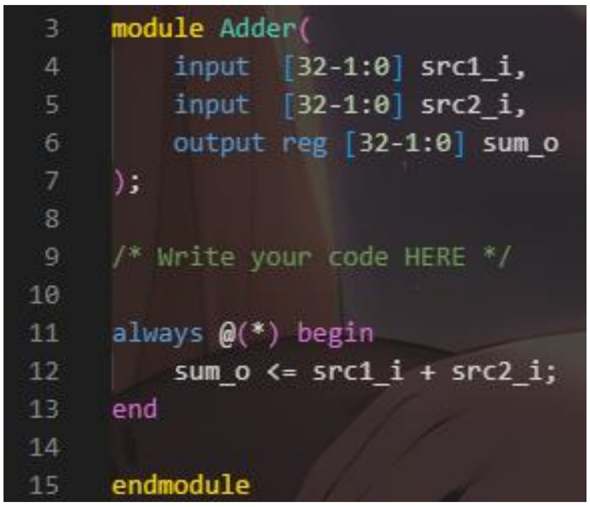
Lab 5-Report

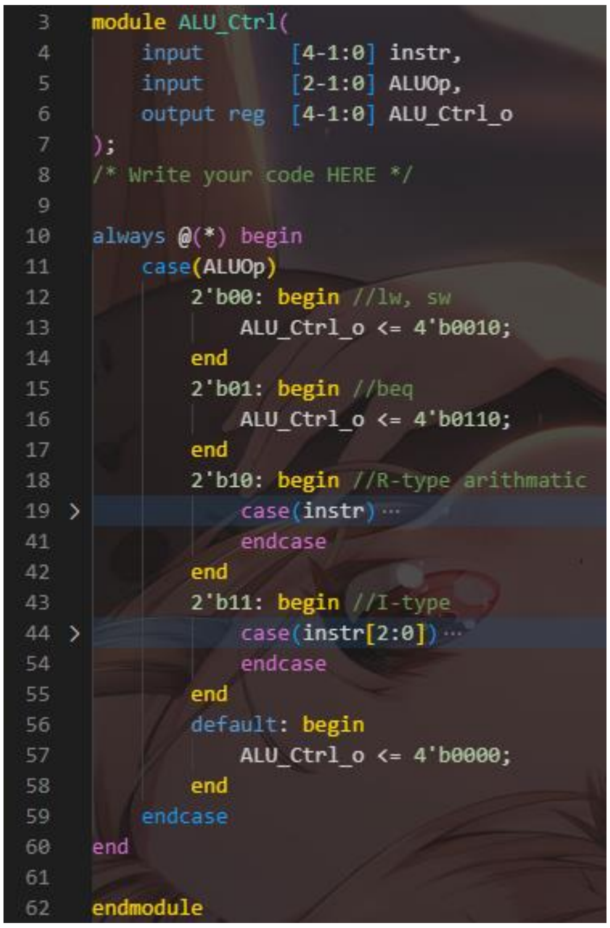
1. Description of descriptions

I. Adder.v: just add.



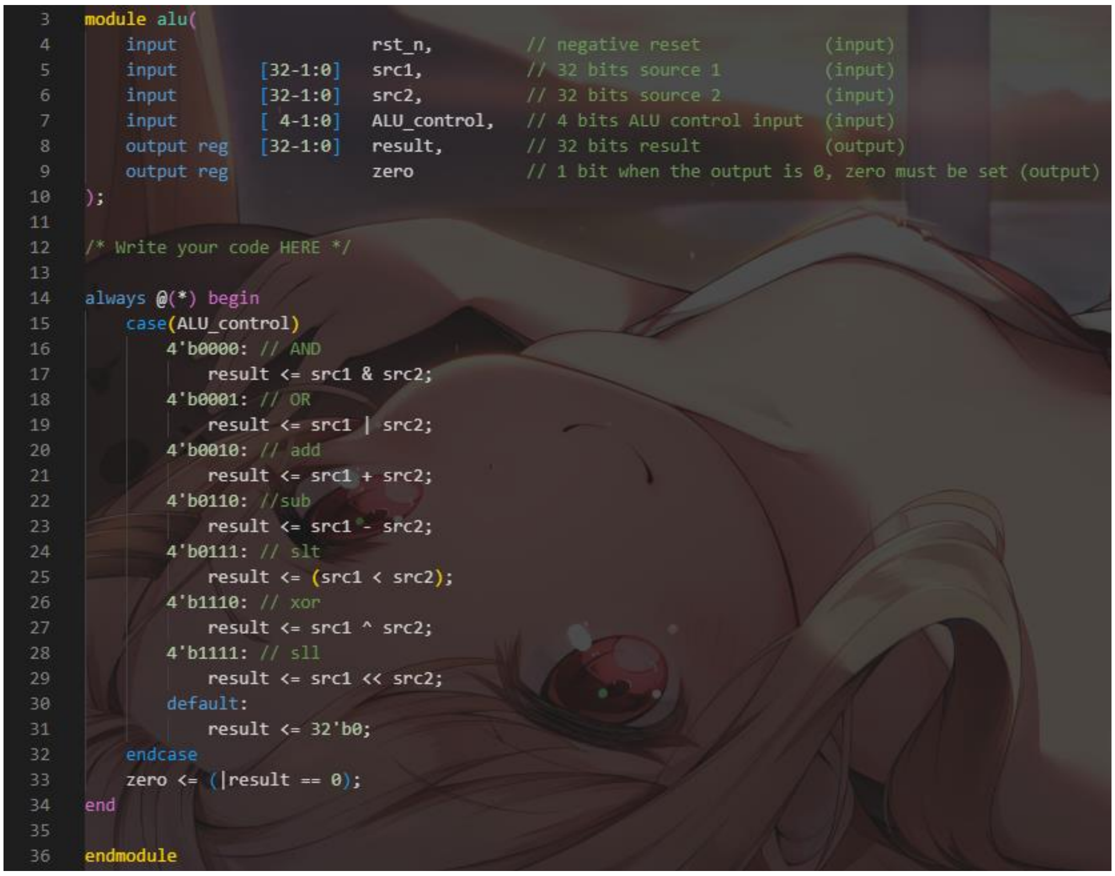
```
module Adder(
    input  [32-1:0] src1_i,
    input  [32-1:0] src2_i,
    output reg [32-1:0] sum_o
);

/* Write your code HERE */

always @(*) begin
    sum_o <= src1_i + src2_i;
end

endmodule
```

II. ALU_Ctrl.v: select output according to ALUOp and instr.



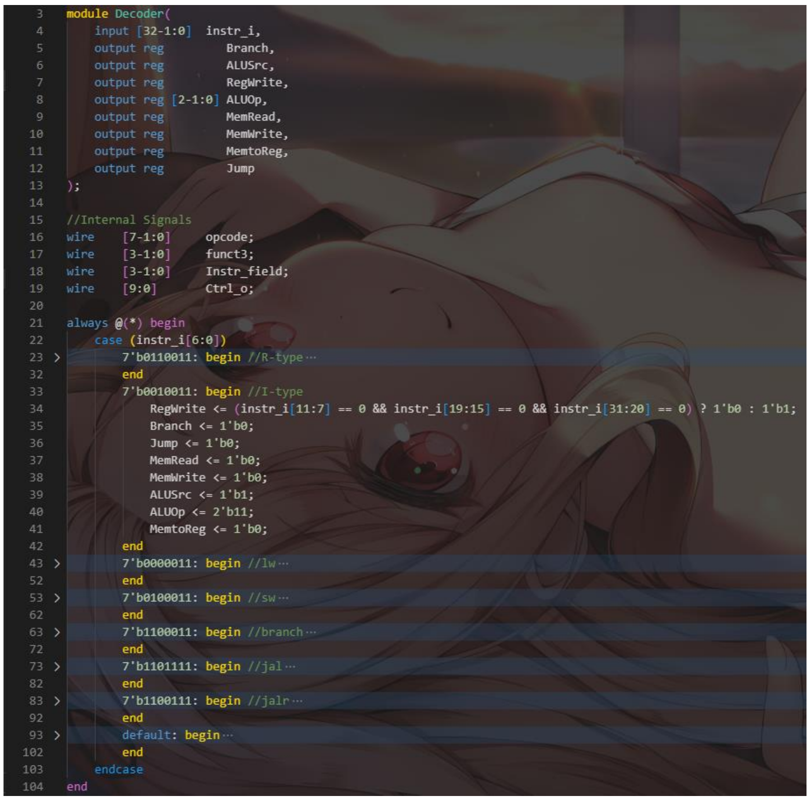
```
module ALU_Ctrl(
    input        [4-1:0] instr,
    input        [2-1:0] ALUOp,
    output reg   [4-1:0] ALU_Ctrl_o
);
/* Write your code HERE */

always @(*) begin
    case(ALUOp)
        2'b00: begin //lw, sw
            ALU_Ctrl_o <= 4'b0010;
        end
        2'b01: begin //beq
            ALU_Ctrl_o <= 4'b0110;
        end
        2'b10: begin //R-type arithmatic
            case(instr) ···
            endcase
        end
        2'b11: begin //I-type
            case(instr[2:0]) ···
            endcase
        end
        default: begin
            ALU_Ctrl_o <= 4'b0000;
        end
    endcase
end

endmodule
```

