Computer Architecture Lab Project

RISCV(Pipeline) Implementation

Group members:

Kuldeep Dileep Lohana (kl04008)

Swaleha Muhammas Saleem (sm5152)

Instructor: Dr. Ayaz ul Hassan Khan

Objective: In this lab project we modified our single cycle RISCV processor implemented in lab 11 to pipeline architecture along with bubble sort as shown in the figure below.

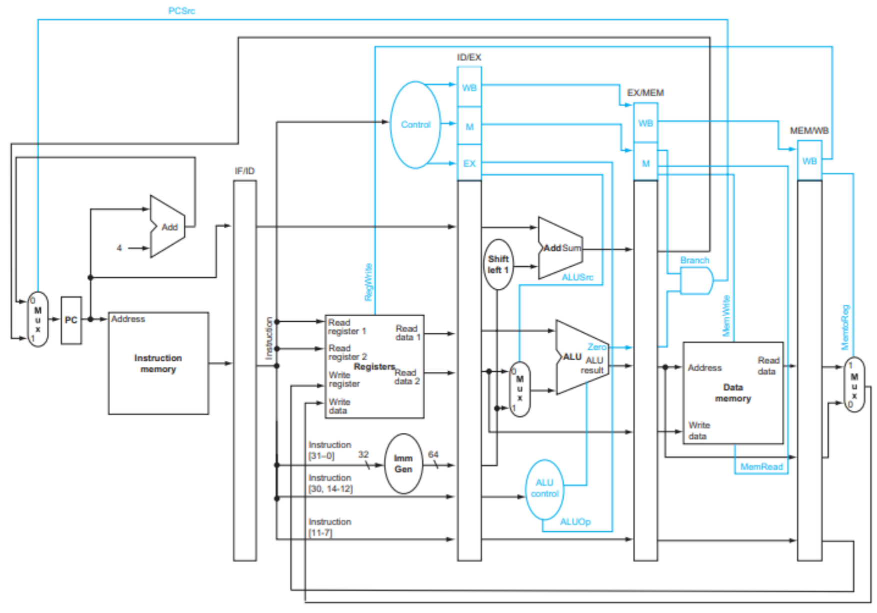
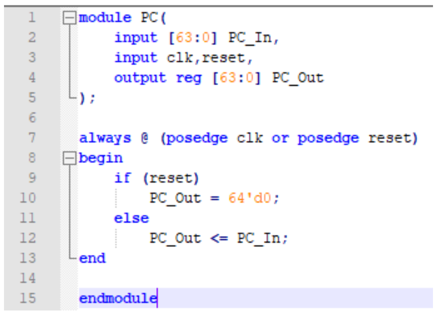


FIGURE 4.49 The pipelined datapath of Figure 4.44, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Note: Above figure misses instruction parser module but we have implemented that in our project.

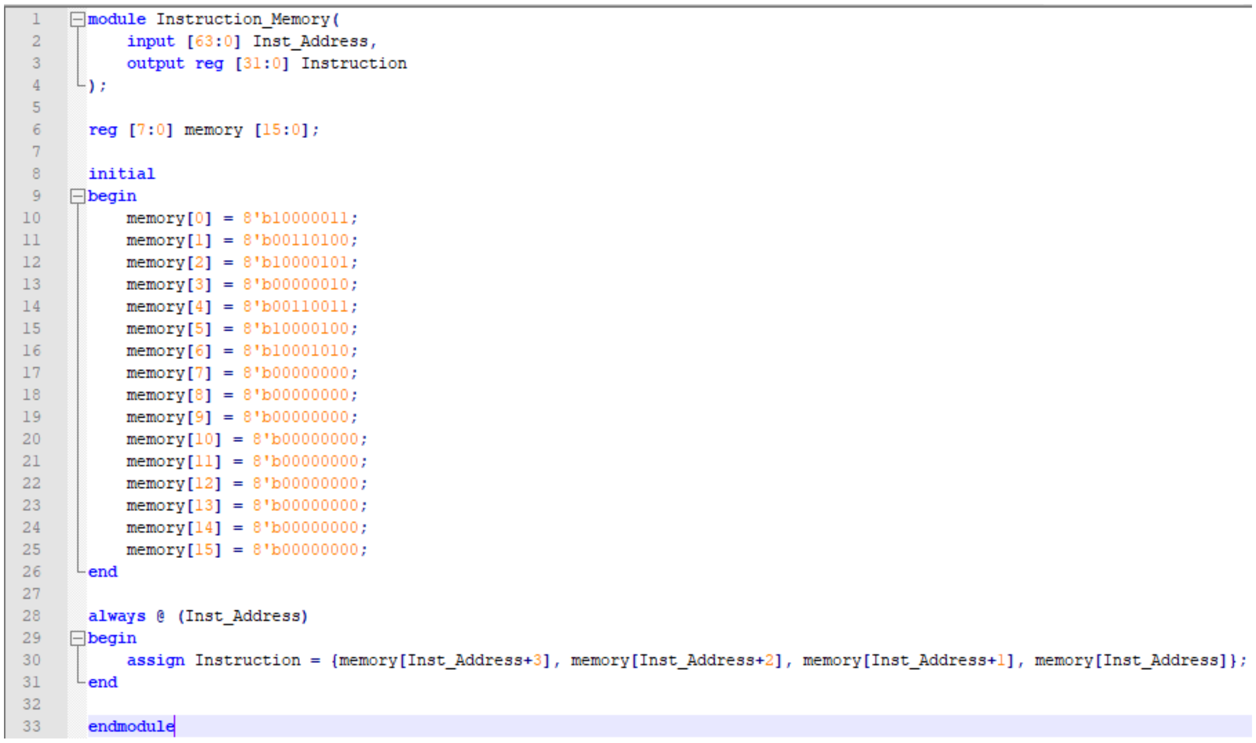
Following are the snips of our codes of all modules with their explanation:

Program counter (PC):



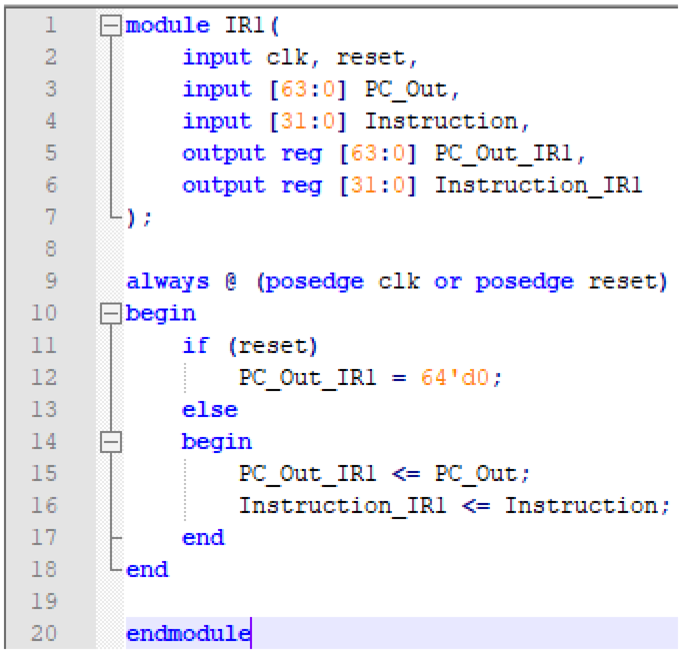
```
module PC(
    input [63:0] PC_In,
    input clk,reset,
    output reg [63:0] PC_Out
);

always @ (posedge clk or posedge reset)
begin
    if (reset)
        PC_Out = 64'd0;
    else
        PC_Out <= PC_In;
end

endmodule
```

Instruction Memory (IM):



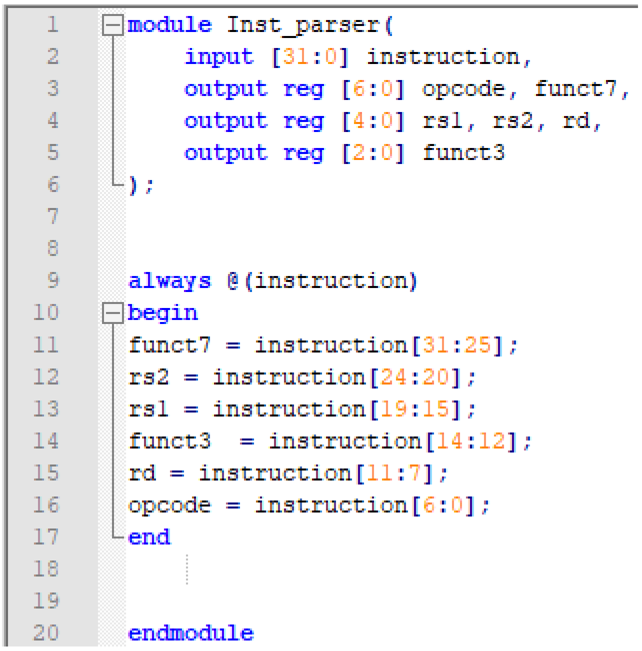
```verilog
module Instruction_Memory(
    input [63:0] Inst_Address,
    output reg [31:0] Instruction
);

reg [7:0] memory [15:0];

initial
begin
    memory[0] = 8'b10000011;
    memory[1] = 8'b00110100;
    memory[2] = 8'b10000101;
    memory[3] = 8'b00000010;
    memory[4] = 8'b00110011;
    memory[5] = 8'b10000100;
    memory[6] = 8'b10001010;
    memory[7] = 8'b00000000;
    memory[8] = 8'b00000000;
    memory[9] = 8'b00000000;
    memory[10] = 8'b00000000;
    memory[11] = 8'b00000000;
    memory[12] = 8'b00000000;
    memory[13] = 8'b00000000;
    memory[14] = 8'b00000000;
    memory[15] = 8'b00000000;
end

always @ (Inst_Address)
begin
    assign Instruction = {memory[Inst_Address+3], memory[Inst_Address+2], memory[Inst_Address+1], memory[Inst_Address]};
end

endmodule
```

IF/ID Pipeline register:



