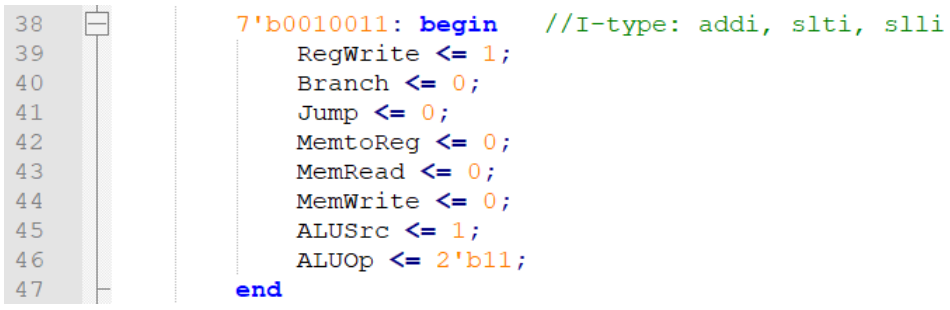
LAB 5 : 5-Stage Pipeline Processor

Group 17 109550135 范恩宇 109550156 曾偉杰

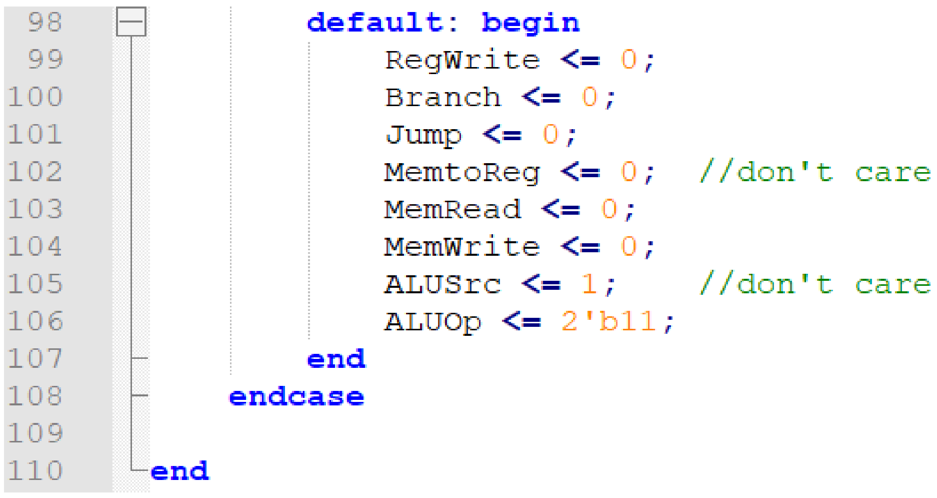
Part I. Detailed description of the implementation :

1. Decoder.v:

Basically having the same structure as our previous one . However , we don't need two "write_back" and only need one "ALUSrc" this time . Besides , since NOP is encoded as "ADDI x0,x0,0" according to the Lab5-slide , we set only the "ALUOp" of I-type commands as the same with the one of NOP this time .