III. Alu.v: perform calculations according to ALU control signal.



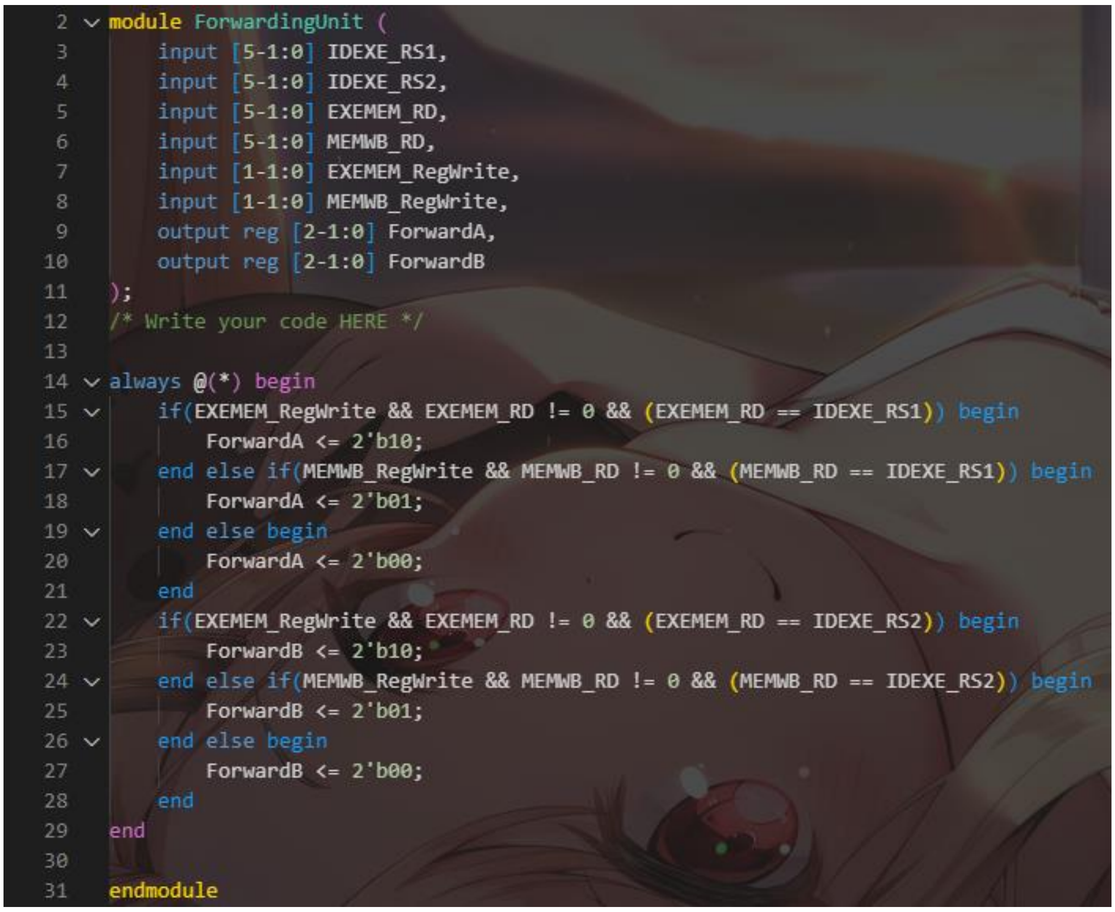
```
module alu(
    input                   rst_n,          // negative reset            (input)
    input        [32-1:0]   src1,           // 32 bits source 1          (input)
    input        [32-1:0]   src2,           // 32 bits source 2          (input)
    input        [ 4-1:0]   ALU_control,    // 4 bits ALU control input  (input)
    output reg   [32-1:0]   result,         // 32 bits result            (output)
    output reg              zero            // 1 bit when the output is 0, zero must be set (output)
);

/* Write your code HERE */

always @(*) begin
    case(ALU_control)
        4'b0000: // AND
            result <= src1 & src2;
        4'b0001: // OR
            result <= src1 | src2;
        4'b0010: // add
            result <= src1 + src2;
        4'b0110: //sub
            result <= src1 - src2;
        4'b0111: // slt
            result <= (src1 < src2);
        4'b1110: // xor
            result <= src1 ^ src2;
        4'b1111: // sll
            result <= src1 << src2;
        default:
            result <= 32'b0;
    endcase
    zero <= (|result == 0);
end

endmodule
```

IV. Decoder.v: output corresponding signal by opcode and detecting NOP at the same time.



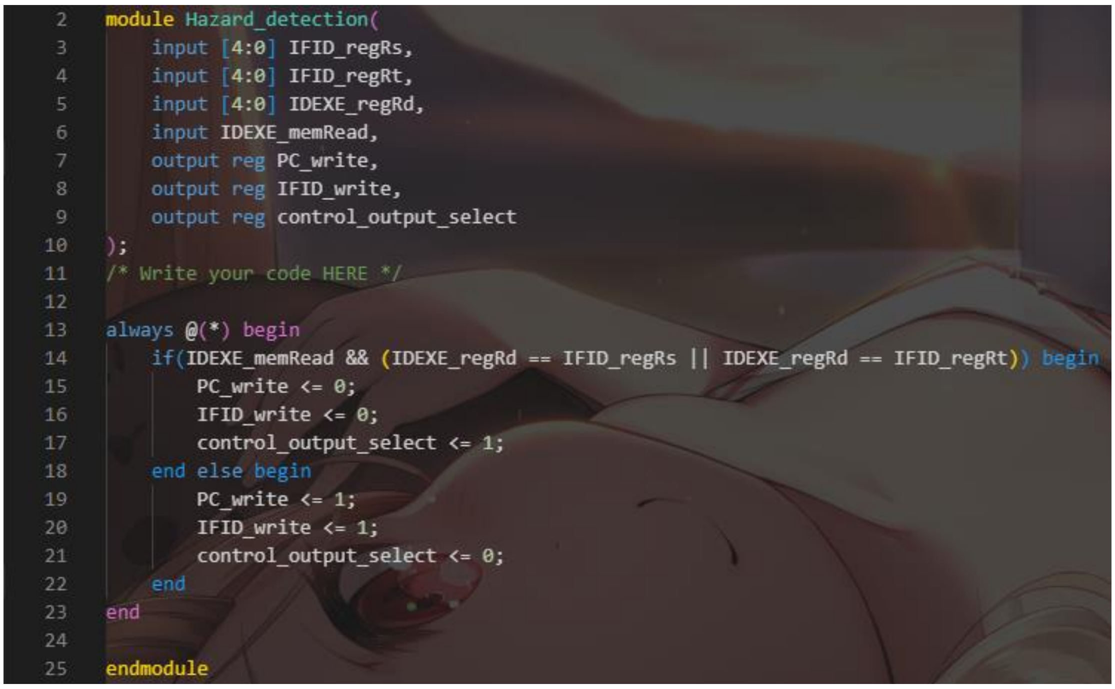
```
module Decoder(
    input [32-1:0]  instr_i,
    output reg         Branch,
    output reg         ALUSrc,
    output reg         RegWrite,
    output reg [2-1:0] ALUOp,
    output reg         MemRead,
    output reg         MemWrite,
    output reg         MemtoReg,
    output reg         Jump
);

//Internal Signals
wire    [7-1:0]     opcode;
wire    [3-1:0]     funct3;
wire    [3-1:0]     Instr_field;
wire    [9:0]       Ctrl_o;

always @(*) begin
    case (instr_i[6:0])
        7'b0110011: begin //R-type···
        end
        7'b0010011: begin //I-type
            RegWrite <= (instr_i[11:7] == 0 && instr_i[19:15] == 0 && instr_i[31:20] == 0) ? 1'b0 : 1'b1;
            Branch <= 1'b0;
            Jump <= 1'b0;
            MemRead <= 1'b0;
            MemWrite <= 1'b0;
            ALUSrc <= 1'b1;
            ALUOp <= 2'b11;
            MemtoReg <= 1'b0;
        end
        7'b0000011: begin //lw···
        end
        7'b0100011: begin //sw···
        end
        7'b1100011: begin //branch···
        end
        7'b1101111: begin //jal···
        end
        7'b1100111: begin //jalr···
        end
        default: begin···
        end
    endcase
end
```

V. Forwarding.v: test EXE forward, else test MEM forward, else no forward.



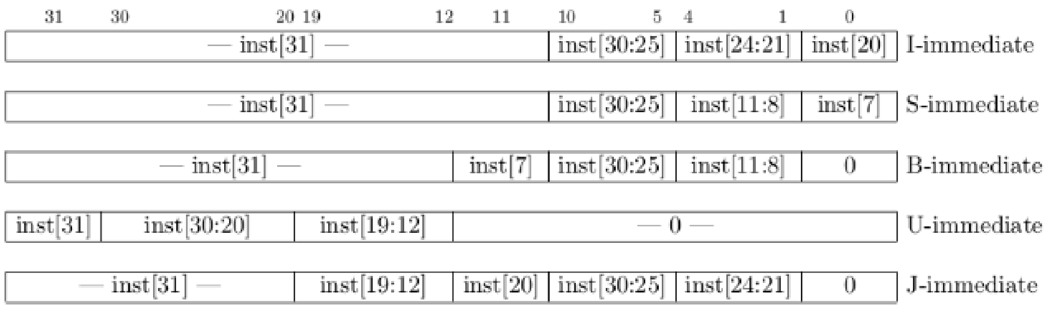
```
module ForwardingUnit (
    input [5-1:0] IDEXE_RS1,
    input [5-1:0] IDEXE_RS2,
    input [5-1:0] EXEMEM_RD,
    input [5-1:0] MEMWB_RD,
    input [1-1:0] EXEMEM_RegWrite,
    input [1-1:0] MEMWB_RegWrite,
    output reg [2-1:0] ForwardA,
    output reg [2-1:0] ForwardB
);
/* Write your code HERE */

always @(*) begin
    if(EXEMEM_RegWrite && EXEMEM_RD != 0 && (EXEMEM_RD == IDEXE_RS1)) begin
        ForwardA <= 2'b10;
    end else if(MEMWB_RegWrite && MEMWB_RD != 0 && (MEMWB_RD == IDEXE_RS1)) begin
        ForwardA <= 2'b01;
    end else begin
        ForwardA <= 2'b00;
    end
    if(EXEMEM_RegWrite && EXEMEM_RD != 0 && (EXEMEM_RD == IDEXE_RS2)) begin
        ForwardB <= 2'b10;
    end else if(MEMWB_RegWrite && MEMWB_RD != 0 && (MEMWB_RD == IDEXE_RS2)) begin
        ForwardB <= 2'b01;
    end else begin
        ForwardB <= 2'b00;
    end
end

endmodule
```