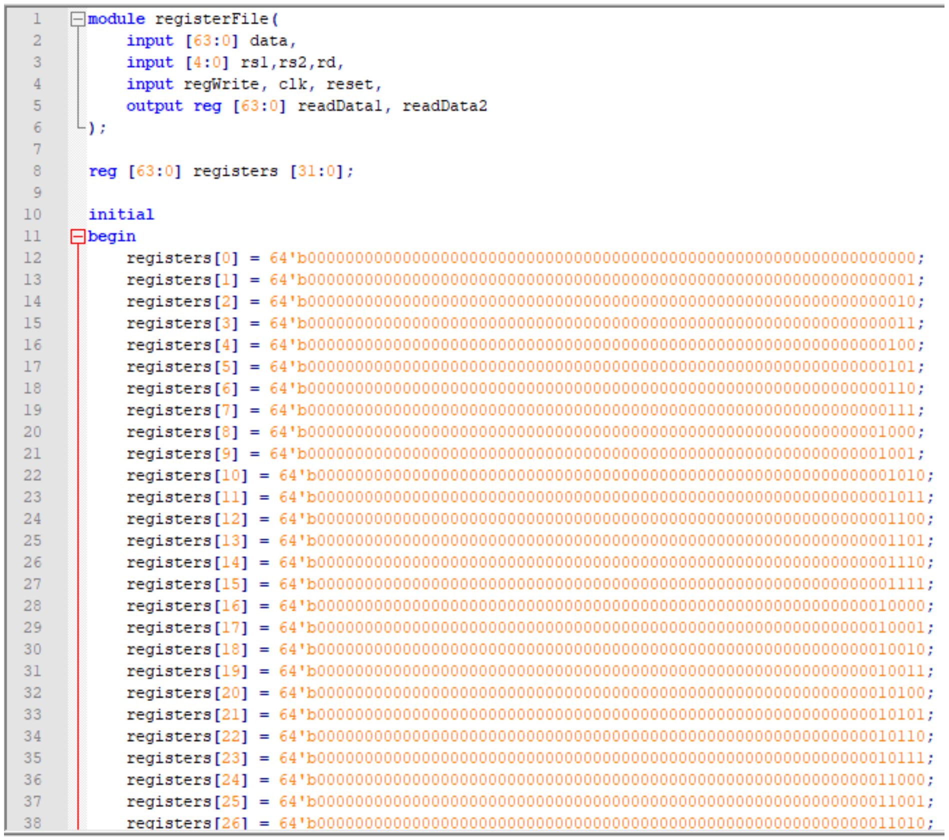
```verilog
module IR1(
    input clk, reset,
    input [63:0] PC_Out,
    input [31:0] Instruction,
    output reg [63:0] PC_Out_IR1,
    output reg [31:0] Instruction_IR1
);

always @ (posedge clk or posedge reset)
begin
    if (reset)
        PC_Out_IR1 = 64'd0;
    else
    begin
        PC_Out_IR1 <= PC_Out;
        Instruction_IR1 <= Instruction;
    end
end

endmodule
```

Instruction Parser:

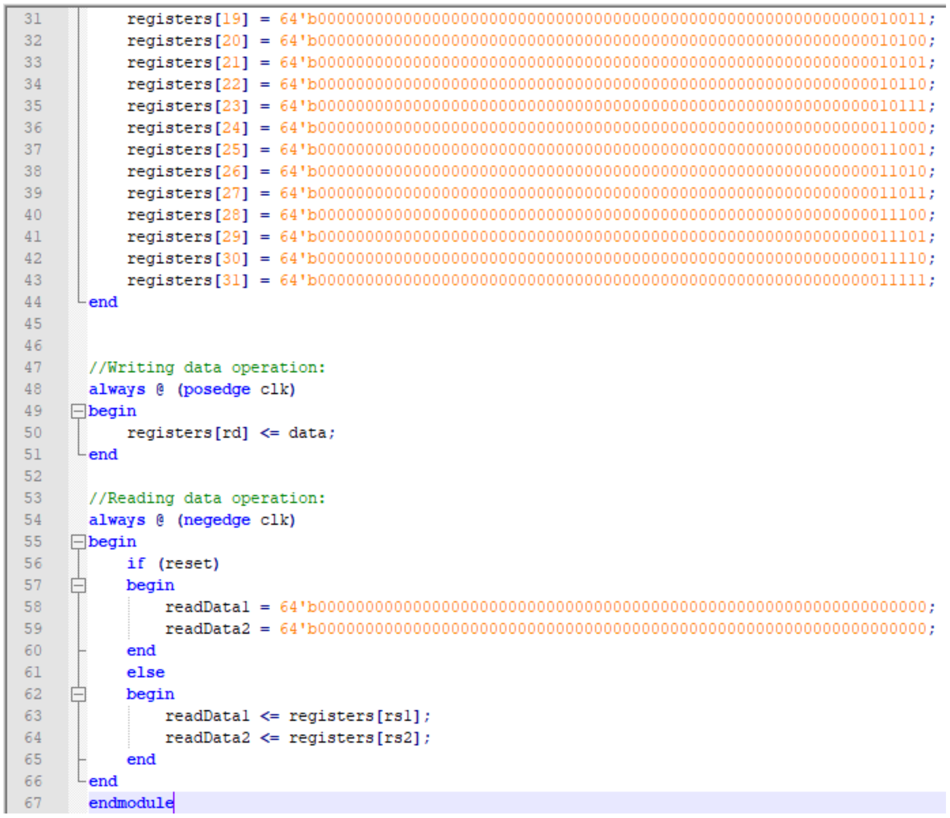


```
module Inst_parser(
    input [31:0] instruction,
    output reg [6:0] opcode, funct7,
    output reg [4:0] rs1, rs2, rd,
    output reg [2:0] funct3
);


always @(instruction)
begin
funct7 = instruction[31:25];
rs2 = instruction[24:20];
rs1 = instruction[19:15];
funct3  = instruction[14:12];
rd = instruction[11:7];
opcode = instruction[6:0];
end


endmodule
```

Register File:

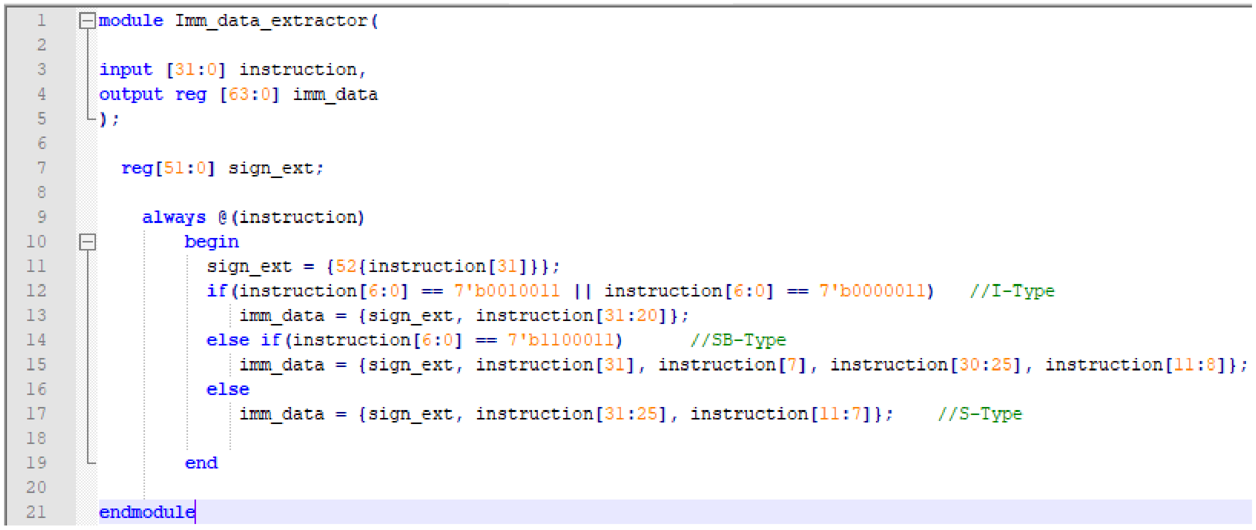


```
module registerFile(
    input [63:0] data,
    input [4:0] rs1,rs2,rd,
    input regWrite, clk, reset,
    output reg [63:0] readData1, readData2
);

reg [63:0] registers [31:0];

initial
begin
    registers[0] = 64'b0000000000000000000000000000000000000000000000000000000000000000;
    registers[1] = 64'b0000000000000000000000000000000000000000000000000000000000000001;
    registers[2] = 64'b0000000000000000000000000000000000000000000000000000000000000010;
    registers[3] = 64'b0000000000000000000000000000000000000000000000000000000000000011;
    registers[4] = 64'b0000000000000000000000000000000000000000000000000000000000000100;
    registers[5] = 64'b0000000000000000000000000000000000000000000000000000000000000101;
    registers[6] = 64'b0000000000000000000000000000000000000000000000000000000000000110;
    registers[7] = 64'b0000000000000000000000000000000000000000000000000000000000000111;
    registers[8] = 64'b0000000000000000000000000000000000000000000000000000000000001000;
    registers[9] = 64'b0000000000000000000000000000000000000000000000000000000000001001;
    registers[10] = 64'b0000000000000000000000000000000000000000000000000000000000001010;
    registers[11] = 64'b0000000000000000000000000000000000000000000000000000000000001011;
    registers[12] = 64'b0000000000000000000000000000000000000000000000000000000000001100;
    registers[13] = 64'b0000000000000000000000000000000000000000000000000000000000001101;
    registers[14] = 64'b0000000000000000000000000000000000000000000000000000000000001110;
    registers[15] = 64'b0000000000000000000000000000000000000000000000000000000000001111;
    registers[16] = 64'b0000000000000000000000000000000000000000000000000000000000010000;
    registers[17] = 64'b0000000000000000000000000000000000000000000000000000000000010001;
    registers[18] = 64'b0000000000000000000000000000000000000000000000000000000000010010;
    registers[19] = 64'b0000000000000000000000000000000000000000000000000000000000010011;
    registers[20] = 64'b0000000000000000000000000000000000000000000000000000000000010100;
    registers[21] = 64'b0000000000000000000000000000000000000000000000000000000000010101;
    registers[22] = 64'b0000000000000000000000000000000000000000000000000000000000010110;
    registers[23] = 64'b0000000000000000000000000000000000000000000000000000000000010111;
    registers[24] = 64'b0000000000000000000000000000000000000000000000000000000000011000;
    registers[25] = 64'b0000000000000000000000000000000000000000000000000000000000011001;
    registers[26] = 64'b0000000000000000000000000000000000000000000000000000000000011010;
```