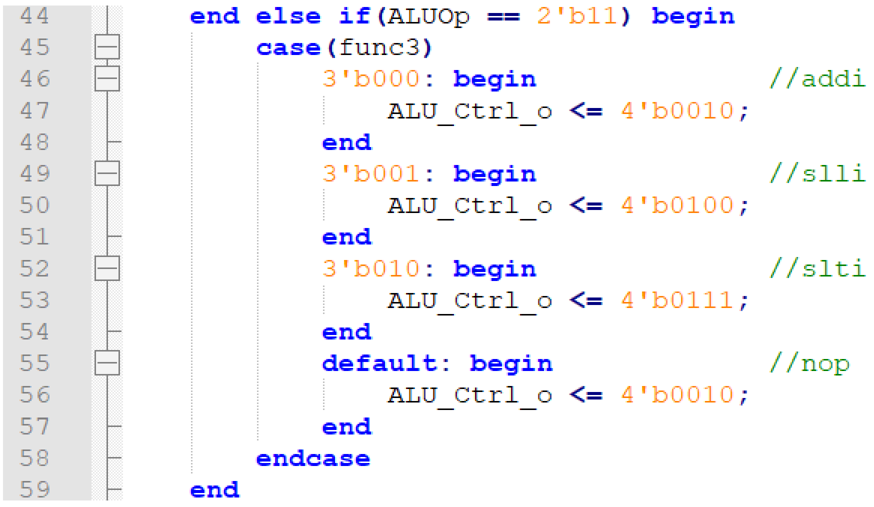
```
        7'b0010011: begin   //I-type: addi, slti, slli
            RegWrite <= 1;
            Branch <= 0;
            Jump <= 0;
            MemtoReg <= 0;
            MemRead <= 0;
            MemWrite <= 0;
            ALUSrc <= 1;
            ALUOp <= 2'b11;
        end
```



```
            default: begin
                RegWrite <= 0;
                Branch <= 0;
                Jump <= 0;
                MemtoReg <= 0;  //don't care
                MemRead <= 0;
                MemWrite <= 0;
                ALUSrc <= 1;    //don't care
                ALUOp <= 2'b11;
            end
        endcase

end
```

2. ALU_Ctrl.v :

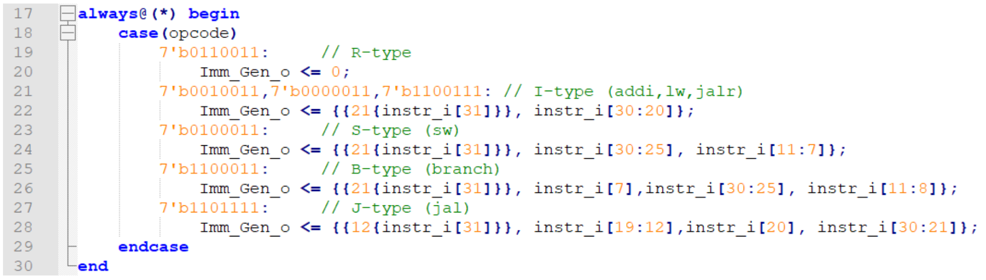
Also having the same structure as our previous one , but we make it able to deal with more commands as the slide asked us to do . In addition , I-type commands and NOP are both accessed by “ALUOp == 2'b11” this time , and be further determined by “func3”.



```verilog
end else if(ALUOp == 2'b11) begin
    case(func3)
        3'b000: begin                       //addi
            ALU_Ctrl_o <= 4'b0010;
        end
        3'b001: begin                       //slli
            ALU_Ctrl_o <= 4'b0100;
        end
        3'b010: begin                       //slti
            ALU_Ctrl_o <= 4'b0111;
        end
        default: begin                      //nop
            ALU_Ctrl_o <= 4'b0010;
        end
    endcase
end
```

3. Imm_Gen.v :

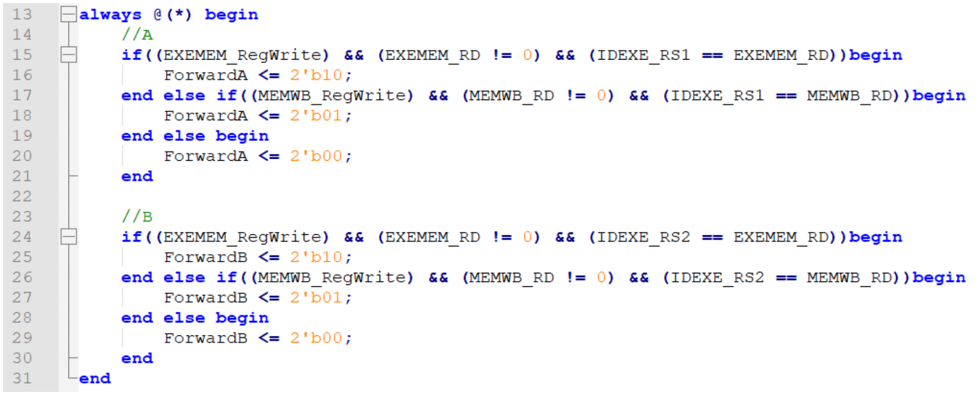
The structure is a little different from our previous one , but it’s basically the same . We chose different cases through “opcode” . Then in each case , we set “Imm_Gen_o” in the form that meets the requirement of the slide .



```verilog
always@(*) begin
    case(opcode)
        7'b0110011:     // R-type
            Imm_Gen_o <= 0;
        7'b0010011,7'b0000011,7'b1100111: // I-type (addi,lw,jalr)
            Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[30:20]};
        7'b0100011:     // S-type (sw)
            Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[30:25], instr_i[11:7]};
        7'b1100011:     // B-type (branch)
            Imm_Gen_o <= {{21{instr_i[31]}}, instr_i[7],instr_i[30:25], instr_i[11:8]};
        7'b1101111:     // J-type (jal)
            Imm_Gen_o <= {{12{instr_i[31]}}, instr_i[19:12],instr_i[20], instr_i[30:21]};
    endcase
end
```