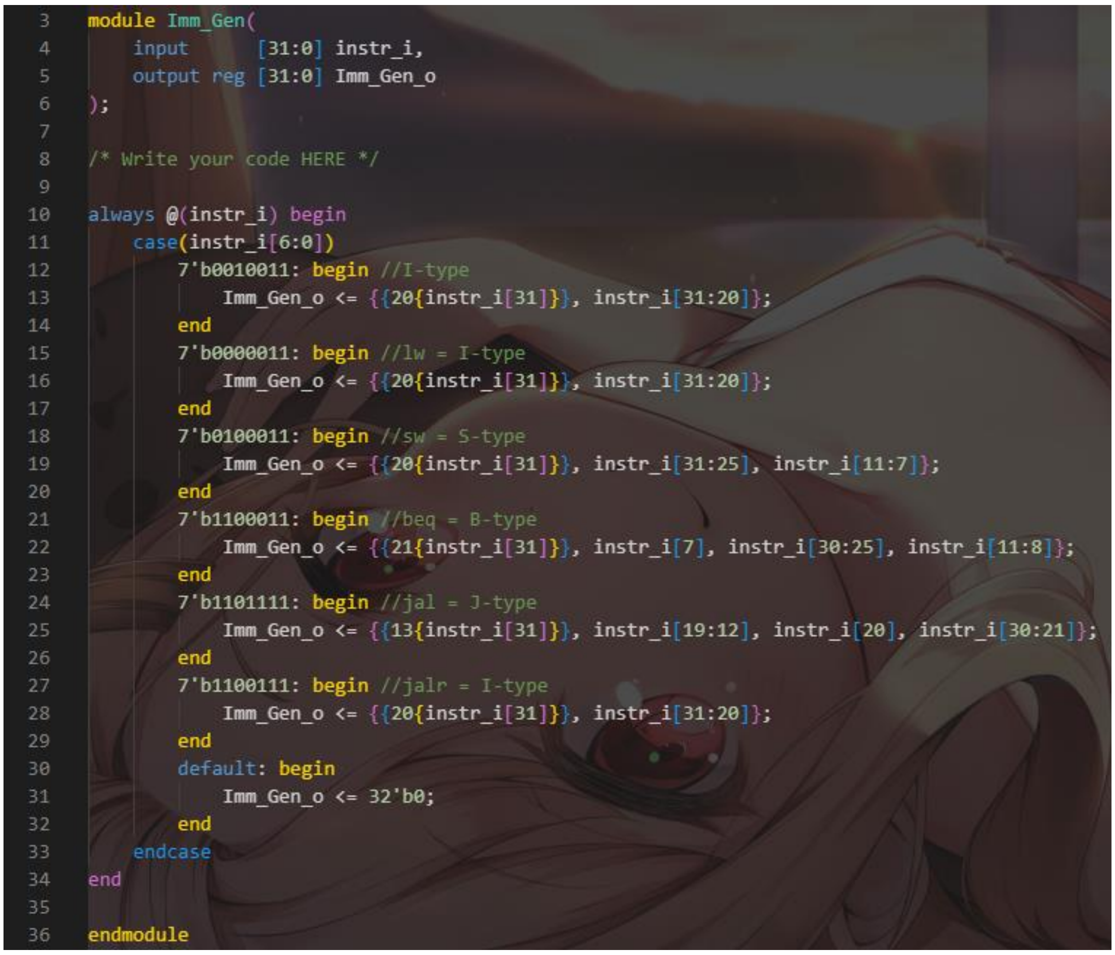
VI. Hazard_detection.v: if EXE is lw and EXE.rd = ID.rs1 or rs2, then disable the write of IFID_reg and PC_write, and set the IDEX_reg control signal to 0's.



```
module Hazard_detection(
    input [4:0] IFID_regRs,
    input [4:0] IFID_regRt,
    input [4:0] IDEXE_regRd,
    input IDEXE_memRead,
    output reg PC_write,
    output reg IFID_write,
    output reg control_output_select
);
/* Write your code HERE */

always @(*) begin
    if(IDEXE_memRead && (IDEXE_regRd == IFID_regRs || IDEXE_regRd == IFID_regRt)) begin
        PC_write <= 0;
        IFID_write <= 0;
        control_output_select <= 1;
    end else begin
        PC_write <= 1;
        IFID_write <= 1;
        control_output_select <= 0;
    end
end

endmodule
```

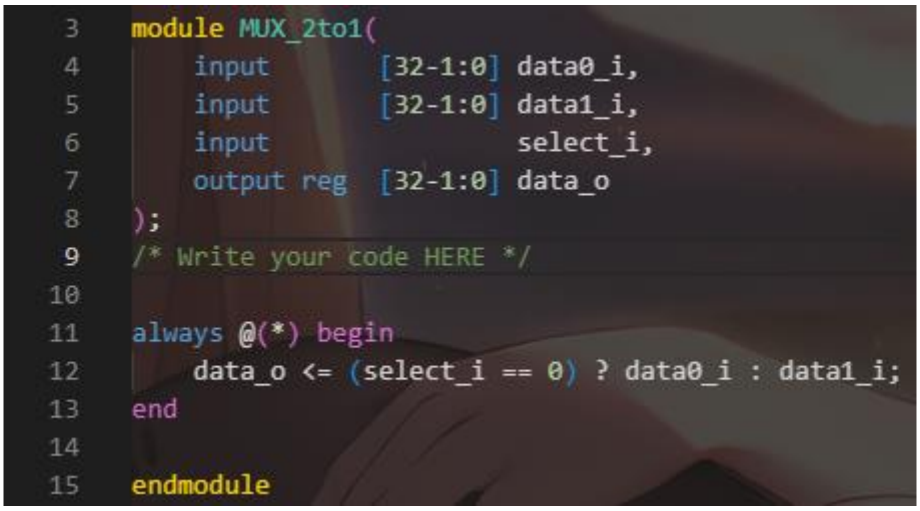
VII. Imm_gen.v: generate the immediate according to the type of instruction and the chart below.


31 30 20 19 12 11 10 5 4 1 0
— inst[31] — | inst[30:25] | inst[24:21] | inst[20] | I-immediate
— inst[31] — | inst[30:25] | inst[11:8] | inst[7] | S-immediate
— inst[31] — | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate
inst[31] | inst[30:20] | inst[19:12] | — 0 — | U-immediate
— inst[31] — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate

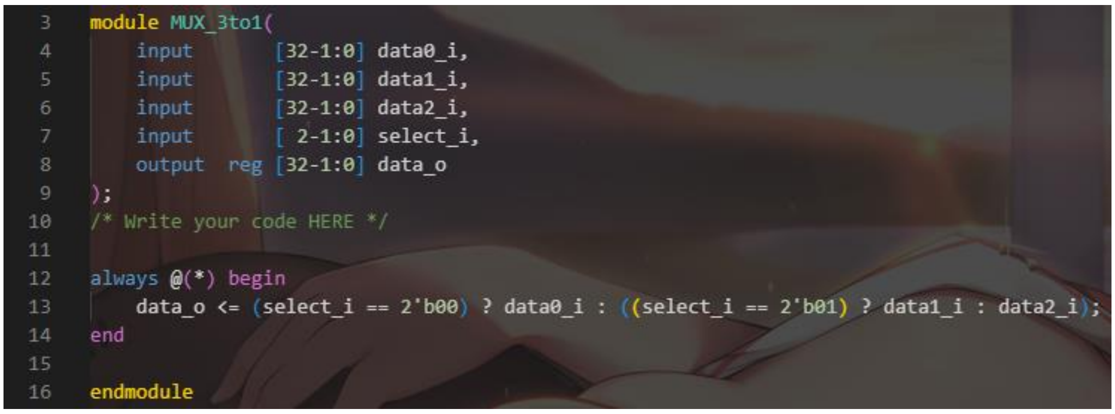


```verilog
module Imm_Gen(
    input       [31:0] instr_i,
    output reg  [31:0] Imm_Gen_o
);

/* Write your code HERE */

always @(instr_i) begin
    case(instr_i[6:0])
        7'b0010011: begin //I-type
            Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31:20]};
        end
        7'b0000011: begin //lw = I-type
            Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31:20]};
        end
        7'b0100011: begin //sw = S-type
            Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31:25], instr_i[11:7]};
        end
        7'b1100011: begin //beq = B-type
            Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[7], instr_i[30:25], instr_i[11:8]};
        end
        7'b1101111: begin //jal = J-type
            Imm_Gen_o <= {{13{instr_i[31]}}, instr_i[19:12], instr_i[20], instr_i[30:21]};
        end
        7'b1100111: begin //jalr = I-type
            Imm_Gen_o <= {{20{instr_i[31]}}, instr_i[31:20]};
        end
        default: begin
            Imm_Gen_o <= 32'b0;
        end
    endcase
end

endmodule
```