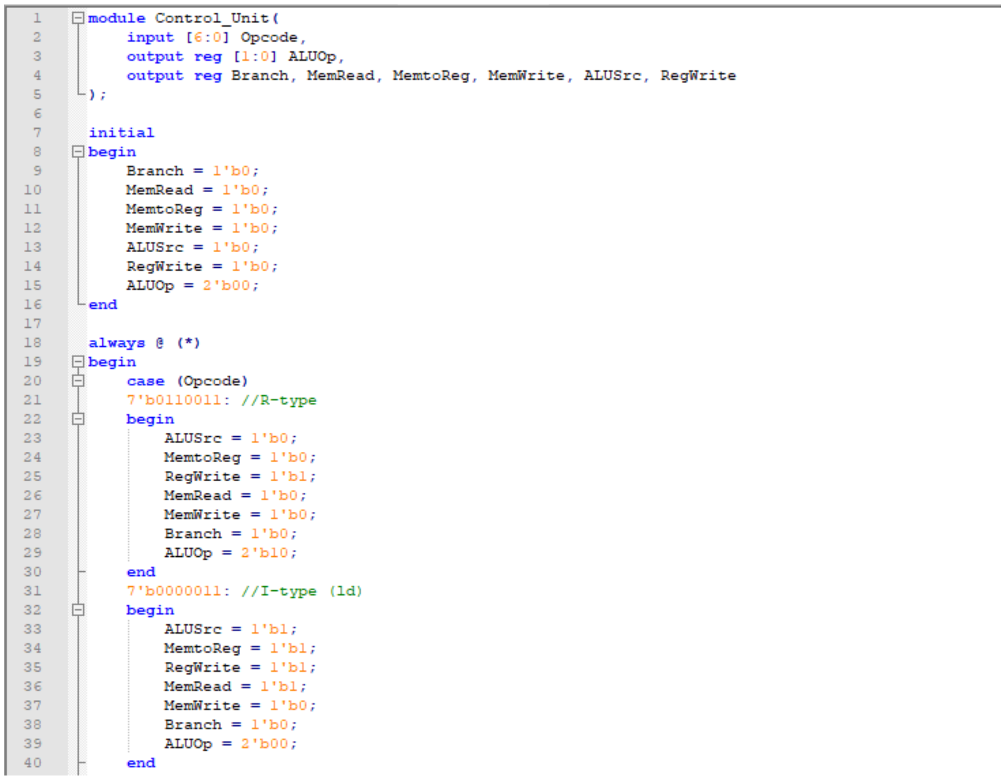
```verilog
    registers[19] = 64'b0000000000000000000000000000000000000000000000000000000000010011;
    registers[20] = 64'b0000000000000000000000000000000000000000000000000000000000010100;
    registers[21] = 64'b0000000000000000000000000000000000000000000000000000000000010101;
    registers[22] = 64'b0000000000000000000000000000000000000000000000000000000000010110;
    registers[23] = 64'b0000000000000000000000000000000000000000000000000000000000010111;
    registers[24] = 64'b0000000000000000000000000000000000000000000000000000000000011000;
    registers[25] = 64'b0000000000000000000000000000000000000000000000000000000000011001;
    registers[26] = 64'b0000000000000000000000000000000000000000000000000000000000011010;
    registers[27] = 64'b0000000000000000000000000000000000000000000000000000000000011011;
    registers[28] = 64'b0000000000000000000000000000000000000000000000000000000000011100;
    registers[29] = 64'b0000000000000000000000000000000000000000000000000000000000011101;
    registers[30] = 64'b0000000000000000000000000000000000000000000000000000000000011110;
    registers[31] = 64'b0000000000000000000000000000000000000000000000000000000000011111;
end


//Writing data operation:
always @ (posedge clk)
begin
    registers[rd] <= data;
end

//Reading data operation:
always @ (negedge clk)
begin
    if (reset)
    begin
        readData1 = 64'b0000000000000000000000000000000000000000000000000000000000000000;
        readData2 = 64'b0000000000000000000000000000000000000000000000000000000000000000;
    end
    else
    begin
        readData1 <= registers[rs1];
        readData2 <= registers[rs2];
    end
end
endmodule
```

Immediate Data Extractor:

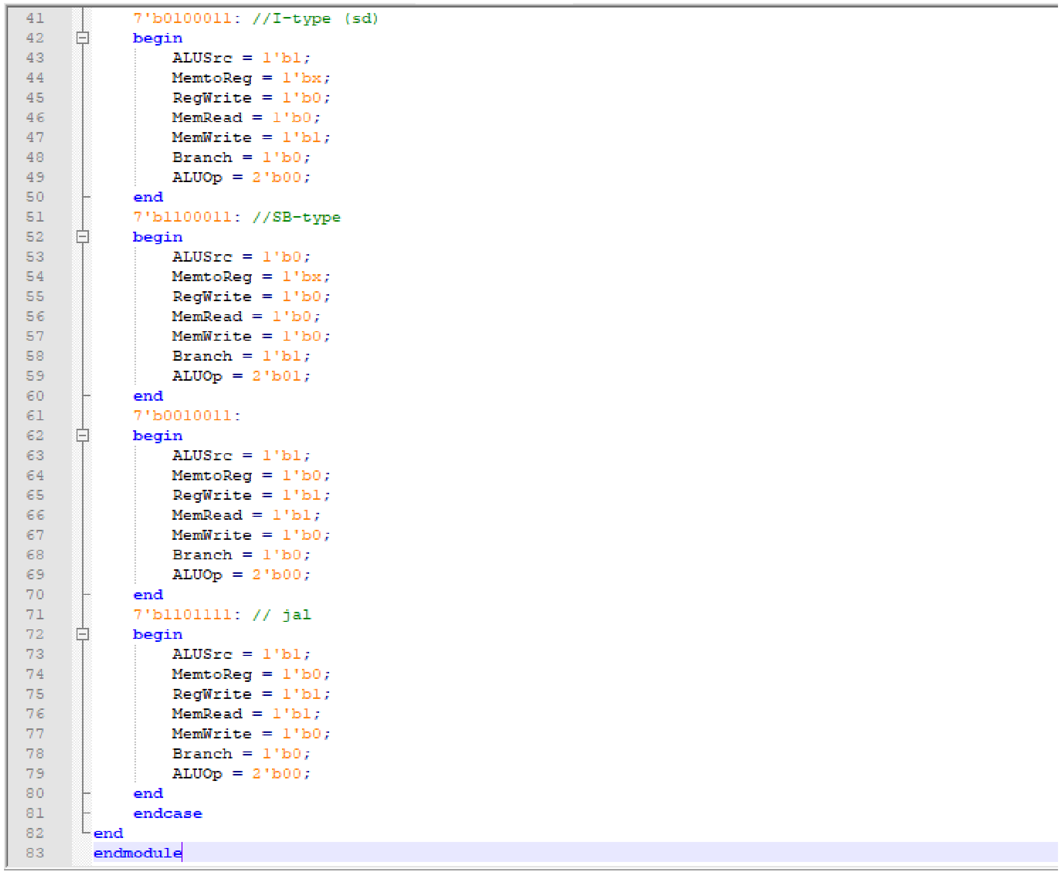


```verilog
module Imm_data_extractor(

input [31:0] instruction,
output reg [63:0] imm_data
);

  reg[51:0] sign_ext;

    always @(instruction)
        begin
          sign_ext = {52{instruction[31]}};
          if(instruction[6:0] == 7'b0010011 || instruction[6:0] == 7'b0000011)    //I-Type
              imm_data = {sign_ext, instruction[31:20]};
          else if(instruction[6:0] == 7'b1100011)      //SB-Type
              imm_data = {sign_ext, instruction[31], instruction[7], instruction[30:25], instruction[11:8]};
          else
              imm_data = {sign_ext, instruction[31:25], instruction[11:7]};    //S-Type

        end

endmodule
```

Control Unit:

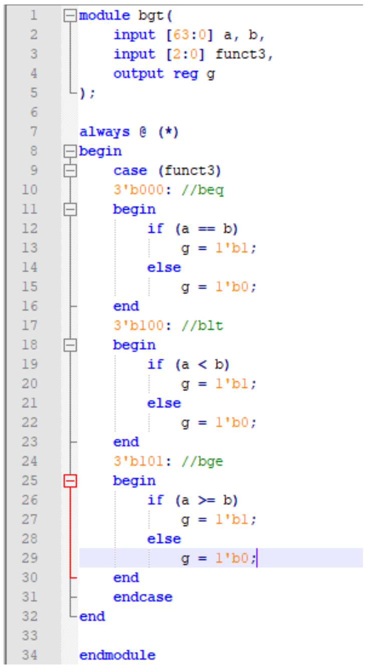


```verilog
module Control_Unit(
    input [6:0] Opcode,
    output reg [1:0] ALUOp,
    output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite
);

initial
begin
    Branch = 1'b0;
    MemRead = 1'b0;
    MemtoReg = 1'b0;
    MemWrite = 1'b0;
    ALUSrc = 1'b0;
    RegWrite = 1'b0;
    ALUOp = 2'b00;
end

always @ (*)
begin
    case (Opcode)
    7'b0110011: //R-type
    begin
        ALUSrc = 1'b0;
        MemtoReg = 1'b0;
        RegWrite = 1'b1;
        MemRead = 1'b0;
        MemWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 2'b10;
    end
    7'b0000011: //I-type (ld)
    begin
        ALUSrc = 1'b1;
        MemtoReg = 1'b1;
        RegWrite = 1'b1;
        MemRead = 1'b1;
        MemWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 2'b00;
    end
```



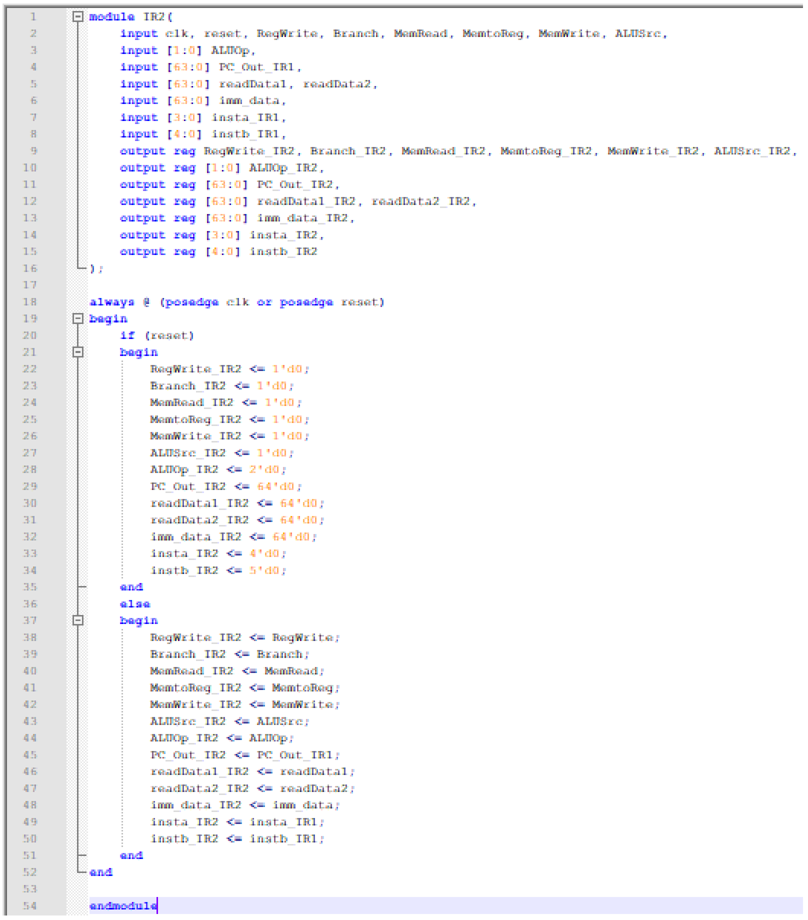
```verilog
    7'b0100011: //I-type (sd)
    begin
        ALUSrc = 1'b1;
        MemtoReg = 1'bx;
        RegWrite = 1'b0;
        MemRead = 1'b0;
        MemWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 2'b00;
    end
    7'b1100011: //SB-type
    begin
        ALUSrc = 1'b0;
        MemtoReg = 1'bx;
        RegWrite = 1'b0;
        MemRead = 1'b0;
        MemWrite = 1'b0;
        Branch = 1'b1;
        ALUOp = 2'b01;
    end
    7'b0010011:
    begin
        ALUSrc = 1'b1;
        MemtoReg = 1'b0;
        RegWrite = 1'b1;
        MemRead = 1'b1;
        MemWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 2'b00;
    end
    7'b1101111: // jal
    begin
        ALUSrc = 1'b1;
        MemtoReg = 1'b0;
        RegWrite = 1'b1;
        MemRead = 1'b1;
        MemWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 2'b00;
    end
    endcase
end
endmodule
```

