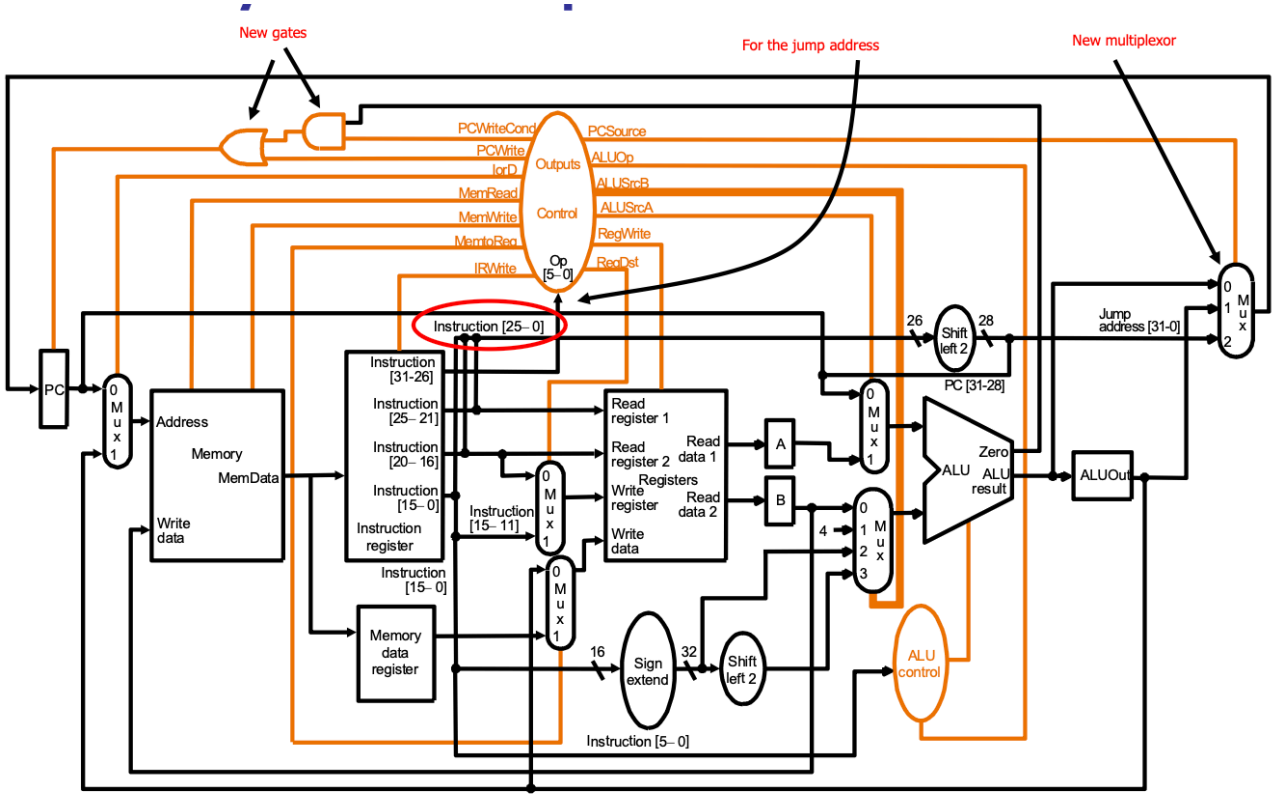
COMPUTER ARCHITECTURE PROJECT

Group 37

Aditya Balasubramanian - 2021A8PS1475H
Priyansh Ahuja - 2021A3PS2975H
Viswajith Sunkerla - 2021A3PS2581H



The microprocessor has been designed with this computer archotectural diagram in mind.