VIII. MUX 2 to 1, 3 to 1: select and output.

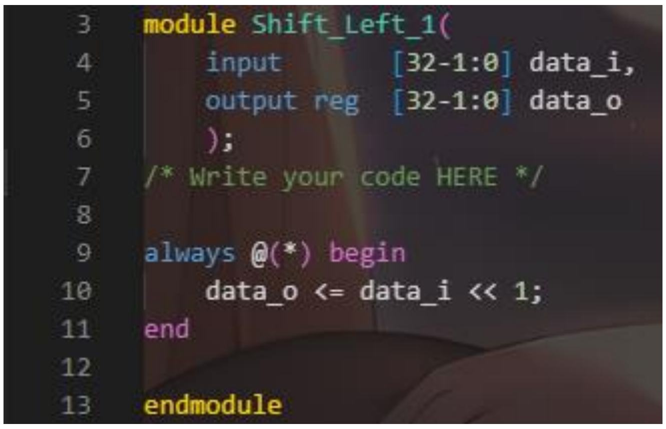


```verilog
module MUX_2to1(
    input          [32-1:0] data0_i,
    input          [32-1:0] data1_i,
    input                   select_i,
    output reg     [32-1:0] data_o
);
/* Write your code HERE */

always @(*) begin
    data_o <= (select_i == 0) ? data0_i : data1_i;
end

endmodule
```



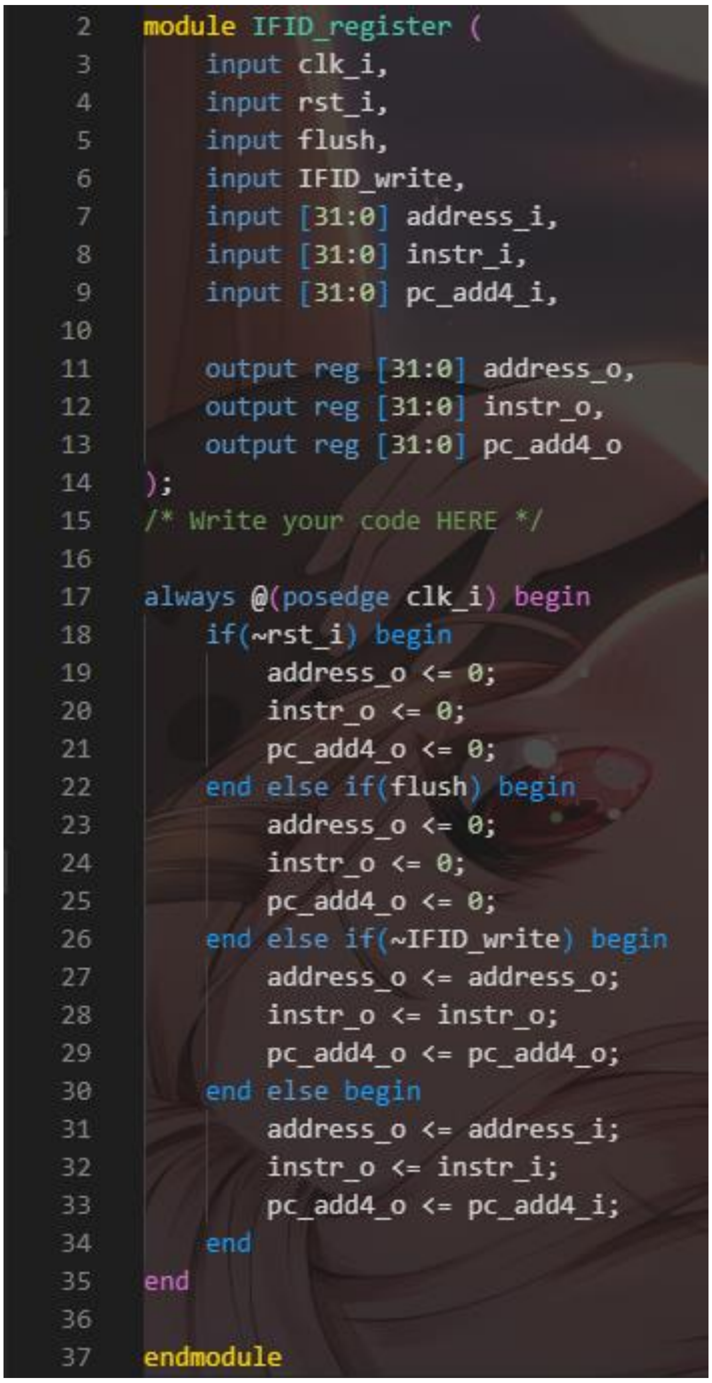
```verilog
module MUX_3to1(
    input        [32-1:0] data0_i,
    input        [32-1:0] data1_i,
    input        [32-1:0] data2_i,
    input        [ 2-1:0] select_i,
    output  reg  [32-1:0] data_o
);
/* Write your code HERE */

always @(*) begin
    data_o <= (select_i == 2'b00) ? data0_i : ((select_i == 2'b01) ? data1_i : data2_i);
end

endmodule
```

IX. Shift_left_1.v: shift left 1 and output.



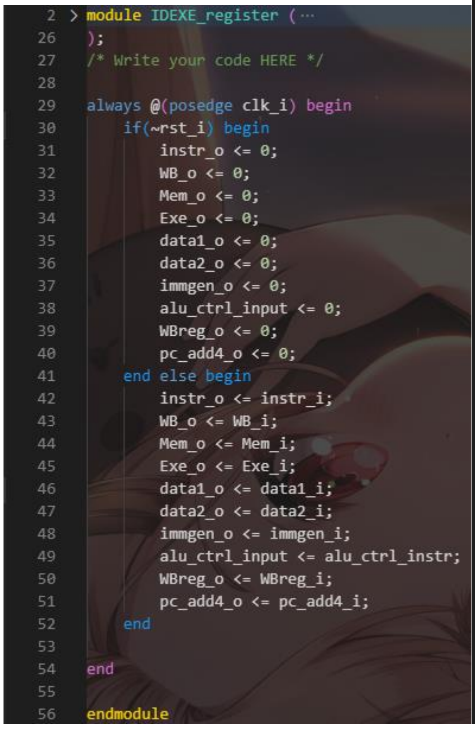
```verilog
module Shift_Left_1(
    input          [32-1:0] data_i,
    output reg     [32-1:0] data_o
    );
/* Write your code HERE */

always @(*) begin
    data_o <= data_i << 1;
end

endmodule
```

X. IFID_register: for every clock, reset or flush if needed. Else if write is disabled, then do not change the outputs. Else output the inputs.

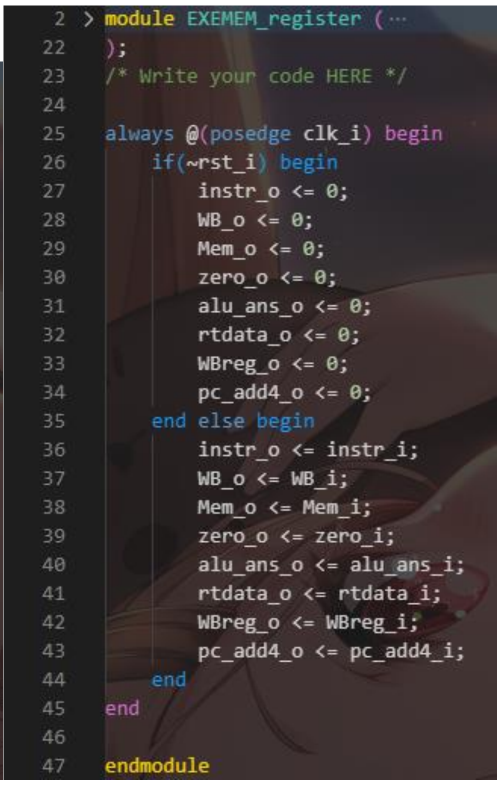


```verilog
module IFID_register (
    input clk_i,
    input rst_i,
    input flush,
    input IFID_write,
    input [31:0] address_i,
    input [31:0] instr_i,
    input [31:0] pc_add4_i,

    output reg [31:0] address_o,
    output reg [31:0] instr_o,
    output reg [31:0] pc_add4_o
);
/* Write your code HERE */

always @(posedge clk_i) begin
    if(~rst_i) begin
        address_o <= 0;
        instr_o <= 0;
        pc_add4_o <= 0;
    end else if(flush) begin
        address_o <= 0;
        instr_o <= 0;
        pc_add4_o <= 0;
    end else if(~IFID_write) begin
        address_o <= address_o;
        instr_o <= instr_o;
        pc_add4_o <= pc_add4_o;
    end else begin
        address_o <= address_i;
        instr_o <= instr_i;
        pc_add4_o <= pc_add4_i;
    end
end

endmodule
```