BGT:



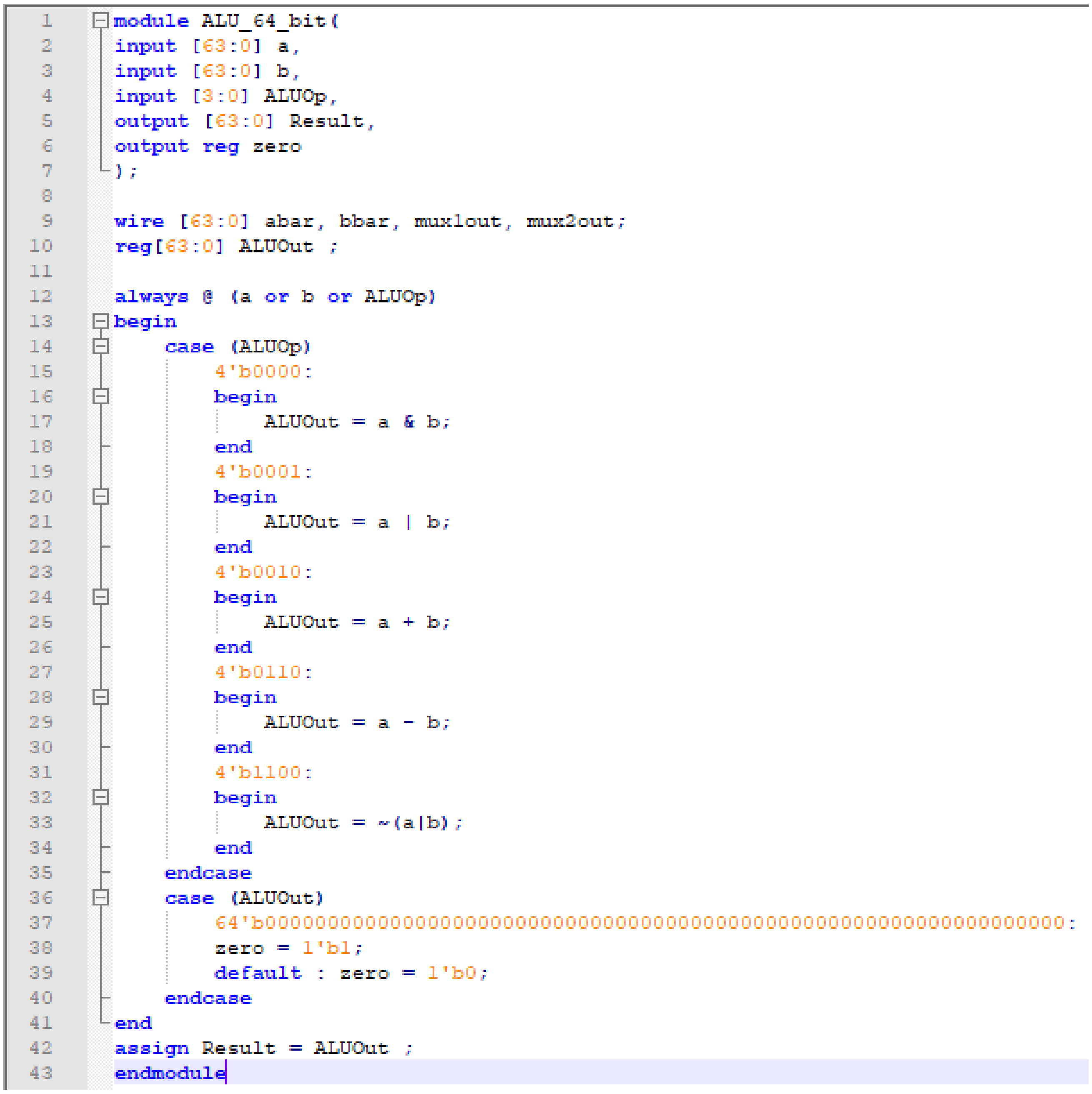
```verilog
module bgt(
    input [63:0] a, b,
    input [2:0] funct3,
    output reg g
);

always @ (*)
begin
    case (funct3)
    3'b000: //beq
    begin
        if (a == b)
            g = 1'b1;
        else
            g = 1'b0;
    end
    3'b100: //blt
    begin
        if (a < b)
            g = 1'b1;
        else
            g = 1'b0;
    end
    3'b101: //bge
    begin
        if (a >= b)
            g = 1'b1;
        else
            g = 1'b0;
    end
    endcase
end

endmodule
```

ID/EX Pipeline register:



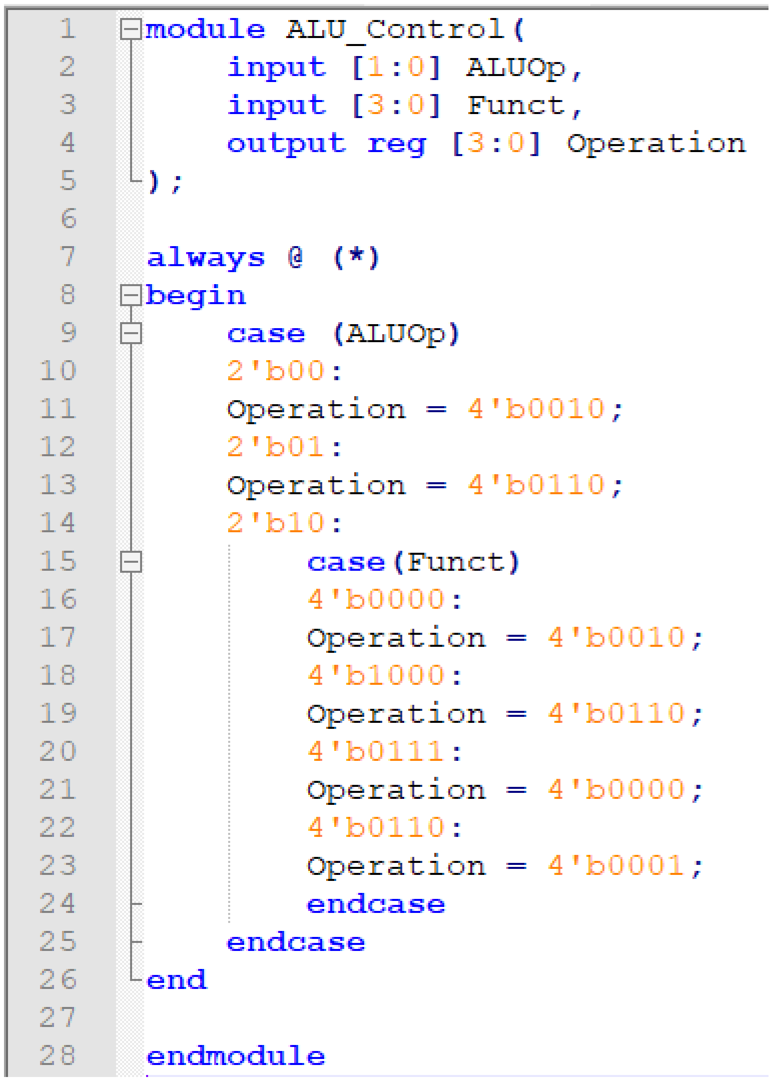
```verilog
module IR2(
    input clk, reset, RegWrite, Branch, MemRead, MemtoReg, MemWrite, ALUSrc,
    input [1:0] ALUOp,
    input [63:0] PC_Out_IR1,
    input [63:0] readData1, readData2,
    input [63:0] imm_data,
    input [3:0] insta_IR1,
    input [4:0] instb_IR1,
    output reg RegWrite_IR2, Branch_IR2, MemRead_IR2, MemtoReg_IR2, MemWrite_IR2, ALUSrc_IR2,
    output reg [1:0] ALUOp_IR2,
    output reg [63:0] PC_Out_IR2,
    output reg [63:0] readData1_IR2, readData2_IR2,
    output reg [63:0] imm_data_IR2,
    output reg [3:0] insta_IR2,
    output reg [4:0] instb_IR2
);

always @ (posedge clk or posedge reset)
begin
    if (reset)
    begin
        RegWrite_IR2 <= 1'd0;
        Branch_IR2 <= 1'd0;
        MemRead_IR2 <= 1'd0;
        MemtoReg_IR2 <= 1'd0;
        MemWrite_IR2 <= 1'd0;
        ALUSrc_IR2 <= 1'd0;
        ALUOp_IR2 <= 2'd0;
        PC_Out_IR2 <= 64'd0;
        readData1_IR2 <= 64'd0;
        readData2_IR2 <= 64'd0;
        imm_data_IR2 <= 64'd0;
        insta_IR2 <= 4'd0;
        instb_IR2 <= 5'd0;
    end
    else
    begin
        RegWrite_IR2 <= RegWrite;
        Branch_IR2 <= Branch;
        MemRead_IR2 <= MemRead;
        MemtoReg_IR2 <= MemtoReg;
        MemWrite_IR2 <= MemWrite;
        ALUSrc_IR2 <= ALUSrc;
        ALUOp_IR2 <= ALUOp;
        PC_Out_IR2 <= PC_Out_IR1;
        readData1_IR2 <= readData1;
        readData2_IR2 <= readData2;
        imm_data_IR2 <= imm_data;
        insta_IR2 <= insta_IR1;
        instb_IR2 <= instb_IR1;
    end
end

endmodule
```

ALU:



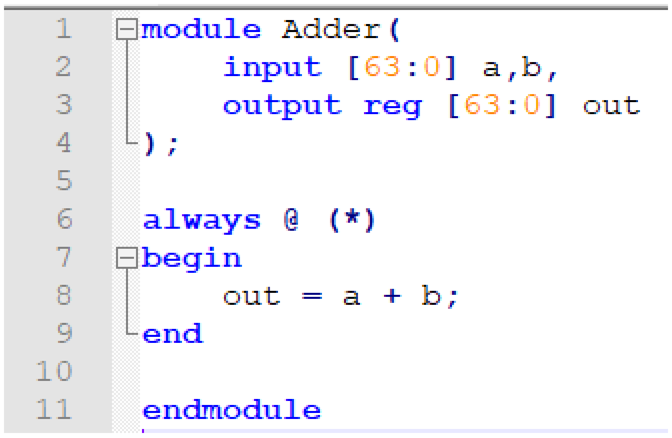
```verilog
module ALU_64_bit(
input [63:0] a,
input [63:0] b,
input [3:0] ALUOp,
output [63:0] Result,
output reg zero
);

wire [63:0] abar, bbar, mux1out, mux2out;
reg[63:0] ALUOut ;

always @ (a or b or ALUOp)
begin
    case (ALUOp)
        4'b0000:
        begin
            ALUOut = a & b;
        end
        4'b0001:
        begin
            ALUOut = a | b;
        end
        4'b0010:
        begin
            ALUOut = a + b;
        end
        4'b0110:
        begin
            ALUOut = a - b;
        end
        4'b1100:
        begin
            ALUOut = ~(a|b);
        end
    endcase
    case (ALUOut)
        64'b0000000000000000000000000000000000000000000000000000000000000000:
        zero = 1'b1;
        default : zero = 1'b0;
    endcase
end
assign Result = ALUOut ;
endmodule
```