The program counter contains the following instructions at the appropriate addresses. The data registers that hold values are also initialised as

0000 BEQ reg1, reg2, 7

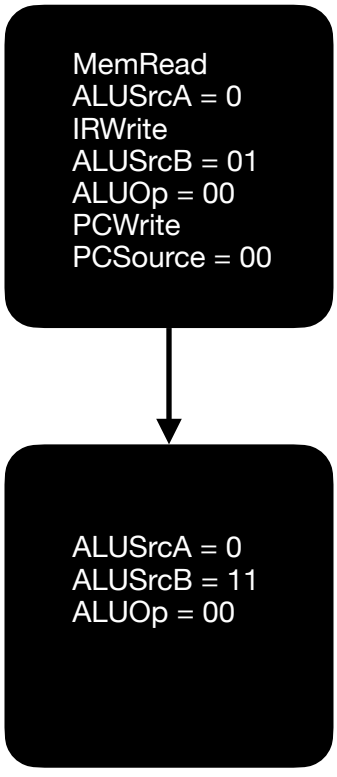
0004 ORI reg5, reg4, 10

reg1 = 26677D

reg2 = 2667DA
reg4 = 55F3

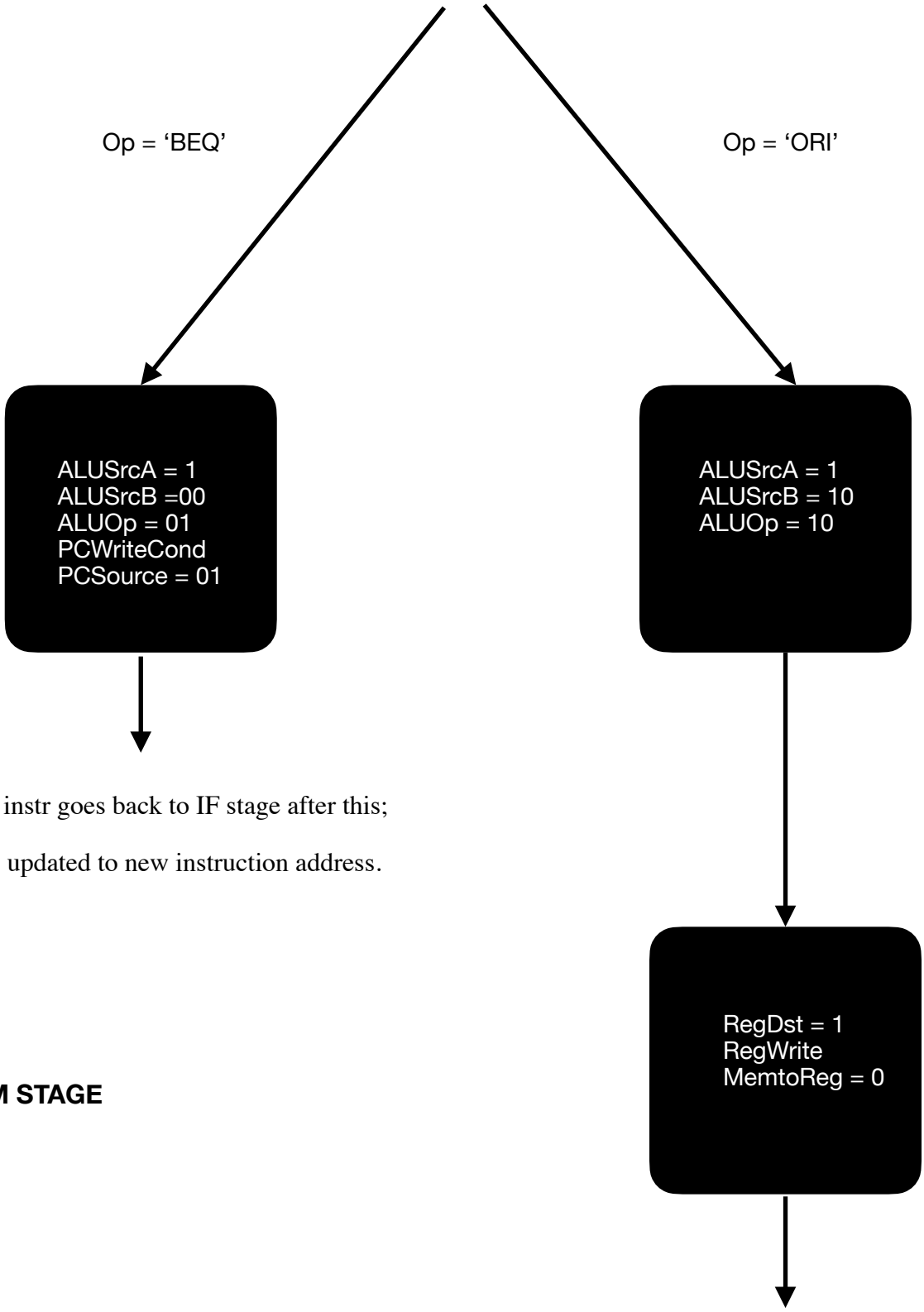
Finite State Machine for the given instructions:

INSTRUCTION FETCH



INSTRUCTION DECODE

EXECUTION STAGE



BEQ instr goes back to IF stage after this;

PC is updated to new instruction address.

MEM STAGE

ORI instr goes back to IF stage after this;

WB STAGE : There is NO need for a WB Stage for these instructions.

VERILOG CODE :

Top Module:

```
module MIPS(
    clock,
    rst,
    state
    );

    input clock;
    input rst;
    output [3:0] state;
    wire alu_Zero;
    wire sig_Branch;
    wire sig_PCWrite;
    wire sig_MemWrite;
    wire sig_IRWrite;
    wire sig_RegDst;
    wire sig_ALUSrcA;
    wire sig_MemtoReg;
    wire [2:0] sig_ALUControl;
    wire [1:0] sig_ALUSrcB;
    wire sig_IorD;
    wire _Zero;
    wire _Pc_En;
```

```
wire [31:0] _Pc;
wire [31:0] _Pc_Prime;
wire [31:0] _Adr;
wire [31:0] _Rd;
wire [31:0] _Instr;
wire [31:0] _Data;
wire [4:0] _A3;
wire [31:0] _Wd3;
wire [31:0] _Rd1;
wire [31:0] _Rd2;
wire [31:0] _Sign_Imm;
wire [31:0] _Reg_A;
wire [31:0] _Reg_B;
wire [31:0] _Src_A;
wire [31:0] _Src_B;
wire [31:0] _Alu_Result;
wire _Pc_Src;
wire [31:0] _Alu_Out;
wire [31:0] _Sign_Imm_Shifted;

Control_Unit control (
.clock(clock),
.rst(rst),
.instr_Opcode(_Instr[31:26]),
.sig_MemtoReg(sig_MemtoReg),
.sig_RegDst(sig_RegDst),
.sig_IorD(sig_IorD),
.sig_PCSrc(_Pc_Src),
.sig_ALUSrcB(sig_ALUSrcB),
.sig_ALUSrcA(sig_ALUSrcA),
```

```
.sig_IRWrite(sig_IRWrite),
.sig_MemWrite(sig_MemWrite),
.sig_PCWrite(sig_PCWrite),
.sig_Branch(sig_Branch),
.sig_RegWrite(RegWrite),
.state(state),
.alu_Control(sig_ALUControl),
.instr_Function(_Instr[5:0])
);


Register_En pc (
.clock(clock),.rst(rst),
.enable(_Pc_En),
.in(_Pc_Prime),
.out(_Pc)
);


Mux_2_Bit #(.DATA_WIDTH(32)) mux_i_or_d(
.input_0(_Pc),
.input_1(_Alu_Out),
.selector(sig_IorD),
.mux_Out(_Adr)
);


Memory #(.DATA_WIDTH(50))memory (
.clock(clock),
.sig_MemWrite(sig_MemWrite),
```

```
.adr(_Adr),
.wd(_Reg_B),
.rd(_Rd)
);

Register_En instr (
.clock(clock),
.enable(sig_IRWrite),
.in(_Rd),
.out(_Instr)
);

Register data (
.clock(clock),
.in(_Rd),
.out(_Data)
);

Mux_2_Bit #(.DATA_WIDTH(5)) mux_reg_dst(
.input_0(_Instr[20:16]),
.input_1(_Instr[15:11]),
.selector(sig_RegDst),
.mux_Out(_A3)
);
```

```
Mux_2_Bit #(.DATA_WIDTH(32)) mux_mem_to_reg (
.input_0(_Alu_Out),
.input_1(_Data),
.selector(sig_MemtoReg),
.mux_Out(_Wd3)
);


Register_File register_file (
.clock(clock),
.a1(_Instr[25:21]),
.a2(_Instr[20:16]),
.a3(_A3),
.wd3(_Wd3),
.sig_RegWrite(RegWrite),
.rd1(_Rd1),
.rd2(_Rd2)
);


Sign_Extention sign_extention (
.immidiate(_Instr[15:0]),
.sign_Imm(_Sign_Imm)
);


Register reg_a (
.clock(clock),
.in(_Rd1),
.out(_Reg_A)
```

```
);


Register reg_b (
.clock(clock),
.in(_Rd2),
.out(_Reg_B)
);


Mux_2_Bit #(.DATA_WIDTH(32)) mux_alu_src_a(
.input_0(_Pc),
.input_1(_Reg_A),
.selector(sig_ALUSrcA),
.mux_Out(_Src_A)
);


Mux_4_Bit #(.DATA_WIDTH(32)) mux_alu_src_b (
.input_0(_Reg_B),
.input_1(4), //4should be here
.input_2(_Sign_Imm),
.input_3(_Sign_Imm_Shifted),
.selector(sig_ALUSrcB),
.mux_Out(_Src_B)
);


Shifter1 #(.DATA_WIDTH(32)) shifter_1 (
.sign_Imm(_Sign_Imm),
```

```
.shifted_Sign_Imm(_Sign_Imm_Shifted)
);


Alu alu (
.clock(clock),
.alu_Control(sig_ALUControl),
.src_A(_Src_A),
.src_B(_Src_B),
.alu_Result(_Alu_Result),
.alu_Zero(_Zero)
);


Control_Branch control_branch(
.sig_Branch(sig_Branch),
.alu_Zero(_Zero),
.sig_PCWrite(sig_PCWrite),
.pc_En(_Pc_En)
);


Register alu_out (
.clock(clock),
.in(_Alu_Result),
.out(_Alu_Out)
);


Mux_2_Bit #(.DATA_WIDTH(32)) mux_pc_src (
```

```
    .input_0(_Alu_Result),
    .input_1(_Alu_Out),
    .selector(_Pc_Src),
    .mux_Out(_Pc_Prime)
    );
endmodule
```