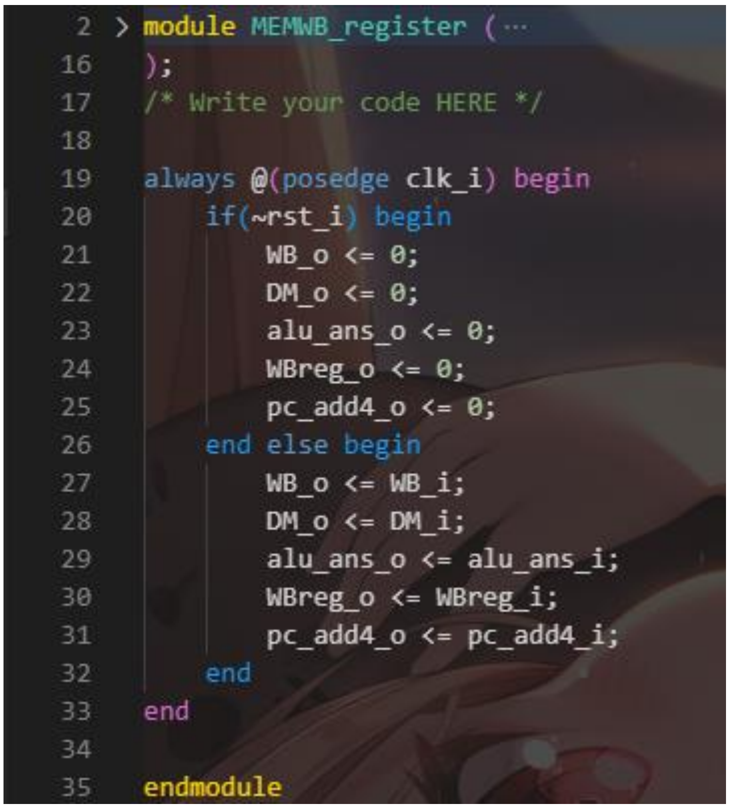
XI. Other registers: For every clock reset the register if ~rst_i, else output the inputs.



```verilog
module IDEXE_register (…
);
/* Write your code HERE */

always @(posedge clk_i) begin
    if(~rst_i) begin
        instr_o <= 0;
        WB_o <= 0;
        Mem_o <= 0;
        Exe_o <= 0;
        data1_o <= 0;
        data2_o <= 0;
        immgen_o <= 0;
        alu_ctrl_input <= 0;
        WBreg_o <= 0;
        pc_add4_o <= 0;
    end else begin
        instr_o <= instr_i;
        WB_o <= WB_i;
        Mem_o <= Mem_i;
        Exe_o <= Exe_i;
        data1_o <= data1_i;
        data2_o <= data2_i;
        immgen_o <= immgen_i;
        alu_ctrl_input <= alu_ctrl_instr;
        WBreg_o <= WBreg_i;
        pc_add4_o <= pc_add4_i;
    end

end

endmodule
```



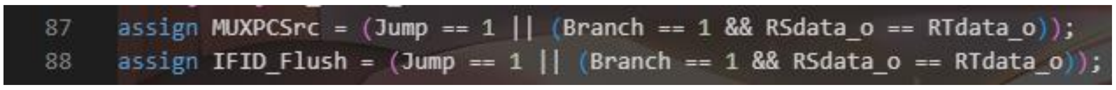
```verilog
module EXEMEM_register (…
);
/* Write your code HERE */

always @(posedge clk_i) begin
    if(~rst_i) begin
        instr_o <= 0;
        WB_o <= 0;
        Mem_o <= 0;
        zero_o <= 0;
        alu_ans_o <= 0;
        rtdata_o <= 0;
        WBreg_o <= 0;
        pc_add4_o <= 0;
    end else begin
        instr_o <= instr_i;
        WB_o <= WB_i;
        Mem_o <= Mem_i;
        zero_o <= zero_i;
        alu_ans_o <= alu_ans_i;
        rtdata_o <= rtdata_i;
        WBreg_o <= WBreg_i;
        pc_add4_o <= pc_add4_i;
    end
end

endmodule
```



```verilog
module MEMWB_register (…
);
/* Write your code HERE */

always @(posedge clk_i) begin
    if(~rst_i) begin
        WB_o <= 0;
        DM_o <= 0;
        alu_ans_o <= 0;
        WBreg_o <= 0;
        pc_add4_o <= 0;
    end else begin
        WB_o <= WB_i;
        DM_o <= DM_i;
        alu_ans_o <= alu_ans_i;
        WBreg_o <= WBreg_i;
        pc_add4_o <= pc_add4_i;
    end
end

endmodule
```

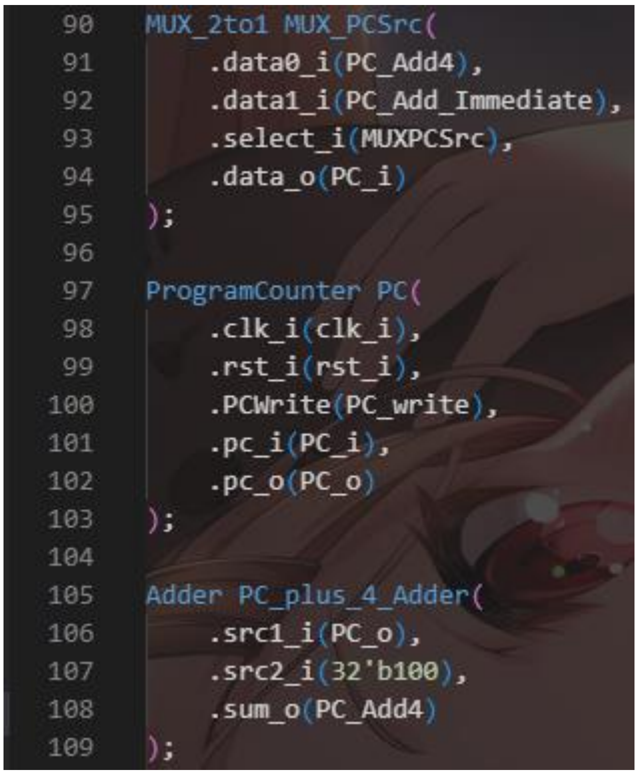
XII. Pipeline_CPU.v

i. Select PC src and flush IFID when jump or branch taken.



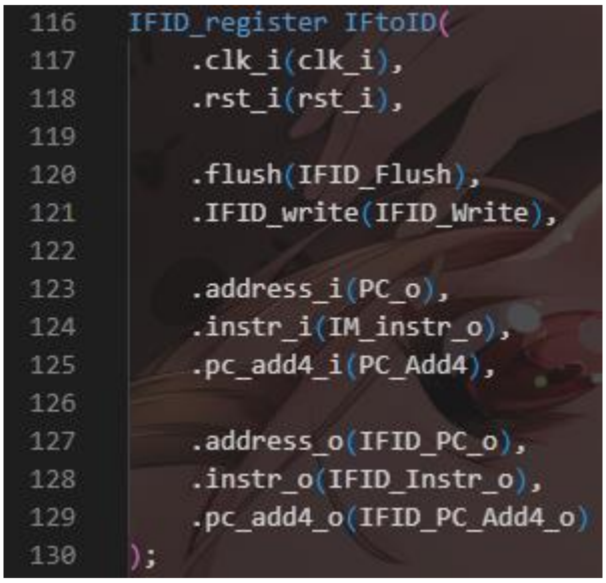
```verilog
assign MUXPCSrc = (Jump == 1 || (Branch == 1 && RSdata_o == RTdata_o));
assign IFID_Flush = (Jump == 1 || (Branch == 1 && RSdata_o == RTdata_o));
```

ii. PC part: add 4 and write back each cycle.



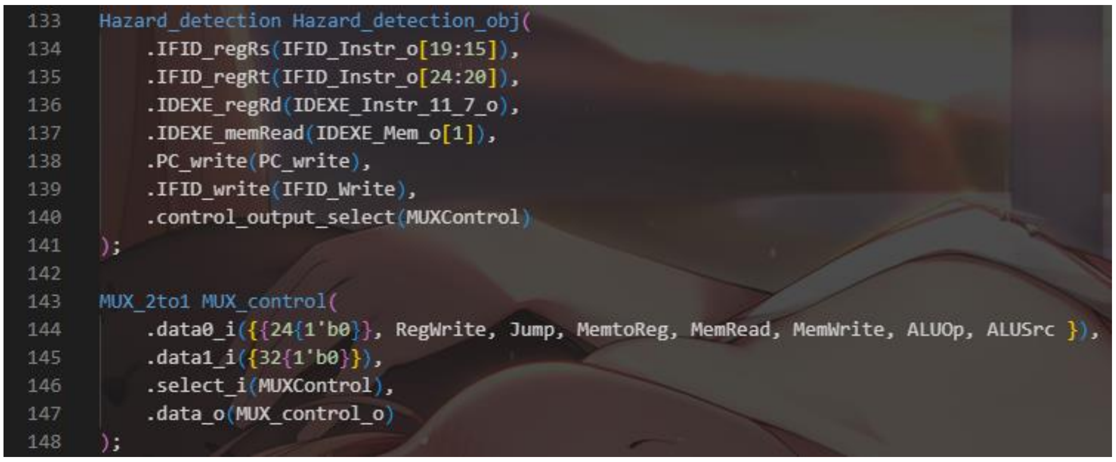
```
MUX_2to1 MUX_PCSrc(
    .data0_i(PC_Add4),
    .data1_i(PC_Add_Immediate),
    .select_i(MUXPCSrc),
    .data_o(PC_i)
);

ProgramCounter PC(
    .clk_i(clk_i),
    .rst_i(rst_i),
    .PCWrite(PC_write),
    .pc_i(PC_i),
    .pc_o(PC_o)
);

Adder PC_plus_4_Adder(
    .src1_i(PC_o),
    .src2_i(32'b100),
    .sum_o(PC_Add4)
);
```

iii. Update the IFID pipeline register.



