode == 6'b001100) begin   // J

                PCSrc1 = 1'b0;
                PCSrc0 = 1'b1;
```

```
            RegWrite = 1'b0;
        R_mem = 1'b0;
            W_mem = 1'b0;
                push = 1'b0;
              end

        else if(OpCode == 6'b001101) begin    // Call

                        PCSrc1 = 1'b0;
                        PCSrc0 = 1'b1;
                    RegWrite = 1'b0;
                R_mem = 1'b0;
                W_mem = 1'b0;
                        push = 1'b1;
              end
        else if (OpCode == 6'b001110) begin   // RET

                        PCSrc1 = 1'b0;
                        PCSrc0 = 1'b1;
                    RegWrite = 1'b0;
                R_mem = 1'b0;
                W_mem = 1'b0;
```

```
                end

            else if (OpCode == 6'b001111) begin/// S-Type for stack
push

                    push = 1'b1;

            end


            else if (OpCode == 6'b010000) begin          //pop

                    pop = 1'b1;
                    RegWrite = 1'b1;
            end


        end

     //Next extend data if needed
    else if (current_state == 1) begin
```

```
            immediate16 = IR[17:2];
            extended_immediate16 = {16'b0000000000000000, immediate16};
            immediate26 = IR[25:0];
            extended_immediate26 = {6'b000000, immediate26};

    // Assign addresses for registers
                AddressOfRd =  IR[25:22];
            AddressOfRs1 =  IR[21:18];
            AddressOfRs2 = IR[17:14];



                Registers[0] = 32'h00000008;                    //0001 1000 0000
0010 1000 0000 0000 0110
                Registers[1] = 32'h00000007;
                Registers[8] = 32'h00000002;



                //$display (PC);
            //$display ("%b", IR);
            //$display ("\n");

                Rd = Registers[AddressOfRd];
                Rs1 = Registers[AddressOfRs1];
```

```
        Rs2 = Registers[AddressOfRs2];



        case (ALUSrc0)
                1'b1:Y = immediate16;
                1'b0:Y = Rs2;
        endcase
        case (IR[31:26])
6'b001100: begin // Jump (J)
    current_state = 4;
    outputOfUpAdder = PC + extended_immediate26;
    PC = outputOfUpAdder;
end
6'b001101: begin // Call
    current_state = 4;
    empty = (top == 0);
    full = (top == 15);

    // Push operation
    if (push && !pop && !full) begin
        stack[top] = PC;
        top = top + 1;
    end
```

```
    PC = PC + extended_immediate26;
end
6'b001110: begin // Return
    current_state = 4;
    PC = stack[top];
end
6'b001111: begin // Push Rd
    current_state = 4;
    empty = (top == 0);
    full = (top == 15);

    // Push Rd
    if (push && !pop && !full) begin
        stack[top] = Rd;
        top = top + 1;
    end
end
6'b010000: begin // Pop Rd
    current_state = 4;
    Rd = stack[top]; // Store the top of the stack in Rd
    top = top - 1;
end
default: begin
    // Handle other instructions if needed
```

```
 end
endcase



 end

else if (current_state == 2)  begin

        operandA = Rd;
        operandB = Y;

       // Perform ALU operation based on ALUOp

          case (ALUOp)
             2'b00: outputOfALU = operandA + operandB;   // Addition
             2'b01: outputOfALU = operandA - operandB;   // Subtraction
             2'b10: outputOfALU = operandA & operandB;   // Bitwise AND
             default: outputOfALU = 32'b0;              // Default case: result is
0
          endcase


       // Set flags
```

```
                zero = (outputOfALU == 32'b0);              // Set zero flag if the
result is zero
                greater_than = (outputOfALU > 32'h0);       // Set greater than
flag
                less_than = (outputOfALU < 32'h0);          // Set less than flag


            // Check instructions

                outputOfBTA = 32'b0;                        // Initialize outputOfBTA

                // Check BGT instruction
                if (OpCode == 6'b001000) begin
                    if (greater_than) begin
                        current_state = 4;
                        outputOfBTA <= PC + extended_immediate16;
                        PC <= outputOfBTA;
                    end else begin
                        PC <= PC + 1;
                        current_state = 4;
                    end


                // Check BLT instruction
```

```
if (OpCode == 6'b001001) begin
    if (less_than) begin
        current_state = 4;
        outputOfBTA <= PC + extended_immediate16;
        PC <= outputOfBTA;
    end else begin
        PC <= PC + 1;
        current_state = 4;
    end
end