ALU Control:



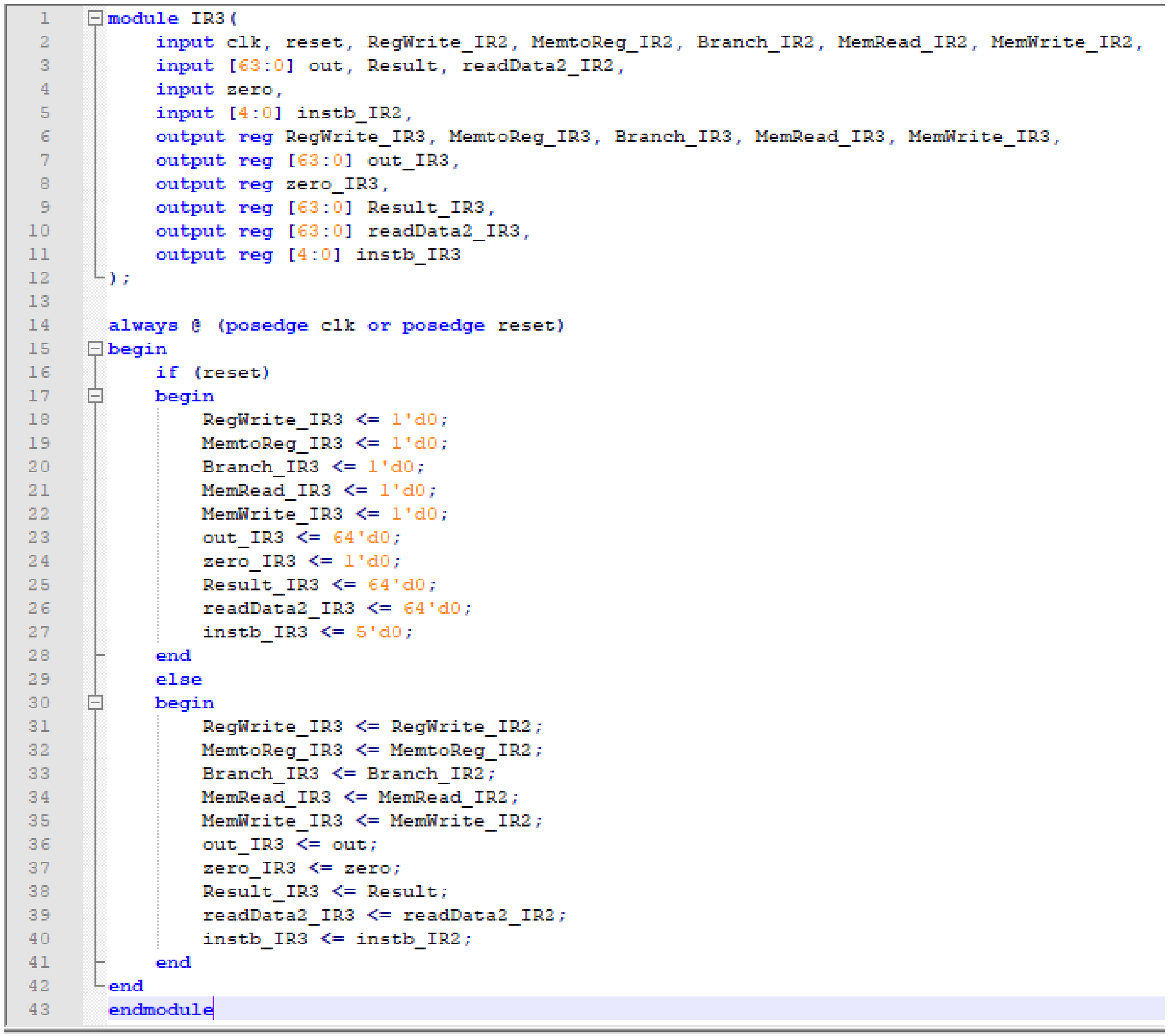
```
module ALU_Control(
    input [1:0] ALUOp,
    input [3:0] Funct,
    output reg [3:0] Operation
);

always @ (*)
begin
    case (ALUOp)
    2'b00:
    Operation = 4'b0010;
    2'b01:
    Operation = 4'b0110;
    2'b10:
        case(Funct)
        4'b0000:
        Operation = 4'b0010;
        4'b1000:
        Operation = 4'b0110;
        4'b0111:
        Operation = 4'b0000;
        4'b0110:
        Operation = 4'b0001;
        endcase
    endcase
end

endmodule
```

Adder:



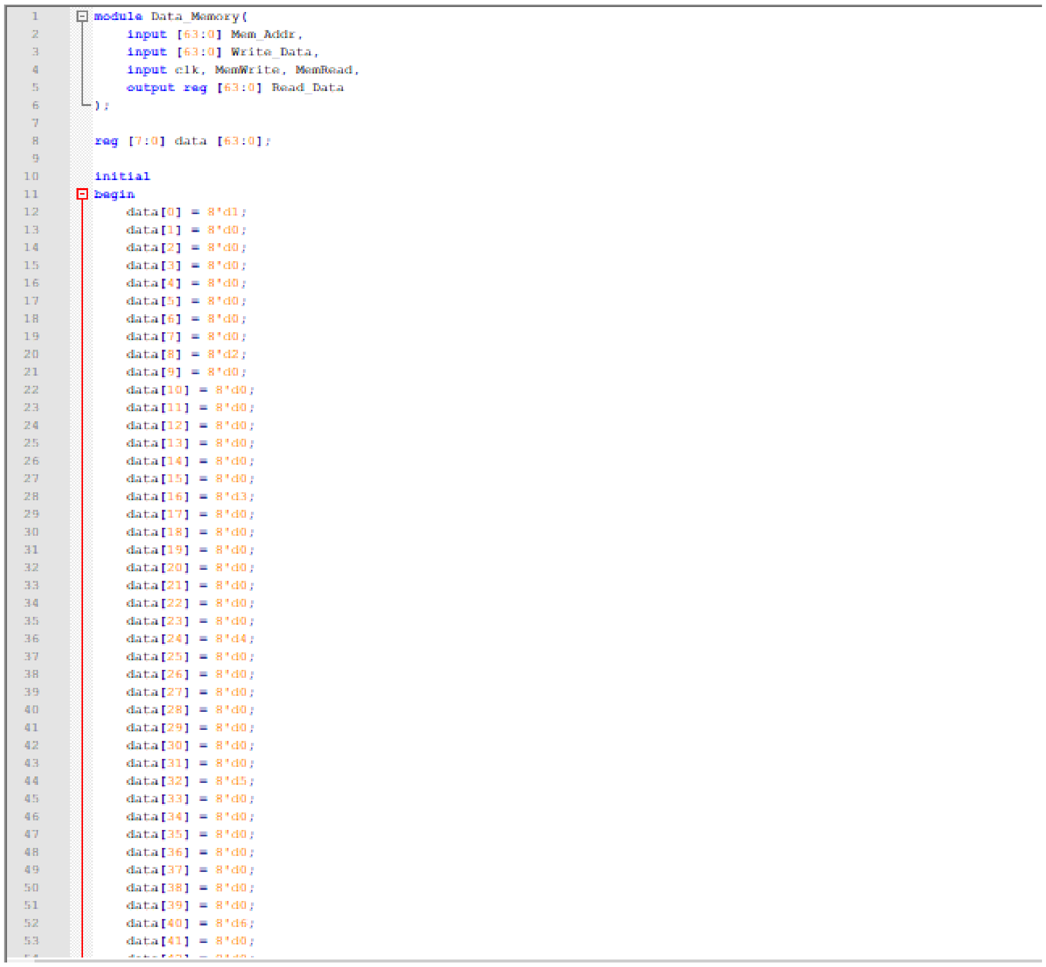
```
module Adder(
    input [63:0] a,b,
    output reg [63:0] out
);

always @ (*)
begin
    out = a + b;
end

endmodule
```

EX/MEM Pipeline Register:

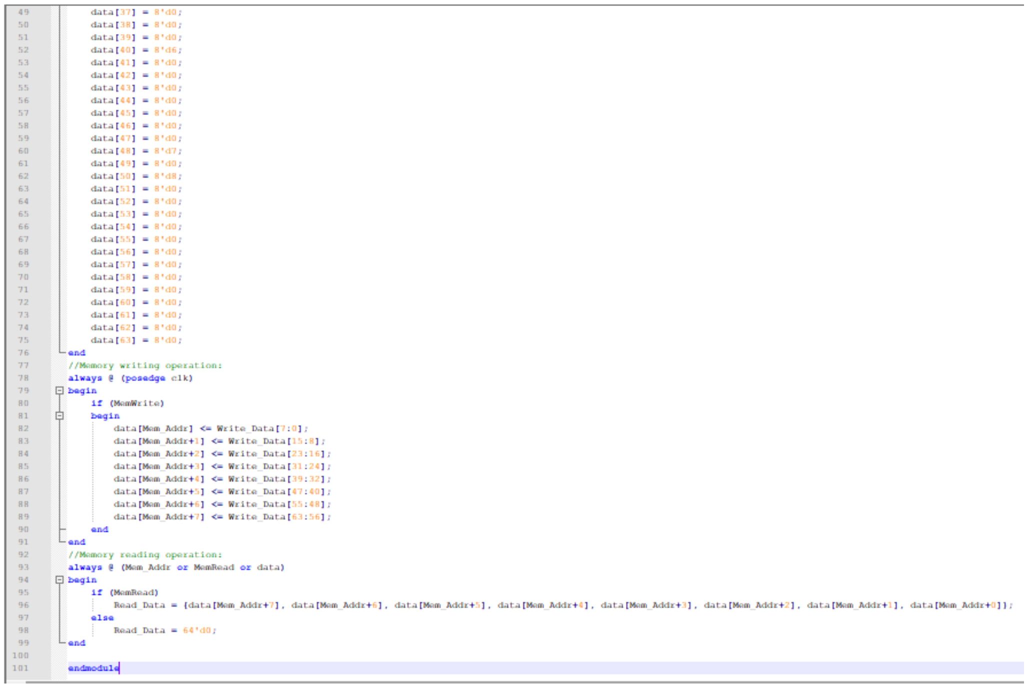


```
module IR3(
    input clk, reset, RegWrite_IR2, MemtoReg_IR2, Branch_IR2, MemRead_IR2, MemWrite_IR2,
    input [63:0] out, Result, readData2_IR2,
    input zero,
    input [4:0] instb_IR2,
    output reg RegWrite_IR3, MemtoReg_IR3, Branch_IR3, MemRead_IR3, MemWrite_IR3,
    output reg [63:0] out_IR3,
    output reg zero_IR3,
    output reg [63:0] Result_IR3,
    output reg [63:0] readData2_IR3,
    output reg [4:0] instb_IR3
);

always @ (posedge clk or posedge reset)
begin
    if (reset)
    begin
        RegWrite_IR3 <= 1'd0;
        MemtoReg_IR3 <= 1'd0;
        Branch_IR3 <= 1'd0;
        MemRead_IR3 <= 1'd0;
        MemWrite_IR3 <= 1'd0;
        out_IR3 <= 64'd0;
        zero_IR3 <= 1'd0;
        Result_IR3 <= 64'd0;
        readData2_IR3 <= 64'd0;
        instb_IR3 <= 5'd0;
    end
    else
    begin
        RegWrite_IR3 <= RegWrite_IR2;
        MemtoReg_IR3 <= MemtoReg_IR2;
        Branch_IR3 <= Branch_IR2;
        MemRead_IR3 <= MemRead_IR2;
        MemWrite_IR3 <= MemWrite_IR2;
        out_IR3 <= out;
        zero_IR3 <= zero;
        Result_IR3 <= Result;
        readData2_IR3 <= readData2_IR2;
        instb_IR3 <= instb_IR2;
    end
end
endmodule
```