4. ForwardingUnit.v :

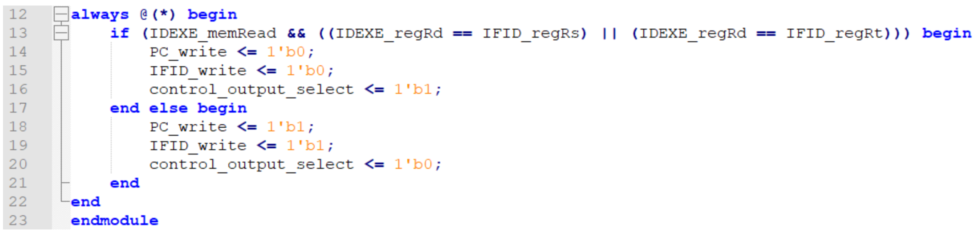
For both “ForwardA” and “ForwardB” are the same operation . If “RegWrite” == 1'b1(needs write back) , “RD” != 0 (not NOP) , and “IDEXE_RS” == “RD” (data dependency exists) , then forward . However in each part , “EXEMEM” has higher priority than “MEMWB” .



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

    //B
    if((EXEMEM_RegWrite) && (EXEMEM_RD != 0) && (IDEXE_RS2 == EXEMEM_RD))begin
        ForwardB <= 2'b10;
    end else if((MEMWB_RegWrite) && (MEMWB_RD != 0) && (IDEXE_RS2 == MEMWB_RD))begin
        ForwardB <= 2'b01;
    end else begin
        ForwardB <= 2'b00;
    end
end
```

5. Hazard_detection.v:

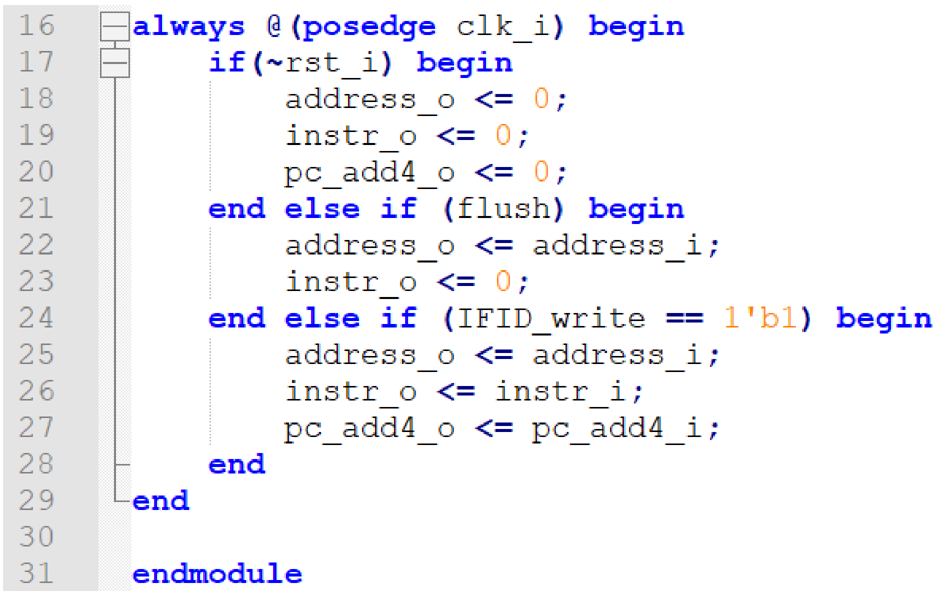
If load(before “&&” in “if()”) and use(after “&&” in “if()”) , then give nop , no write , and set “control_output_select” be 1 .