// Check BEQ instruction
if (OpCode == 6'b001010) begin
    if (zero) begin
        current_state = 4;
        outputOfBTA <= PC + extended_immediate16;
        PC <= outputOfBTA;
    end else begin
        PC <= PC + 1;
        current_state = 4;
    end
end

// Check BNE instruction
```

```
            if (OpCode == 6'b001011) begin
                if (!zero) begin
                    current_state = 4;
                    outputOfBTA <= PC + extended_immediate16;
                    PC <= outputOfBTA;
                end else begin
                    PC <= PC + 1;
                    current_state = 4;
                end
            end
        end

end

else if (current_state == 3) begin


    address     = outputOfALU; /// Address of the memory
    data_in     = Rd; // Store Instruction

    if(W_mem && !R_mem) begin

        /// Store Instruction
        data_memory[address] = data_in;
```

```
                    outputOfTheStore = data_memory[address];
                    //$display ("data_memory[%d] = %d", address,
outputOfTheStore);

                end

                else if(!W_mem && R_mem) begin

                    /// Load Instruction
                    data_memory[14] = 32'h00000009;
                    data_out = data_memory[address];

                end

                ///Check the current instruction is Store or not
                if (OpCode == 6'b000111) begin

                    current_state = 4;

                    ///check if the current instruction is the last instruction in the
function

                        ///Here is the code to choose the correct value for the PC
```

```
                    ///Stack Part
                     empty = (top == 0);
                     full = (top == STACK_DEPTH);

                     // Pop operation
                     if (pop && !push && !empty) begin

                    top = top - 1;  // Move the top pointer down
                            outputOfStack = stack[top];

                               PC = outputOfStack;
                           PC = PC + 1;
                        end

                         else PC = PC + 1;

                end
            end


else if (current_state == 4) begin

        //$display ("Data out = %d", data_out);
```

```
 if (WB == 1)
        outOfDownMux = outputOfALU;
 else
        outOfDownMux = data_out;

 /// Õrite back to the destination Register
if (RegWrite) begin
        Rd = outOfDownMux;
        Registers[AddressOfRd] = Rd;
end

///Here is the code to choose the correct value for the PC
//$display(PC);
///Stack Part
 empty = (top == 0);
 full = (top == STACK_DEPTH);

 // Pop operation
 if (pop && !push && !empty) begin

top = top - 1;  // Move the top pointer down
         outputOfStack = stack[top];

        PC = outputOfStack;
```

```
                PC = PC + 1;
            end

                else PC = PC + 1;
                end



endmodule



`timescale 1ps / 1ps
module allSystem_tb;
    // Inputs
    reg clk;
    reg reset;

    // Outputs
    wire PCSrc1;
    wire PCSrc0;
    wire RegWrite;
```

```
wire ALUSrc0;
wire [2:0] ALUOp;
wire R_mem;
wire W_mem;
wire WB;
wire push;
wire pop;
wire [31:0] Rs1;
wire [31:0] Rs2;
wire [31:0] Rd;
wire [31:0] outputOfALU;
wire [31:0] extended_immediate16;
wire [31:0] extended_immediate26;
wire [31:0] outputOfTheStore;

// Instantiate the unit under test (UUT)
allSystem uut (
    .clk(clk),
    .reset(reset),
    .PCSrc1(PCSrc1),
    .PCSrc0(PCSrc0),
    .RegWrite(RegWrite),
    .ALUSrc0(ALUSrc0),
    .ALUOp(ALUOp),
```

```
        .R_mem(R_mem),
        .W_mem(W_mem),
        .WB(WB),
        .push(push),
        .pop(pop),
        .Rs1(Rs1),
        .Rs2(Rs2),
        .Rd(Rd),
        .outputOfALU(outputOfALU),
        .extended_immediate16(extended_immediate16),
        .extended_immediate26(extended_immediate26),
        .outputOfTheStore(outputOfTheStore)
    );

initial begin current_state = 0; clk = 0; reset = 1; #1ns reset = 0; end

//always @ (posedge clk) $display (current_state);

always #5ns clk = ~clk;


initial
        begin
        #1000;
```

```
        $monitor($realtime,,"ps %h %h %h %h %h %h %h %h %h %h %h %h %h %h %h
%h %h %h %h
",clk,reset,PCSrc1,PCSrc0,RegWrite,ALUSrc0,ALUOp,R_mem,W_mem,WB,push,pop,Rs
1,Rs2,Rd,outputOfALU,extended_immediate16,extended_immediate26,outputOfTheStore)
;

        end


initial #200ns $finish;


endmodule
```

References

Slides.