Data Memory:

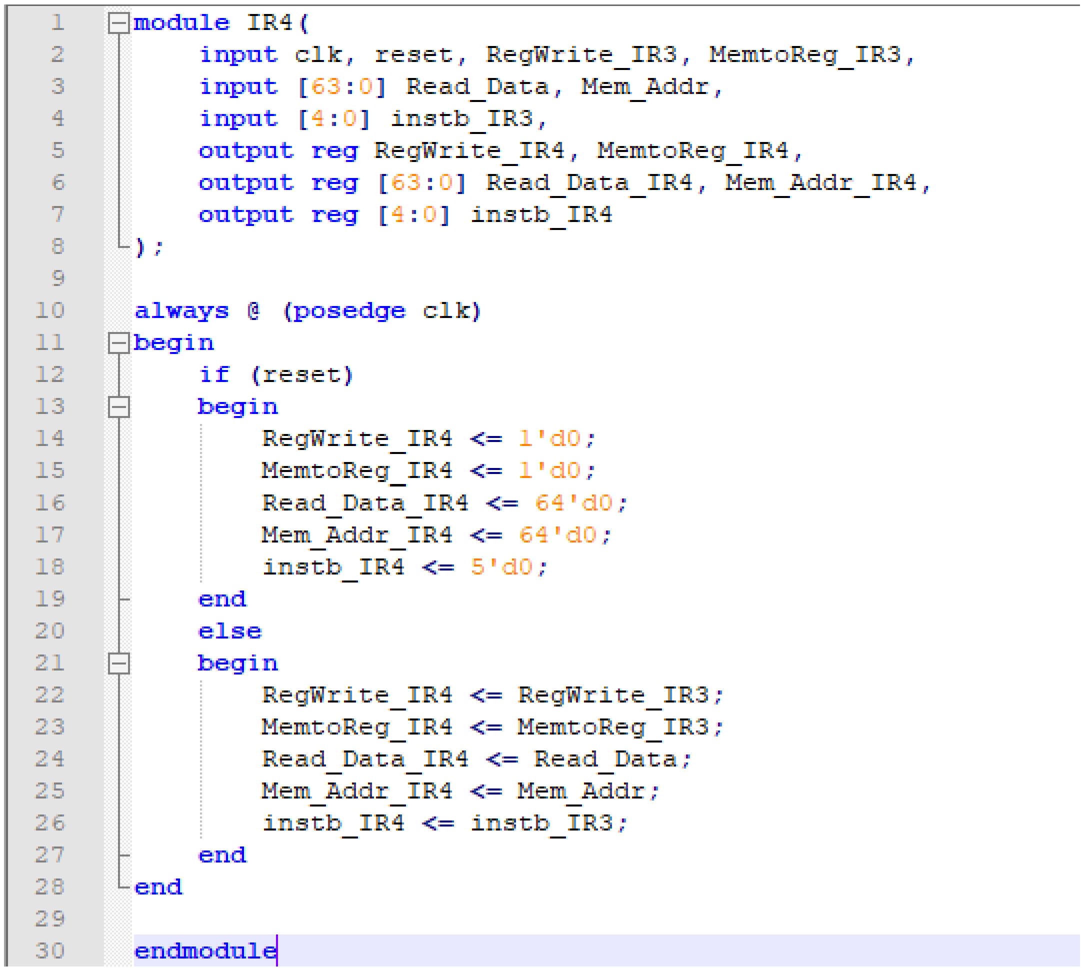


```
module Data_Memory(
    input [63:0] Mem_Addr,
    input [63:0] Write_Data,
    input clk, MemWrite, MemRead,
    output reg [63:0] Read_Data
);

reg [7:0] data [63:0];

initial
begin
    data[0] = 8'd1;
    data[1] = 8'd0;
    data[2] = 8'd0;
    data[3] = 8'd0;
    data[4] = 8'd0;
    data[5] = 8'd0;
    data[6] = 8'd0;
    data[7] = 8'd0;
    data[8] = 8'd2;
    data[9] = 8'd0;
    data[10] = 8'd0;
    data[11] = 8'd0;
    data[12] = 8'd0;
    data[13] = 8'd0;
    data[14] = 8'd0;
    data[15] = 8'd0;
    data[16] = 8'd3;
    data[17] = 8'd0;
    data[18] = 8'd0;
    data[19] = 8'd0;
    data[20] = 8'd0;
    data[21] = 8'd0;
    data[22] = 8'd0;
    data[23] = 8'd0;
    data[24] = 8'd4;
    data[25] = 8'd0;
    data[26] = 8'd0;
    data[27] = 8'd0;
    data[28] = 8'd0;
    data[29] = 8'd0;
    data[30] = 8'd0;
    data[31] = 8'd0;
    data[32] = 8'd5;
    data[33] = 8'd0;
    data[34] = 8'd0;
    data[35] = 8'd0;
    data[36] = 8'd0;
    data[37] = 8'd0;
    data[38] = 8'd0;
    data[39] = 8'd0;
    data[40] = 8'd6;
    data[41] = 8'd0;
```



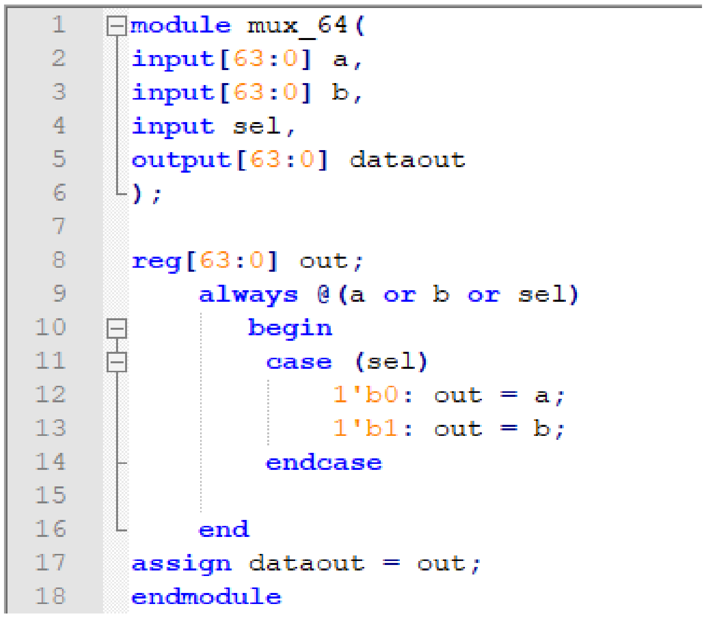
```
    data[37] = 8'd0;
    data[38] = 8'd0;
    data[39] = 8'd0;
    data[40] = 8'd6;
    data[41] = 8'd0;
    data[42] = 8'd0;
    data[43] = 8'd0;
    data[44] = 8'd0;
    data[45] = 8'd0;
    data[46] = 8'd0;
    data[47] = 8'd0;
    data[48] = 8'd7;
    data[49] = 8'd0;
    data[50] = 8'd8;
    data[51] = 8'd0;
    data[52] = 8'd0;
    data[53] = 8'd0;
    data[54] = 8'd0;
    data[55] = 8'd0;
    data[56] = 8'd0;
    data[57] = 8'd0;
    data[58] = 8'd0;
    data[59] = 8'd0;
    data[60] = 8'd0;
    data[61] = 8'd0;
    data[62] = 8'd0;
    data[63] = 8'd0;
end
//Memory writing operation:
always @ (posedge clk)
begin
    if (MemWrite)
    begin
        data[Mem_Addr] <= Write_Data[7:0];
        data[Mem_Addr+1] <= Write_Data[15:8];
        data[Mem_Addr+2] <= Write_Data[23:16];
        data[Mem_Addr+3] <= Write_Data[31:24];
        data[Mem_Addr+4] <= Write_Data[39:32];
        data[Mem_Addr+5] <= Write_Data[47:40];
        data[Mem_Addr+6] <= Write_Data[55:48];
        data[Mem_Addr+7] <= Write_Data[63:56];
    end
end
//Memory reading operation:
always @ (Mem_Addr or MemRead or data)
begin
    if (MemRead)
        Read_Data = {data[Mem_Addr+7], data[Mem_Addr+6], data[Mem_Addr+5], data[Mem_Addr+4], data[Mem_Addr+3], data[Mem_Addr+2], data[Mem_Addr+1], data[Mem_Addr+0]};
    else
        Read_Data = 64'd0;
end

endmodule
```

MEM/WB Pipeline Register:



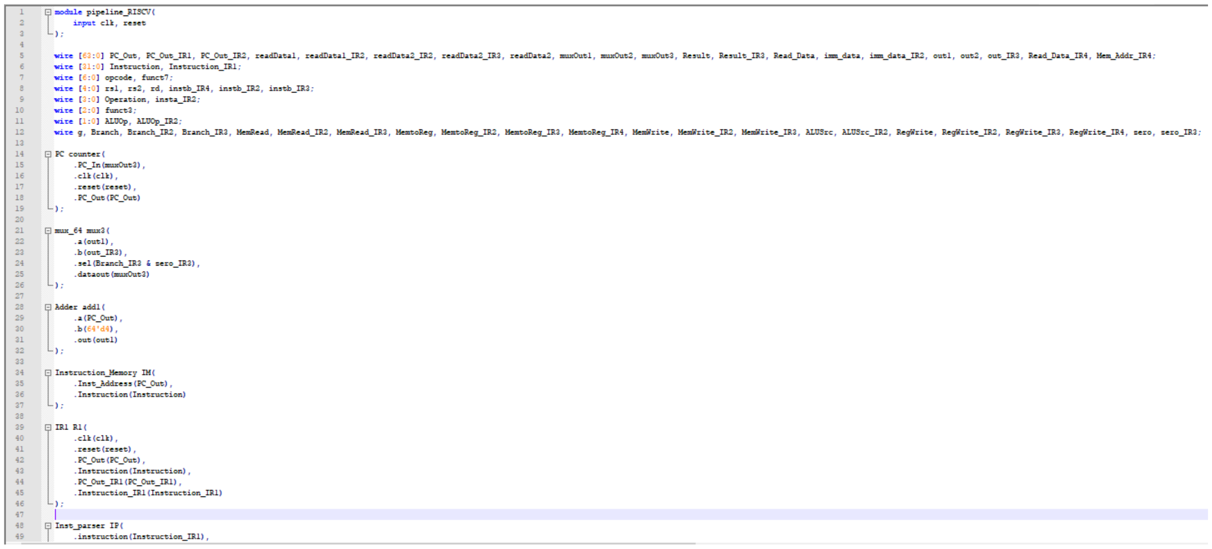
```
module IR4(
    input clk, reset, RegWrite_IR3, MemtoReg_IR3,
    input [63:0] Read_Data, Mem_Addr,
    input [4:0] instb_IR3,
    output reg RegWrite_IR4, MemtoReg_IR4,
    output reg [63:0] Read_Data_IR4, Mem_Addr_IR4,
    output reg [4:0] instb_IR4
);

always @ (posedge clk)
begin
    if (reset)
    begin
        RegWrite_IR4 <= 1'd0;
        MemtoReg_IR4 <= 1'd0;
        Read_Data_IR4 <= 64'd0;
        Mem_Addr_IR4 <= 64'd0;
        instb_IR4 <= 5'd0;
    end
    else
    begin
        RegWrite_IR4 <= RegWrite_IR3;
        MemtoReg_IR4 <= MemtoReg_IR3;
        Read_Data_IR4 <= Read_Data;
        Mem_Addr_IR4 <= Mem_Addr;
        instb_IR4 <= instb_IR3;
    end
end

endmodule
```

Mux:

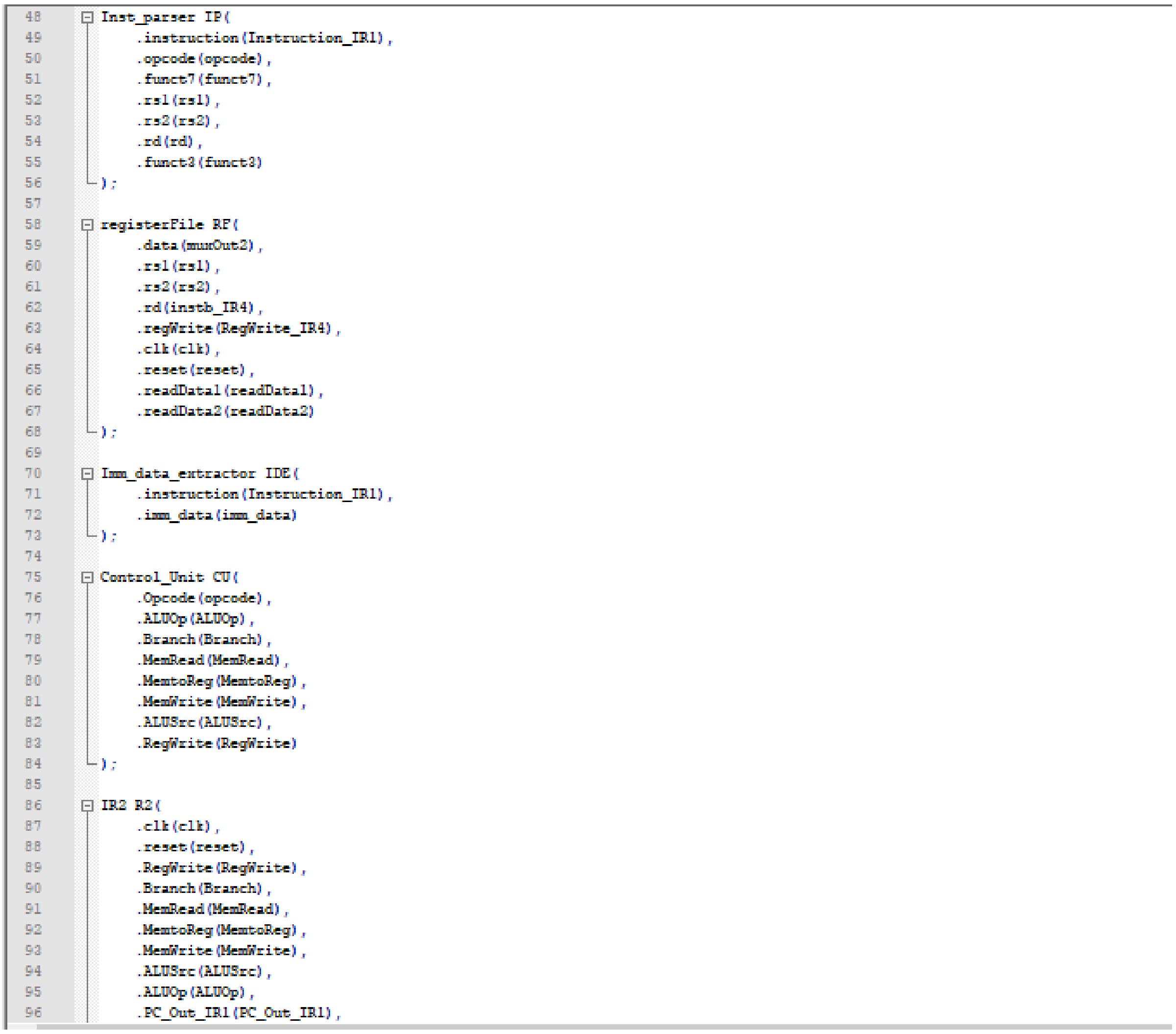


```
module mux_64(
input[63:0] a,
input[63:0] b,
input sel,
output[63:0] dataout
);

reg[63:0] out;
    always @(a or b or sel)
       begin
        case (sel)
            1'b0: out = a;
            1'b1: out = b;
        endcase

    end
assign dataout = out;
endmodule
```

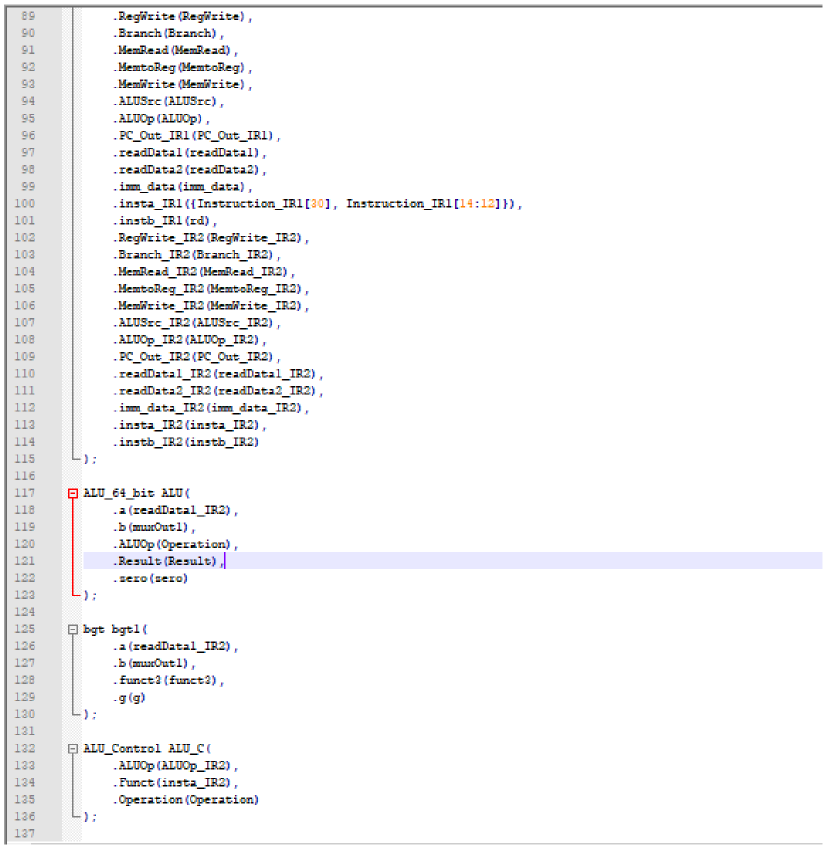
Top module:



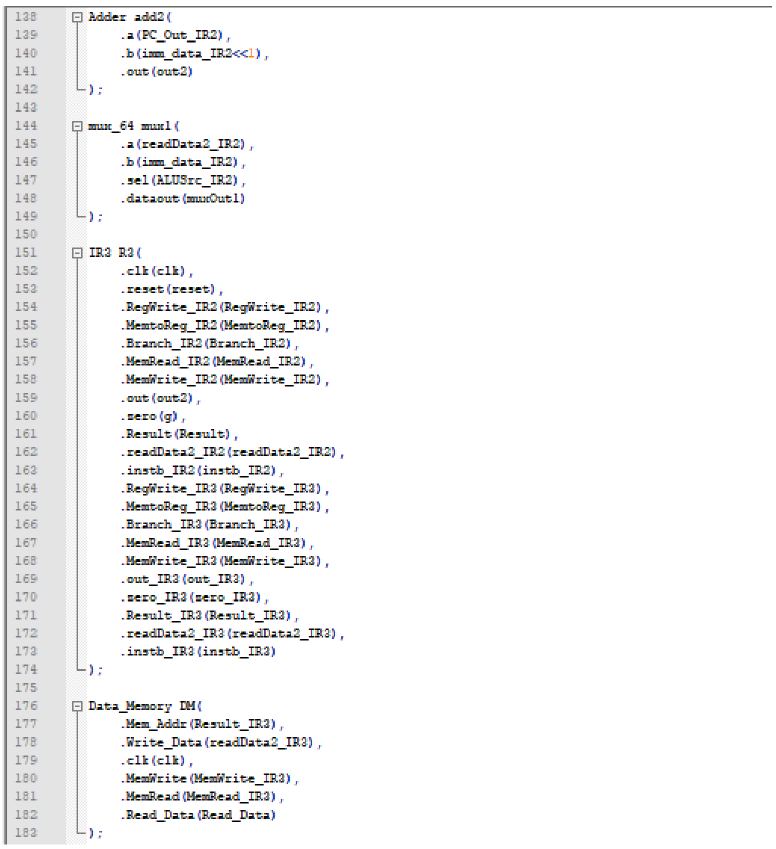
```verilog
module pipeline_RISCV(
    input clk, reset
);

wire [63:0] PC_Out, PC_Out_IR1, PC_Out_IR2, readData1, readData1_IR2, readData2_IR2, readData2_IR3, readData2, muxOut1, muxOut2, muxOut3, Result, Result_IR3, Read_Data, imm_data, imm_data_IR2, out1, out2, out_IR3, Read_Data_IR4, Mem_Addr_IR4;
wire [31:0] Instruction, Instruction_IR1;
wire [6:0] opcode, funct7;
wire [4:0] rs1, rs2, rd, instb_IR4, instb_IR2, instb_IR3;
wire [3:0] Operation, insta_IR2;
wire [2:0] funct3;
wire [1:0] ALUOp, ALUOp_IR2;
wire g, Branch, Branch_IR2, Branch_IR3, MemRead, MemRead_IR2, MemRead_IR3, MemtoReg, MemtoReg_IR2, MemtoReg_IR3, MemtoReg_IR4, MemWrite, MemWrite_IR2, MemWrite_IR3, ALUSrc, ALUSrc_IR2, RegWrite, RegWrite_IR2, RegWrite_IR3, RegWrite_IR4, sero, sero_IR3;

PC counter(
    .PC_In(muxOut3),
    .clk(clk),
    .reset(reset),
    .PC_Out(PC_Out)
);

mux_64 mux3(
    .a(out1),
    .b(out_IR3),
    .sel(Branch_IR3 & sero_IR3),
    .dataout(muxOut3)
);

Adder add1(
    .a(PC_Out),
    .b(64'd4),
    .out(out1)
);

Instruction_Memory IM(
    .Inst_Address(PC_Out),
    .Instruction(Instruction)
);

IR1 R1(
    .clk(clk),
    .reset(reset),
    .PC_Out(PC_Out),
    .Instruction(Instruction),
    .PC_Out_IR1(PC_Out_IR1),
    .Instruction_IR1(Instruction_IR1)
);

Inst_parser IP(
    .instruction(Instruction_IR1),
```