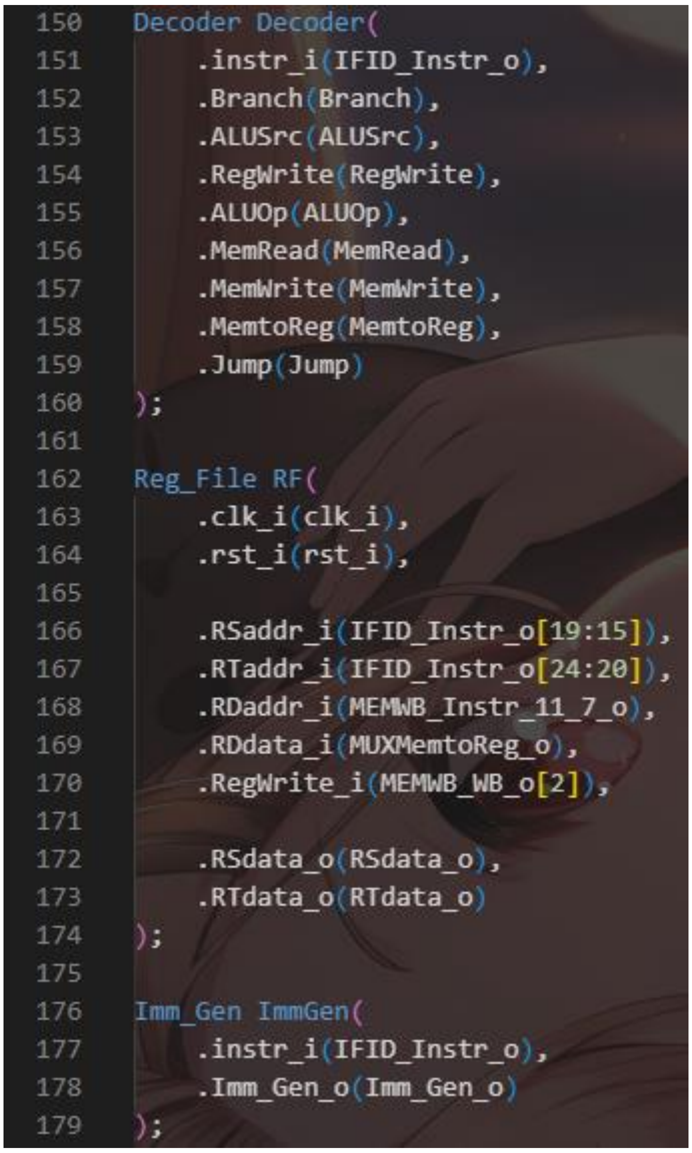
```
IFID_register IFtoID(
    .clk_i(clk_i),
    .rst_i(rst_i),

    .flush(IFID_Flush),
    .IFID_write(IFID_Write),

    .address_i(PC_o),
    .instr_i(IM_instr_o),
    .pc_add4_i(PC_Add4),

    .address_o(IFID_PC_o),
    .instr_o(IFID_Instr_o),
    .pc_add4_o(IFID_PC_Add4_o)
);
```

iv. Hazard detection: if hazard detected, then pause PC write and IFID write, and flush control signals.



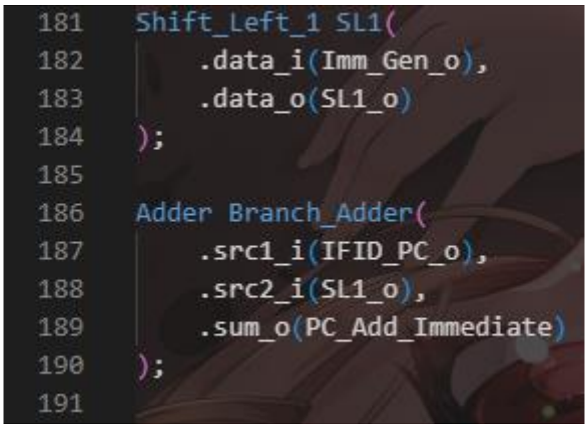
```
Hazard_detection Hazard_detection_obj(
    .IFID_regRs(IFID_Instr_o[19:15]),
    .IFID_regRt(IFID_Instr_o[24:20]),
    .IDEXE_regRd(IDEXE_Instr_11_7_o),
    .IDEXE_memRead(IDEXE_Mem_o[1]),
    .PC_write(PC_write),
    .IFID_write(IFID_Write),
    .control_output_select(MUXControl)
);

MUX_2to1 MUX_control(
    .data0_i({{24{1'b0}}, RegWrite, Jump, MemtoReg, MemRead, MemWrite, ALUOp, ALUSrc }),
    .data1_i({32{1'b0}}),
    .select_i(MUXControl),
    .data_o(MUX_control_o)
);
```

v. Decode the instruction, read/write the register values, and get the immediate value.



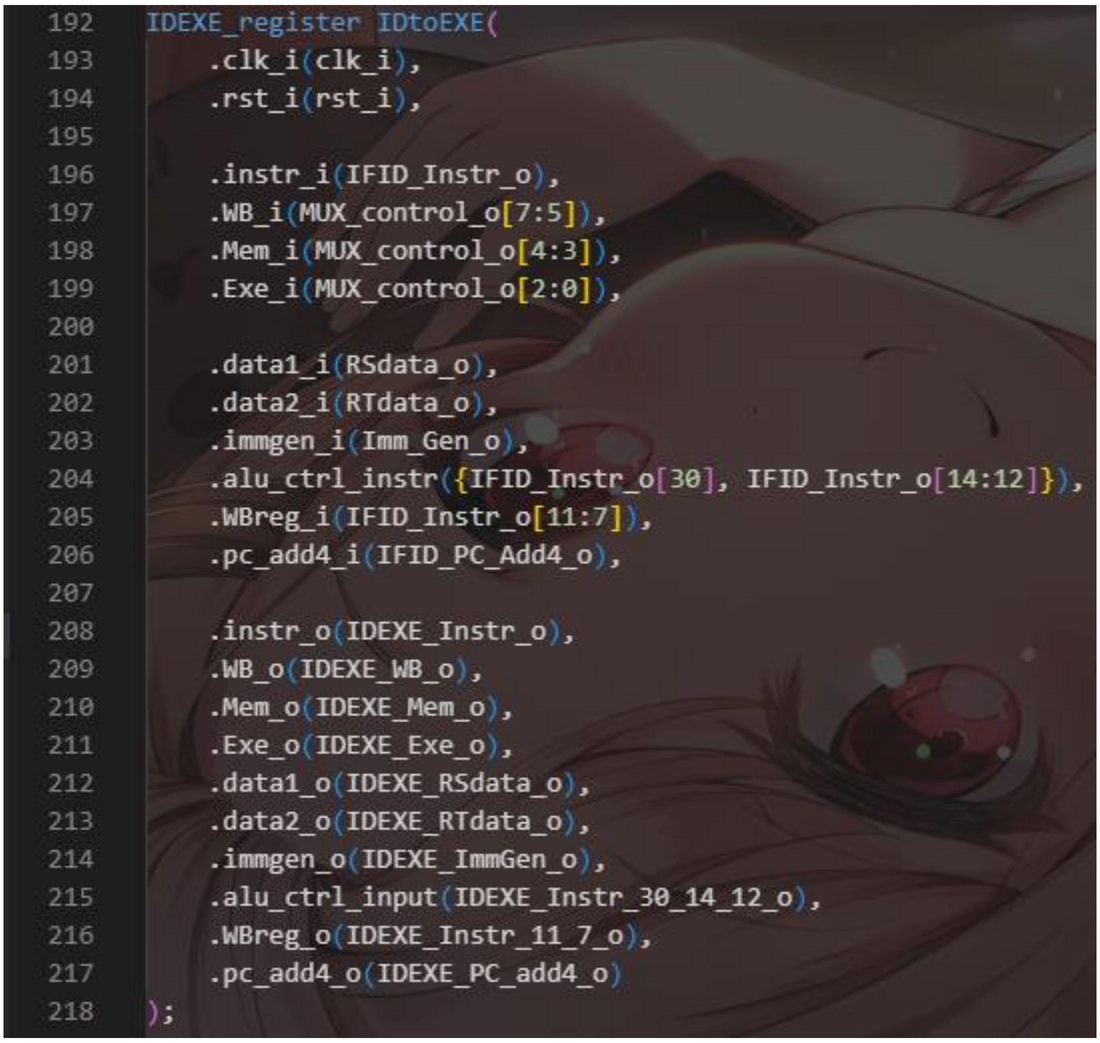
```
Decoder Decoder(
    .instr_i(IFID_Instr_o),
    .Branch(Branch),
    .ALUSrc(ALUSrc),
    .RegWrite(RegWrite),
    .ALUOp(ALUOp),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .MemtoReg(MemtoReg),
    .Jump(Jump)
);

Reg_File RF(
    .clk_i(clk_i),
    .rst_i(rst_i),

    .RSaddr_i(IFID_Instr_o[19:15]),
    .RTaddr_i(IFID_Instr_o[24:20]),
    .RDaddr_i(MEMWB_Instr_11_7_o),
    .RDdata_i(MUXMemtoReg_o),
    .RegWrite_i(MEMWB_WB_o[2]),

    .RSdata_o(RSdata_o),
    .RTdata_o(RTdata_o)
);

Imm_Gen ImmGen(
    .instr_i(IFID_Instr_o),
    .Imm_Gen_o(Imm_Gen_o)
);
```

vi. Calculate the branch/jump destination and feed back to PCMux.