Sub Modules:

RegisterEn

```
module Register_En(
   input clock,
    input rst,
    input [31:0] in,
    input enable,
    output reg [31:0] out
    );

    always@(posedge clock or posedge rst) begin
        if(rst) out<=0;
        else begin
            if(enable) out<=in;
        end
    end
endmodule
```

Memory

```
module Memory(
    clock,
    sig_MemWrite,
```

```
    adr,
    wd,
    rd
    );

    input clock;
    input sig_MemWrite;
    input [31:0] adr;
    input [31:0] wd ;
    output reg [31:0] rd;

    parameter DATA_WIDTH = 50;
    reg [7:0] ram [0:DATA_WIDTH-1];

    initial begin
        {ram[0],ram[1],ram[2],ram[3]} =
32'b000100_00001_00010_0000000000000111;  //BEQ instruction
        {ram[4],ram[5],ram[6],ram[7]} =
32'b001000_00100_00101_0000000000001010; //ORi instruction
        {ram[8],ram[9],ram[10],ram[11]} =
32'b000000_00011_00010_00001_00000_101010;  //R type instruction
(slt)
        //{ram[7],ram[6],ram[5],ram[4]}= 32'b_;
    end
    always @(posedge clock) begin
        if(sig_MemWrite == 0)
            rd = {ram[adr],ram[adr+1],ram[adr+2],ram[adr+3]};
        else
```

```
        ram[adr] = wd;
    end

endmodule
```