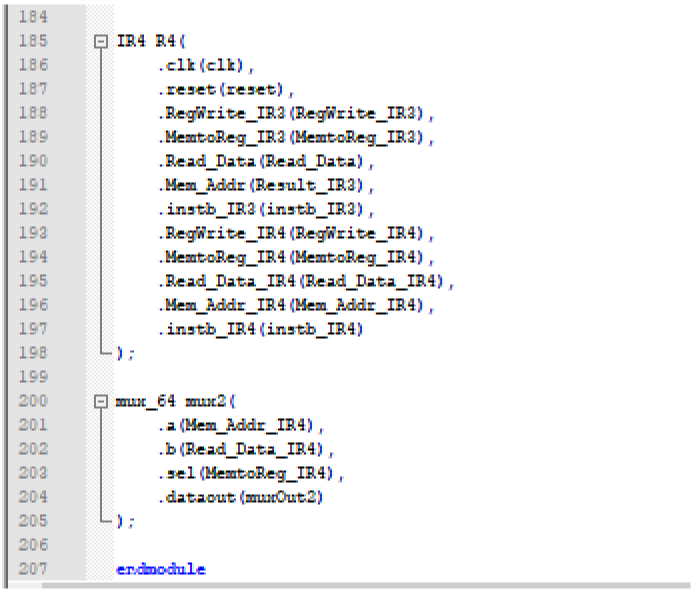
```verilog
Inst_parser IP(
    .instruction(Instruction_IR1),
    .opcode(opcode),
    .funct7(funct7),
    .rs1(rs1),
    .rs2(rs2),
    .rd(rd),
    .funct3(funct3)
);

registerFile RF(
    .data(muxOut2),
    .rs1(rs1),
    .rs2(rs2),
    .rd(instb_IR4),
    .regWrite(RegWrite_IR4),
    .clk(clk),
    .reset(reset),
    .readData1(readData1),
    .readData2(readData2)
);

Imm_data_extractor IDE(
    .instruction(Instruction_IR1),
    .imm_data(imm_data)
);

Control_Unit CU(
    .Opcode(opcode),
    .ALUOp(ALUOp),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .RegWrite(RegWrite)
);

IR2 R2(
    .clk(clk),
    .reset(reset),
    .RegWrite(RegWrite),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .ALUOp(ALUOp),
    .PC_Out_IR1(PC_Out_IR1),
```



```verilog
        .RegWrite(RegWrite),
        .Branch(Branch),
        .MemRead(MemRead),
        .MemtoReg(MemtoReg),
        .MemWrite(MemWrite),
        .ALUSrc(ALUSrc),
        .ALUOp(ALUOp),
        .PC_Out_IR1(PC_Out_IR1),
        .readData1(readData1),
        .readData2(readData2),
        .imm_data(imm_data),
        .insta_IR1({Instruction_IR1[30], Instruction_IR1[14:12]}),
        .instb_IR1(rd),
        .RegWrite_IR2(RegWrite_IR2),
        .Branch_IR2(Branch_IR2),
        .MemRead_IR2(MemRead_IR2),
        .MemtoReg_IR2(MemtoReg_IR2),
        .MemWrite_IR2(MemWrite_IR2),
        .ALUSrc_IR2(ALUSrc_IR2),
        .ALUOp_IR2(ALUOp_IR2),
        .PC_Out_IR2(PC_Out_IR2),
        .readData1_IR2(readData1_IR2),
        .readData2_IR2(readData2_IR2),
        .imm_data_IR2(imm_data_IR2),
        .insta_IR2(insta_IR2),
        .instb_IR2(instb_IR2)
);

ALU_64_bit ALU(
        .a(readData1_IR2),
        .b(muxOut1),
        .ALUOp(Operation),
        .Result(Result),
        .zero(zero)
);

bgt bgt1(
        .a(readData1_IR2),
        .b(muxOut1),
        .funct3(funct3),
        .g(g)
);

ALU_Control ALU_C(
        .ALUOp(ALUOp_IR2),
        .Funct(insta_IR2),
        .Operation(Operation)
);

```

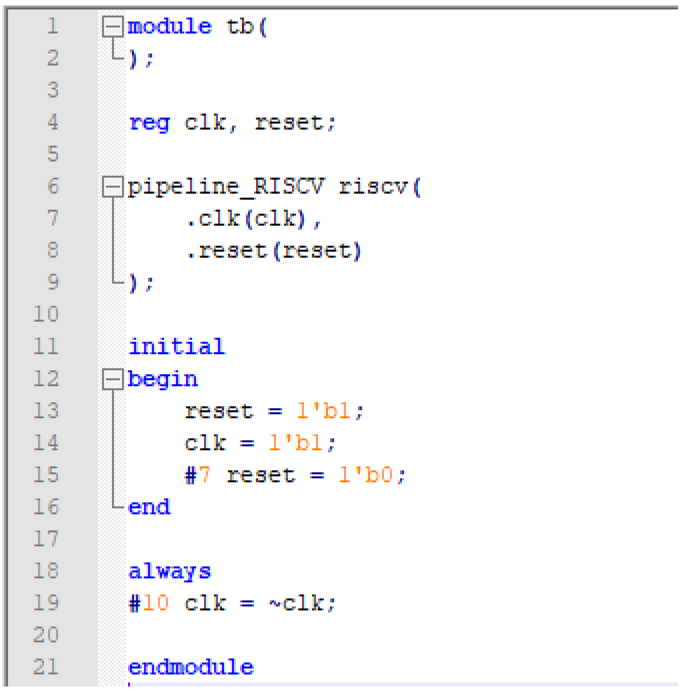


```verilog
Adder add2(
        .a(PC_Out_IR2),
        .b(imm_data_IR2<<1),
        .out(out2)
);

mux_64 mux1(
        .a(readData2_IR2),
        .b(imm_data_IR2),
        .sel(ALUSrc_IR2),
        .dataout(muxOut1)
);

IR3 R3(
        .clk(clk),
        .reset(reset),
        .RegWrite_IR2(RegWrite_IR2),
        .MemtoReg_IR2(MemtoReg_IR2),
        .Branch_IR2(Branch_IR2),
        .MemRead_IR2(MemRead_IR2),
        .MemWrite_IR2(MemWrite_IR2),
        .out(out2),
        .zero(g),
        .Result(Result),
        .readData2_IR2(readData2_IR2),
        .instb_IR2(instb_IR2),
        .RegWrite_IR3(RegWrite_IR3),
        .MemtoReg_IR3(MemtoReg_IR3),
        .Branch_IR3(Branch_IR3),
        .MemRead_IR3(MemRead_IR3),
        .MemWrite_IR3(MemWrite_IR3),
        .out_IR3(out_IR3),
        .zero_IR3(zero_IR3),
        .Result_IR3(Result_IR3),
        .readData2_IR3(readData2_IR3),
        .instb_IR3(instb_IR3)
);

Data_Memory DM(
        .Mem_Addr(Result_IR3),
        .Write_Data(readData2_IR3),
        .clk(clk),
        .MemWrite(MemWrite_IR3),
        .MemRead(MemRead_IR3),
        .Read_Data(Read_Data)
);
```



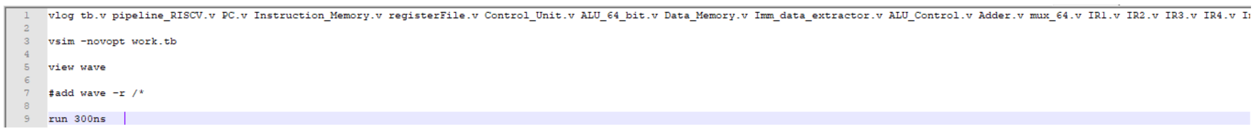
```
IR4 R4(
      .clk(clk),
      .reset(reset),
      .RegWrite_IR3(RegWrite_IR3),
      .MemtoReg_IR3(MemtoReg_IR3),
      .Read_Data(Read_Data),
      .Mem_Addr(Result_IR3),
      .instb_IR3(instb_IR3),
      .RegWrite_IR4(RegWrite_IR4),
      .MemtoReg_IR4(MemtoReg_IR4),
      .Read_Data_IR4(Read_Data_IR4),
      .Mem_Addr_IR4(Mem_Addr_IR4),
      .instb_IR4(instb_IR4)
);

mux_64 mux2(
      .a(Mem_Addr_IR4),
      .b(Read_Data_IR4),
      .sel(MemtoReg_IR4),
      .dataout(muxOut2)
);

endmodule
```

Test bench:



```
module tb(
);

reg clk, reset;

pipeline_RISCV riscv(
    .clk(clk),
    .reset(reset)
);

initial
begin
    reset = 1'b1;
    clk = 1'b1;
    #7 reset = 1'b0;
end

always
#10 clk = ~clk;

endmodule
```

Run.do:



```
vlog tb.v pipeline_RISCV.v PC.v Instruction_Memory.v registerFile.v Control_Unit.v ALU_64_bit.v Data_Memory.v Imm_data_extractor.v ALU_Control.v Adder.v mux_64.v IR1.v IR2.v IR3.v IR4.v I

vsim -novopt work.tb

view wave

#add wave -r /*

run 300ns
```

Implementation:

We had first developed a single cycle RISC V processor in our lab 11. We achieved that by combining all the modules we had been working on throughout this semester. Then we further built on it to develop a 5-stage pipeline RISC V processor which can execute bubble sort. We created a separate module called bgt which would perform functionalities such as branch on greater than and equal to (bge) and branch on less than (blt). The module has a single output (g) which is high when it should branch that is when ((a>=b & instruction is bge) | (a<b & instruction is blt) | (a=b & instruction is beq) otherwise, it will give zero which means the program won't branch. Moreover, we then introduced pipeline registers in between different stages to develop a 5-stage processor. We created individual modules for each of the intermediate register. We then made the connections as shown in fig. 4.49.

Improvement needed:

We still have to perform task 3 that is to detect and handle hazards.