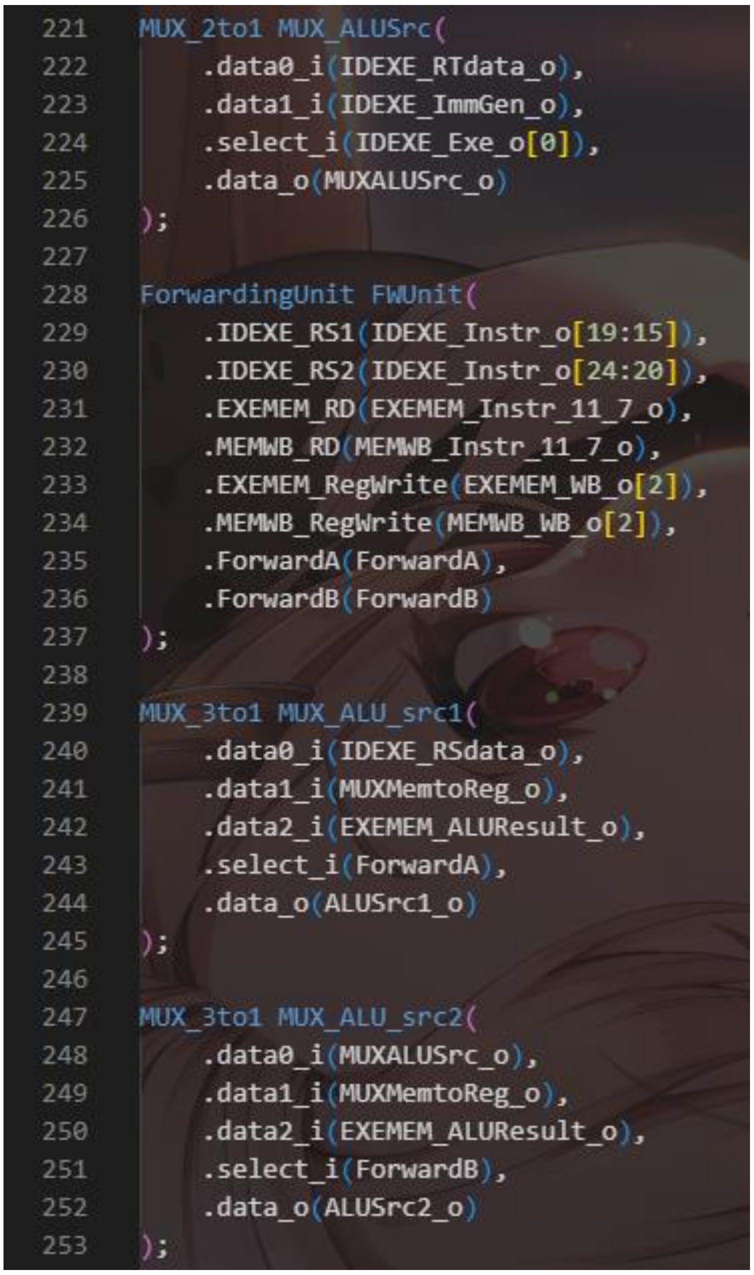
```
Shift_Left_1 SL1(
    .data_i(Imm_Gen_o),
    .data_o(SL1_o)
);

Adder Branch_Adder(
    .src1_i(IFID_PC_o),
    .src2_i(SL1_o),
    .sum_o(PC_Add_Immediate)
);

```

vii. Update the IDEXE register.



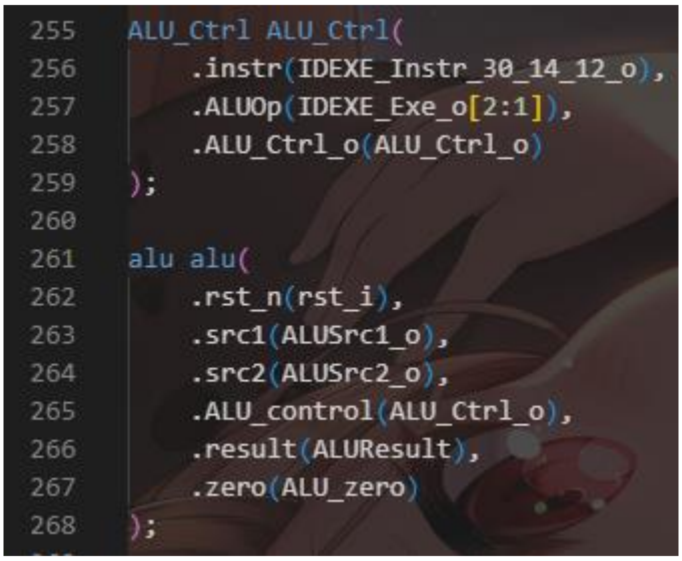
```
IDEXE_register IDtoEXE(
    .clk_i(clk_i),
    .rst_i(rst_i),

    .instr_i(IFID_Instr_o),
    .WB_i(MUX_control_o[7:5]),
    .Mem_i(MUX_control_o[4:3]),
    .Exe_i(MUX_control_o[2:0]),

    .data1_i(RSdata_o),
    .data2_i(RTdata_o),
    .immgen_i(Imm_Gen_o),
    .alu_ctrl_instr({IFID_Instr_o[30], IFID_Instr_o[14:12]}),
    .WBreg_i(IFID_Instr_o[11:7]),
    .pc_add4_i(IFID_PC_Add4_o),

    .instr_o(IDEXE_Instr_o),
    .WB_o(IDEXE_WB_o),
    .Mem_o(IDEXE_Mem_o),
    .Exe_o(IDEXE_Exe_o),
    .data1_o(IDEXE_RSdata_o),
    .data2_o(IDEXE_RTdata_o),
    .immgen_o(IDEXE_ImmGen_o),
    .alu_ctrl_input(IDEXE_Instr_30_14_12_o),
    .WBreg_o(IDEXE_Instr_11_7_o),
    .pc_add4_o(IDEXE_PC_add4_o)
);
```

viii. Get alu source from RSdata and forwarding unit, RTdata, immediate, and forwarding unit. Get alu control signal form ALU_Ctrl.



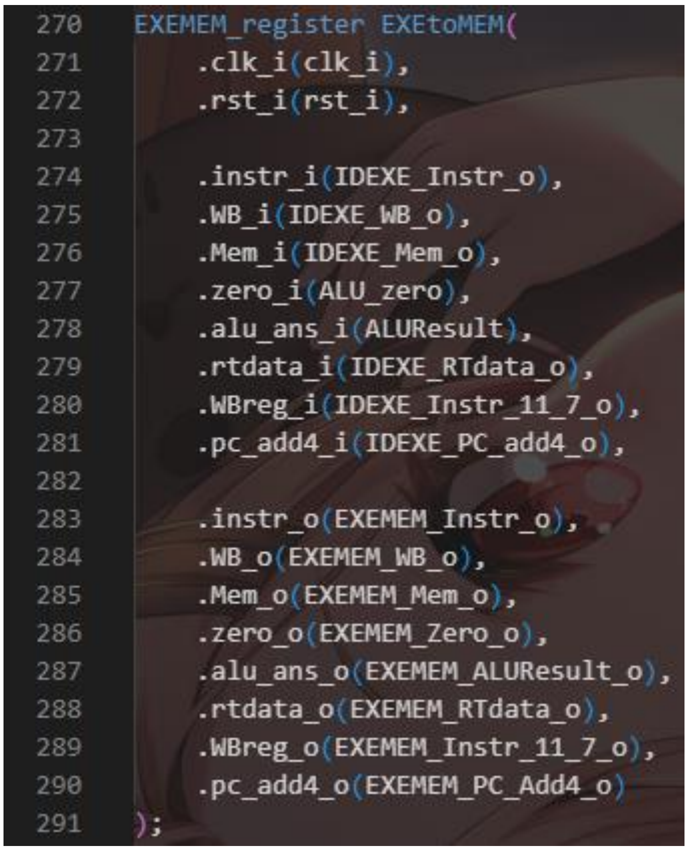
```
MUX_2to1 MUX_ALUSrc(
    .data0_i(IDEXE_RTdata_o),
    .data1_i(IDEXE_ImmGen_o),
    .select_i(IDEXE_Exe_o[0]),
    .data_o(MUXALUSrc_o)
);

ForwardingUnit FWUnit(
    .IDEXE_RS1(IDEXE_Instr_o[19:15]),
    .IDEXE_RS2(IDEXE_Instr_o[24:20]),
    .EXEMEM_RD(EXEMEM_Instr_11_7_o),
    .MEMWB_RD(MEMWB_Instr_11_7_o),
    .EXEMEM_RegWrite(EXEMEM_WB_o[2]),
    .MEMWB_RegWrite(MEMWB_WB_o[2]),
    .ForwardA(ForwardA),
    .ForwardB(ForwardB)
);

MUX_3to1 MUX_ALU_src1(
    .data0_i(IDEXE_RSdata_o),
    .data1_i(MUXMemtoReg_o),
    .data2_i(EXEMEM_ALUResult_o),
    .select_i(ForwardA),
    .data_o(ALUSrc1_o)
);

MUX_3to1 MUX_ALU_src2(
    .data0_i(MUXALUSrc_o),
    .data1_i(MUXMemtoReg_o),
    .data2_i(EXEMEM_ALUResult_o),
    .select_i(ForwardB),
    .data_o(ALUSrc2_o)
);
```

ix. Get alu control signals and send them into alu.