```
always @(*) begin
    if (IDEXE_memRead && ((IDEXE_regRd == IFID_regRs) || (IDEXE_regRd == IFID_regRt))) begin
        PC_write <= 1'b0;
        IFID_write <= 1'b0;
        control_output_select <= 1'b1;
    end else begin
        PC_write <= 1'b1;
        IFID_write <= 1'b1;
        control_output_select <= 1'b0;
    end
end
endmodule
```

6. IFID_register.v:

First , if “~rst_i” , then set all ouputs as 0 . Second , if “flush” exists , then make “address_o” be “address_i” and “instro_o” be 0 . Otherwise , set all values of outputs with the corresponding inputs .

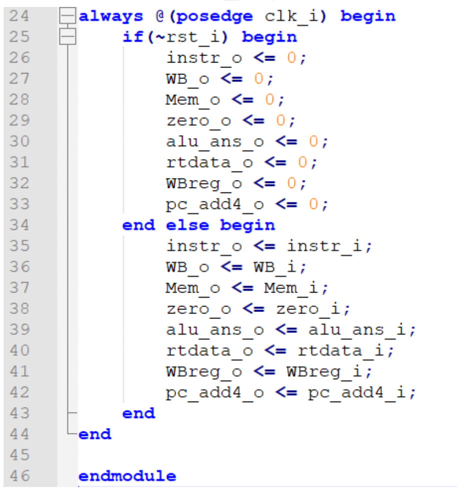


```
always @(posedge clk_i) begin
    if(~rst_i) begin
        address_o <= 0;
        instr_o <= 0;
        pc_add4_o <= 0;
    end else if (flush) begin
        address_o <= address_i;
        instr_o <= 0;
    end else if (IFID_write == 1'b1) begin
        address_o <= address_i;
        instr_o <= instr_i;
        pc_add4_o <= pc_add4_i;
    end
end

endmodule
```

7. EXEMEM_register.v , IDEXE_register.v , and MEMWB_register.v:

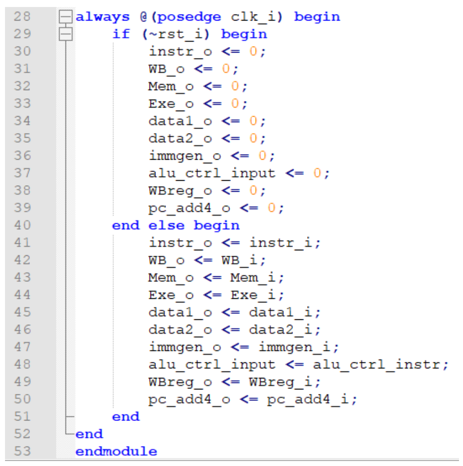
Basically , all are implemented with the same logic . If “~rst_i” , then set all the outputs as 0 , otherwise set their value with the corresponding inputs .

EXEMEM_register.v



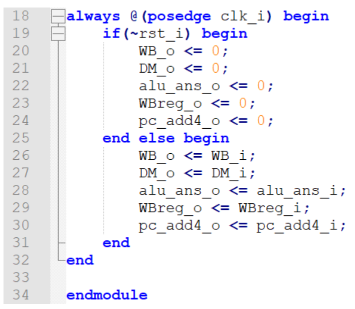
```
always @(posedge clk_i) begin
    if(~rst_i) begin
        instr_o <= 0;
        WB_o <= 0;
        Mem_o <= 0;
        zero_o <= 0;
        alu_ans_o <= 0;
        rtdata_o <= 0;
        WBreg_o <= 0;
        pc_add4_o <= 0;
    end else begin
        instr_o <= instr_i;
        WB_o <= WB_i;
        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
```

IDEXE_register.v