Control Unit

```
module Control_Unit(
    clock,
    rst,
    instr_Opcode,
    instr_Function,
    sig_MemtoReg,
    sig_RegDst,
    sig_IorD,
    sig_PCSrc,
    sig_ALUSrcB,
    sig_ALUSrcA,
    sig_IRWrite,
    sig_MemWrite,
    sig_PCWrite,
    sig_Branch,
    sig_RegWrite,
    state,
    alu_Control
    );

    input clock;
```

```
input rst;
input [5:0] instr_Opcode;
input [5:0] instr_Function;
output sig_MemtoReg;
output sig_RegDst;
output sig_IorD;
output sig_PCSrc;
output [1:0] sig_ALUSrcB;
output sig_ALUSrcA;
output sig_IRWrite;
output sig_MemWrite;
output sig_PCWrite;
output sig_Branch;
output sig_RegWrite;

output reg [3:0] state;
output reg [2:0] alu_Control;

parameter STATE_0 = 0;
parameter STATE_1 = 1;
parameter STATE_2 = 2;
parameter STATE_3 = 3;
parameter STATE_4 = 4;
parameter STATE_5 = 5;
parameter STATE_6 = 6;
parameter STATE_7 = 7;
```

```
parameter STATE_8 = 8;
parameter STATE_9 = 9;
parameter STATE_10 = 10;
parameter STATE_11 = 11;
parameter STATE_12 = 12;
parameter STATE_13 = 13;
parameter STATE_14 = 14;

parameter R_TYPE = 6'b000000; //R_Type
parameter BEQ = 6'b000100;    //Beq
parameter ORI = 6'b001000;   //ORi

wire [1:0] alu_Op;
always@(posedge clock) begin
   if (rst)
      state <= 4'd0;
   else begin
      case(state)
         STATE_0:
            state <= 4'd1;
         STATE_1:begin
            if (instr_Opcode == R_TYPE)
               state <= 4'd6;
            else if (instr_Opcode == ORI)
               state <= 4'd9;
            else if (instr_Opcode == BEQ)
```

```
                state <= 4'd8;
            else
                state <= 4'd0;
        end
        STATE_6: begin
                state <= 4'd7;
        end
        STATE_7:
            state <= 4'd0;
        STATE_8:
            state <= 4'd0;
        STATE_9:
            state <= 4'd10;
        STATE_10:
            state <= 4'd0;
        default : state <= 4'd0;
      endcase
    end
  end

  assign sig_IorD = 1'b0;

  assign sig_ALUSrcA = (rst == 1'b1) ? 1'b0  :(((state == 4'd9)|| (state ==
4'd8)|| (state == 4'd6)) ? 1'b1 : 1'b0);

  assign sig_ALUSrcB = (rst == 1'b1) ? 2'b00 :((state == 4'd0)? 2'b01 :
((state == 4'd1) ? 2'b11 : (((state == 4'd9)) ? 2'b10 : 2'b00)));
```

```
    assign alu_Op = (rst == 1'b1) ? 2'b0  :((state == 4'd8)? 2'b01: ((state ==
4'd6) ? 2'b10 : ((state == 4'd9)? 2'b11: 2'b00)));


    assign sig_PCSrc = (rst == 1'b1) ? 1'b0 :((state == 4'd8) ? 1'b1 : 1'b0);


    assign sig_IRWrite = (rst == 1'b1) ? 1'b0  :((state  == 4'd0) ? 1'b1 :
1'b0);


    assign sig_PCWrite = (rst == 1'b1) ? 1'b0  :((state == 4'd0) ? 1'b1 :
1'b0);


    assign sig_Branch = (rst == 1'b1) ? 1'b0  :((state == 4'd8) ? 1'b1 : 1'b0);


    assign sig_RegDst = (rst == 1'b1) ? 1'b0  :((state == 4'd7)? 1'b1 : 1'b0);


    assign sig_MemtoReg = (rst == 1'b1) ? 1'b0 : 1'b0;


    assign sig_RegWrite = (rst == 1'b1) ? 1'b0  :(((state == 4'd7) || (state ==
4'd10)) ? 1'b1 : 1'b0);


    assign sig_MemWrite = 1'b0;




    always @(alu_Op or instr_Function) begin
        if (alu_Op ==0)
            alu_Control <= 3'b010;              // add
        else if (alu_Op == 2'b01 )
```

```
        alu_Control <= 3'b110;              // sub

    else if (alu_Op == 2'b10 ) begin
        case (instr_Function)
            6'b100000 :
                alu_Control <= 3'b010;  // add
            6'b100010 :
                alu_Control <= 3'b110;  // sub
            6'b100100 :
                alu_Control <= 3'b000;  // and
            6'b100101 :
                alu_Control <= 3'b001;  // or
            6'b101010 :
                alu_Control <= 3'b111;  // slt
            6'b100110 :
                alu_Control <= 3'b101;  // Xor
            default :
                alu_Control <= 3'bxxx;
        endcase
    end
    else if (alu_Op == 2'b11 ) begin
        alu_Control <= 3'b001;
        end
    else
        alu_Control <= 3'bxxx;
end
```

```
endmodule
```