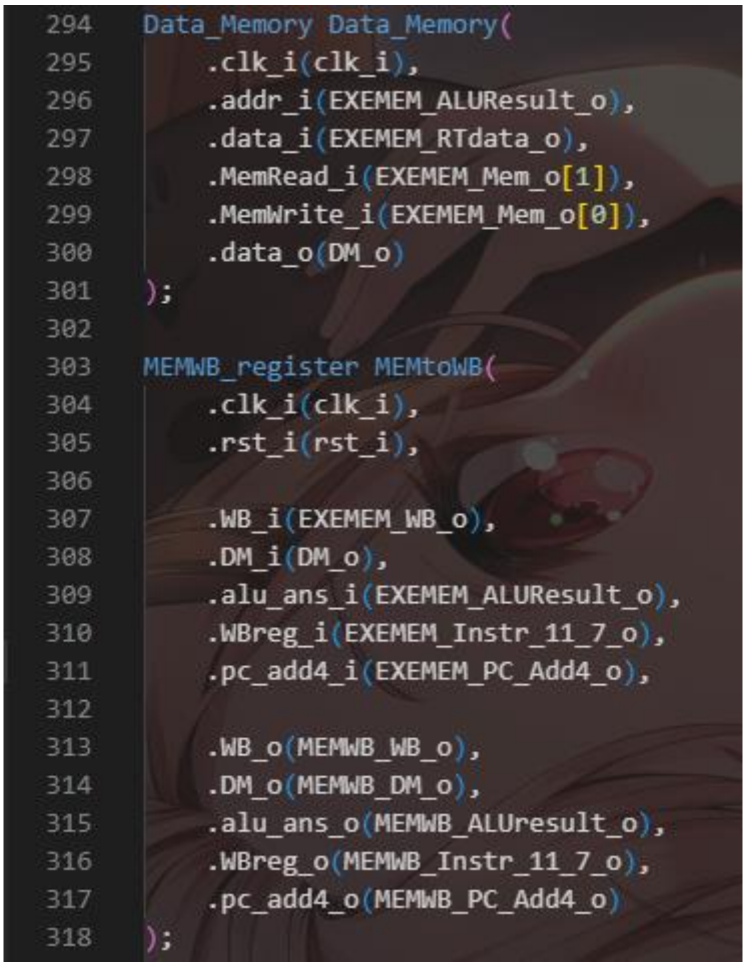
```
ALU_Ctrl ALU_Ctrl(
    .instr(IDEXE_Instr_30_14_12_o),
    .ALUOp(IDEXE_Exe_o[2:1]),
    .ALU_Ctrl_o(ALU_Ctrl_o)
);

alu alu(
    .rst_n(rst_i),
    .src1(ALUSrc1_o),
    .src2(ALUSrc2_o),
    .ALU_control(ALU_Ctrl_o),
    .result(ALUResult),
    .zero(ALU_zero)
);
```

x. Update EXEMEM register.



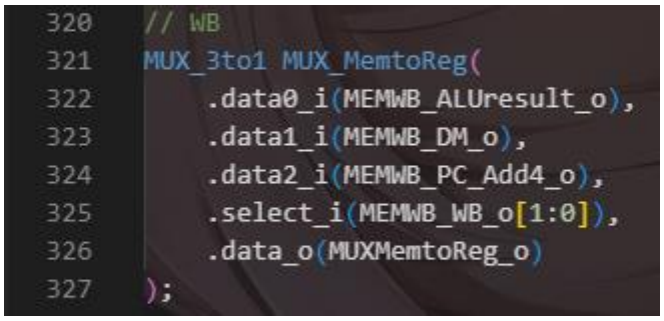
```
EXEMEM_register EXEtoMEM(
    .clk_i(clk_i),
    .rst_i(rst_i),

    .instr_i(IDEXE_Instr_o),
    .WB_i(IDEXE_WB_o),
    .Mem_i(IDEXE_Mem_o),
    .zero_i(ALU_zero),
    .alu_ans_i(ALUResult),
    .rtdata_i(IDEXE_RTdata_o),
    .WBreg_i(IDEXE_Instr_11_7_o),
    .pc_add4_i(IDEXE_PC_add4_o),

    .instr_o(EXEMEM_Instr_o),
    .WB_o(EXEMEM_WB_o),
    .Mem_o(EXEMEM_Mem_o),
    .zero_o(EXEMEM_Zero_o),
    .alu_ans_o(EXEMEM_ALUResult_o),
    .rtdata_o(EXEMEM_RTdata_o),
    .WBreg_o(EXEMEM_Instr_11_7_o),
    .pc_add4_o(EXEMEM_PC_Add4_o)
);
```

xi. Read/write data memory then update MEMWB register.



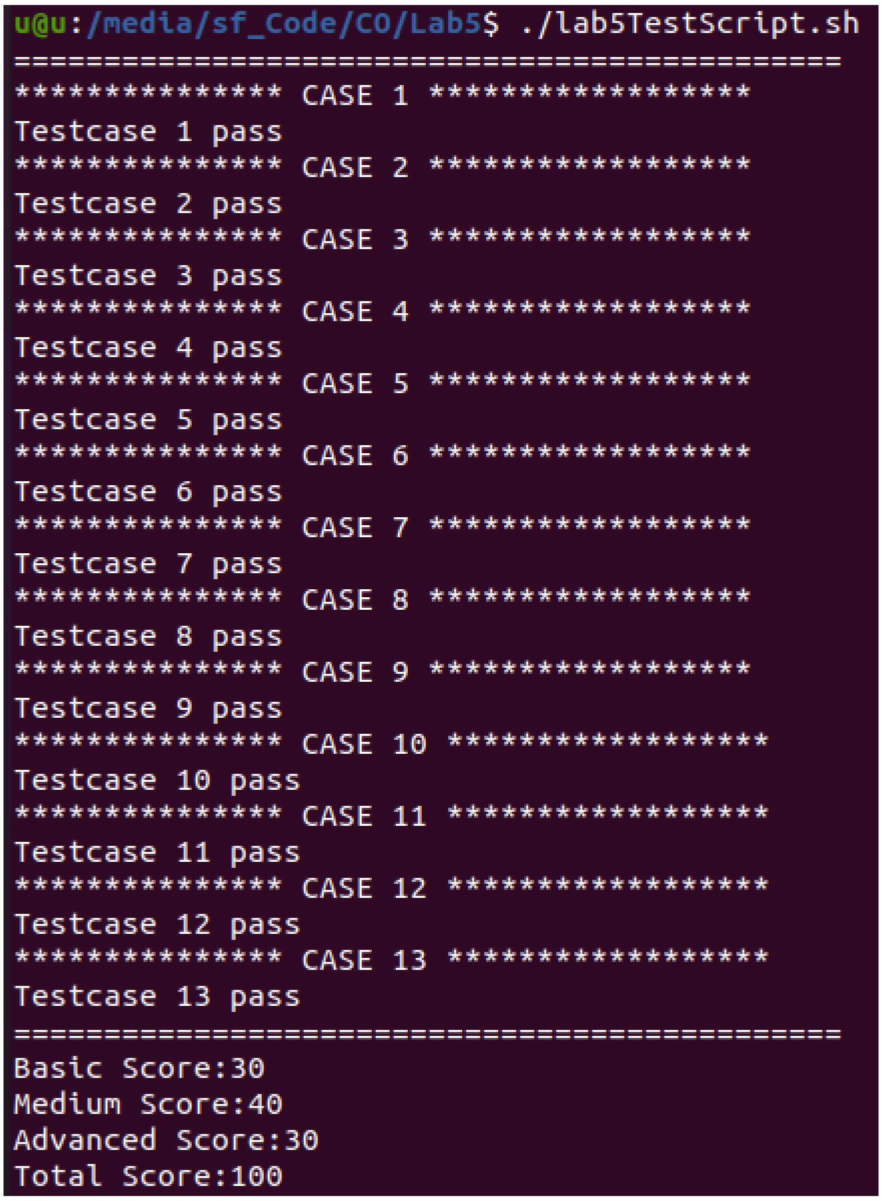
```
Data_Memory Data_Memory(
    .clk_i(clk_i),
    .addr_i(EXEMEM_ALUResult_o),
    .data_i(EXEMEM_RTdata_o),
    .MemRead_i(EXEMEM_Mem_o[1]),
    .MemWrite_i(EXEMEM_Mem_o[0]),
    .data_o(DM_o)
);

MEMWB_register MEMtoWB(
    .clk_i(clk_i),
    .rst_i(rst_i),

    .WB_i(EXEMEM_WB_o),
    .DM_i(DM_o),
    .alu_ans_i(EXEMEM_ALUResult_o),
    .WBreg_i(EXEMEM_Instr_11_7_o),
    .pc_add4_i(EXEMEM_PC_Add4_o),

    .WB_o(MEMWB_WB_o),
    .DM_o(MEMWB_DM_o),
    .alu_ans_o(MEMWB_ALUresult_o),
    .WBreg_o(MEMWB_Instr_11_7_o),
    .pc_add4_o(MEMWB_PC_Add4_o)
);
```

xii. Select the data to write in register file. ALU result by default, if lw then writeback the memory output, if jump taken then writeback pc+4.



```
// WB
MUX_3to1 MUX_MemtoReg(
    .data0_i(MEMWB_ALUresult_o),
    .data1_i(MEMWB_DM_o),
    .data2_i(MEMWB_PC_Add4_o),
    .select_i(MEMWB_WB_o[1:0]),
    .data_o(MUXMemtoReg_o)
);
```

2. Implementation result



```
u@u:/media/sf_Code/CO/Lab5$ ./lab5TestScript.sh
==============================================
*************** CASE 1 ******************
Testcase 1 pass
*************** CASE 2 ******************
Testcase 2 pass
*************** CASE 3 ******************
Testcase 3 pass
*************** CASE 4 ******************
Testcase 4 pass
*************** CASE 5 ******************
Testcase 5 pass
*************** CASE 6 ******************
Testcase 6 pass
*************** CASE 7 ******************
Testcase 7 pass
*************** CASE 8 ******************
Testcase 8 pass
*************** CASE 9 ******************
Testcase 9 pass
*************** CASE 10 ******************
Testcase 10 pass
*************** CASE 11 ******************
Testcase 11 pass
*************** CASE 12 ******************
Testcase 12 pass
*************** CASE 13 ******************
Testcase 13 pass
==============================================
Basic Score:30
Medium Score:40
Advanced Score:30
Total Score:100
```

3. Problem encountered

- I found that some control signals are X and cause some modules can't work properly.
  - I realized I didn't reset the pipeline registers at the beginning. After I corrected that, the result becomes correct.