```
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
```

MEMWB_register.v

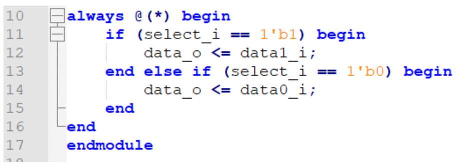


```
always @(posedge clk_i) begin
    if(~rst_i) begin
        WB_o <= 0;
        DM_o <= 0;
        alu_ans_o <= 0;
        WBreg_o <= 0;
        pc_add4_o <= 0;
    end else begin
        WB_o <= WB_i;
        DM_o <= DM_i;
        alu_ans_o <= alu_ans_i;
        WBreg_o <= WBreg_i;
        pc_add4_o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

endmodule
```

8. MUX_2to1 & 3to1.v , and Shift_Left_1.v:

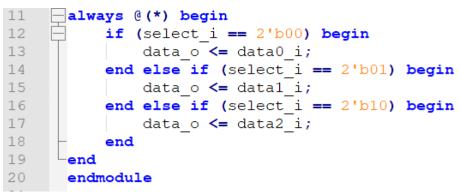
Both multiplexers are implemented with the same logic , getting the corresponding “data_o” with different “select_i” . “Shift_Left_1.v” is just for setting shifted “data_i” as “data_o” .

MUX_2to1.v



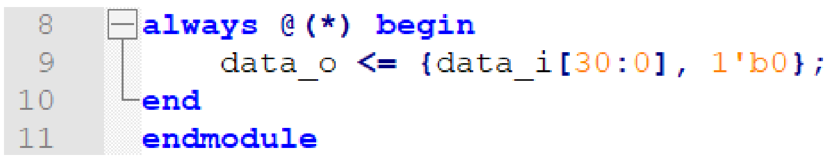
```
always @(*) begin
    if (select_i == 1'b1) begin
        data_o <= data1_i;
    end else if (select_i == 1'b0) begin
        data_o <= data0_i;
    end
end
endmodule
```

MUX_3to1.v



```
always @(*) begin
    if (select_i == 2'b00) begin
        data_o <= data0_i;
    end else if (select_i == 2'b01) begin
        data_o <= data1_i;
    end else if (select_i == 2'b10) begin
        data_o <= data2_i;
    end
end
endmodule
```

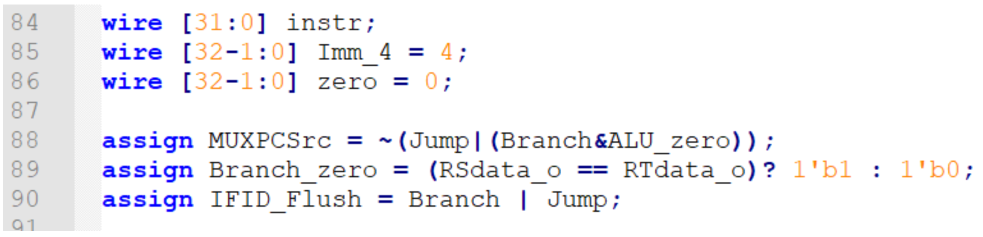
Shift_Left_1.v