Register

```
module Register(
    clock,
    in,
    out
    );

    input clock;
    input [31:0] in;
    output  [31:0]out;

    reg [31:0] out ;
    always @ (posedge clock) begin
        out <= in;
    end

endmodule
```

ALU

```
module Alu(
    clock,
    alu_Control,
    src_A,
    src_B,
    alu_Result,
```

```
alu_Zero
);

input clock;
input [2:0] alu_Control;
input [31:0] src_A;
input [31:0] src_B;
output reg[31:0] alu_Result;
output alu_Zero;

always  @(*) begin
    case(alu_Control)
        3'b010: begin
            alu_Result =  src_A + src_B;
        end
        3'b110: begin
            alu_Result  =  src_A - src_B;
        end
        3'b000:
            alu_Result  =  src_A & src_B;
        3'b001:
            alu_Result  =  src_A | src_B;
        3'b101:
            alu_Result  =  src_A ^ src_B;
        3'b111: begin
            if  (src_A < src_B)
```

```
                alu_Result =1;
            else
                alu_Result =0;
        end
        default:
            alu_Result = 32'hxxxxxxxx;
    endcase
end

assign alu_Zero = (alu_Result == 32'd0) ? 1'd1 : 1'd0;

endmodule
```

Register File

```
module Register_File(
    clock,
    a1,
    a2,
    a3,
    wd3,
    sig_RegWrite,
    rd1,
    rd2
    );

    input clock;
    input [4:0] a1;
```

```
input [4:0] a2;
input [4:0] a3;
input [31:0] wd3;
input sig_RegWrite;
output  [31:0] rd1;
output  [31:0] rd2;

reg [31:0] reg_File  [0:31];
integer i;
initial begin
        reg_File[0] = 32'h0;
        reg_File[1] = 32'h26677D;
        reg_File[2] = 32'h2667da;
        reg_File[3] = 32'h0;
        reg_File[4] = 32'h55F3;
        for(i=5; i<32 ;i=i+1)begin
            reg_File[i]=32'd0;
        end
end

assign rd1 = reg_File[a1];
assign rd2 = reg_File[a2];

always @(posedge clock) begin
    if (sig_RegWrite == 1)
        reg_File[a3] <= wd3;
```

```
    end

endmodule
```

Sign Extension

```
module Sign_Extention(
    immidiate,
    sign_Imm
    );

    input [15:0] immidiate;
    output reg [31:0] sign_Imm;

    always @(*) begin
        sign_Imm[31:0] <= { {16{immidiate[15]}}, immidiate[15:0]};
    end

endmodule
```

mux2

```
module Mux_2_Bit(
    input_0,
    input_1,
    selector,
    mux_Out
    );

    parameter DATA_WIDTH = 32;
```

```
    input [DATA_WIDTH -1:0] input_0;
    input [DATA_WIDTH -1:0] input_1;
    input selector;
    output reg [DATA_WIDTH -1:0] mux_Out;

    always @ (*) begin
      case (selector)
        1'b0:
          mux_Out = input_0;
        1'b1:
          mux_Out = input_1;
        default:
          mux_Out = 32'bx;
      endcase
    end

endmodule
```

Mux4

```
module Mux_4_Bit(
  input_0,
  input_1,
  input_2,
  input_3,
  selector,
  mux_Out
  );
```

```
  parameter DATA_WIDTH = 32;
  input [DATA_WIDTH -1:0] input_0;
  input [DATA_WIDTH -1:0] input_1;
  input [DATA_WIDTH -1:0] input_2;
  input [DATA_WIDTH -1:0] input_3;
  input [1:0]selector;
  output reg [DATA_WIDTH -1:0] mux_Out;

  always @ (*) begin
    case (selector)
      2'b00 :
        mux_Out = input_0;
      2'b01 :
        mux_Out = input_1;
      2'b10 :
        mux_Out = input_2;
      2'b11 :
        mux_Out = input_3;
      default :
        mux_Out = 32'bx;
    endcase
  end

endmodule
```

Shifter

```
module Shifter1(
    sign_Imm,
    shifted_Sign_Imm
    );

    parameter DATA_WIDTH = 32;
    input [DATA_WIDTH - 1:0] sign_Imm;
    output [DATA_WIDTH -1:0] shifted_Sign_Imm;

    assign shifted_Sign_Imm = sign_Imm << 2;

endmodule
```

Control Branch

```
module Control_Branch(
    sig_Branch,
    alu_Zero,
    sig_PCWrite,
    pc_En
    );

    input sig_Branch;
    input alu_Zero;
    input sig_PCWrite;
    output pc_En;

    wire out1,out2;
```

```
    assign out1 = sig_Branch & alu_Zero;
    assign pc_En = out1 | sig_PCWrite;

endmodule
```

Test Bench:

```
module MIPS_tb;

    // Inputs
    reg clock;
    reg rst;

    // Outputs
    wire [3:0] state;

    // Instantiate the Unit Under Test (UUT)
    MIPS uut (
        .clock(clock),
        .rst(rst),
        .state(state)
    );
    always #5 clock =~clock;
    initial begin
        // Initialize Inputs
```

```
        clock = 0;
        rst = 1;#1;
        rst = 0;#200 $finish;


    end
endmodule
```

Explanation of the code for the given instructions:

Sub Modules:

RegisterEn

This Verilog module implements a register with an enable signal. It synchronously updates the output register out with the input in on the

rising edge of the clock if the enable signal is asserted, otherwise maintains the current value.

Memory

This Verilog module represents a memory unit with read and write functionality. It uses a block RAM implementation to store data, accessing and updating memory based on the clock signal and control signals.

Control Unit

This Verilog module acts as the control unit for a multi-cycle microprocessor. It determines control signals based on the current state and instruction opcode, facilitating the execution of instructions by enabling various functional units such as the ALU, memory unit, and register file. The module also includes logic for state transitions and ALU operation selection based on the instruction function code.

Register

This Verilog module implements a synchronous register that stores the input value on the rising edge of the clock signal, forwarding it to the output.

ALU

This Verilog module represents an Arithmetic Logic Unit (ALU) in a multi-cycle microprocessor. It performs arithmetic and logical operations based on the control signals, producing the result on the output. Additionally, it generates a flag indicating whether the result is zero or not.

Register File

This Verilog module implements a register file in a multi-cycle microprocessor. It contains 32 registers and supports read and write operations based on control signals and input addresses, allowing for data storage and retrieval during instruction execution.

Sign Extension

This Verilog module performs sign extension on a 16-bit immediate value to 32 bits. It replicates the most significant bit of the input to fill the upper 16 bits of the output, effectively extending the signed value to a wider size.

The MUX modules represent the multiplexers whose o/p is dependent on their respective control signals.

Top/Main Module:

The "MIPS" module represents the top-level design of a MIPS microprocessor implemented using various submodules such as registers, ALU, memory, control unit, multiplexers, and signal extenders.

This simply involves instantiation of all the sub-modules and running og the required instructions.

- Control Unit: It generates control signals based on the current state and instruction opcode, facilitating the execution of instructions by enabling various functional units.
- Register File: It stores and retrieves data from 32 registers based on control signals and input addresses.
- Memory: It serves as the data memory unit, allowing read and write operations based on the address and control signals.
- ALU (Arithmetic Logic Unit): It performs arithmetic and logical operations based on the control signals and input operands.
- Sign Extension: It extends a 16-bit immediate value to 32 bits by replicating the most significant bit.
- Multiplexers: They select one of the two input signals based on the value of the selector input.
- Registers: They store data and forward it to the output on the rising edge of the clock signal.
- Shifter: It performs shifting operations on input data.
- Control Branch: It controls the branching mechanism based on the branch condition and ALU result.

- Top-level connections: It interconnects all submodules, facilitating data flow and control signal propagation throughout the microprocessor.