```
always @(*) begin
    data_o <= {data_i[30:0], 1'b0};
end
endmodule
```

9. Pipeline_CPU.v:

Consists of all other modules whose corresponding unit exist in the structure of CPU . Connect each modules with each other through the required datapaths and assign them with specific signals to make the cpu run correctly . Also , add wires for some inputs that requires specific numbers or instructions . In addition , assign "~(Jump|(Branch&ALU_zero))" , "equality of data output of RS&RT", and "Branch | Jump" with "MUXPCSrc" , "Branch_zero" , and "IFID_Flush" respectively .

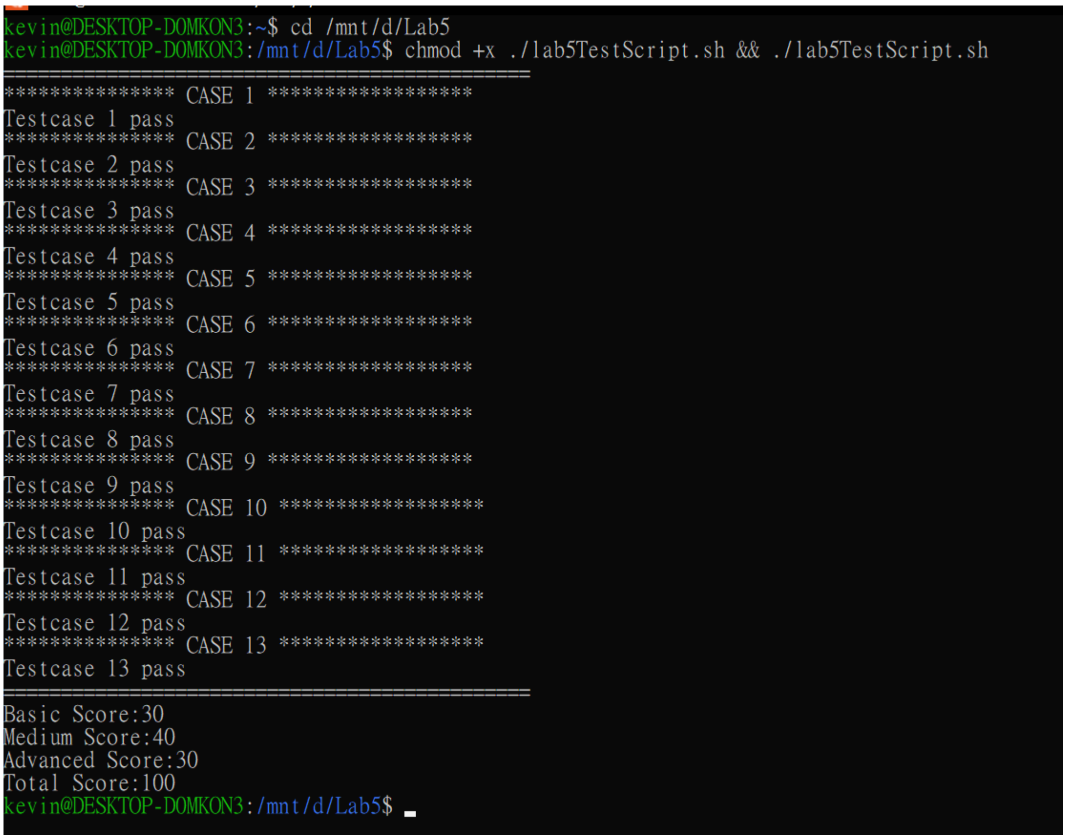


```
wire [31:0] instr;
wire [32-1:0] Imm_4 = 4;
wire [32-1:0] zero = 0;

assign MUXPCSrc = ~(Jump|(Branch&ALU_zero));
assign Branch_zero = (RSdata_o == RTdata_o)? 1'b1 : 1'b0;
assign IFID_Flush = Branch | Jump;
```

10. Rest of files that we've implemented before:

All with the same structure as before .

Part II. Implementation results :



```
kevin@DESKTOP-DOMKON3:~$ cd /mnt/d/Lab5
kevin@DESKTOP-DOMKON3:/mnt/d/Lab5$ chmod +x ./lab5TestScript.sh && ./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
kevin@DESKTOP-DOMKON3:/mnt/d/Lab5$
```

Part III. Problems encountered and solutions :

There are definitely much more details that we need to be cautious of in lab5 than in lab 4 . When dealing with this lab , we met 4 problems .

First , we once forgot to add operations that deal with nop in “Decoder.v” and “ALU_Ctrl.v” , making the values from each file sometimes wrong .

Second , the structure of this cpu is a more complicated task for us this time . It took us a long time to find out that we miss-connected “data0_i” in “MUX_2to1 MUX_ALUSrc” . Both of us didn’t find this error even if we both had checked it for several times .

Third , the lecture slide seems to have no information of how to connect 3to1 Mux in phase “WB” , this also take us a while to figure it out .

Finally in “IFID_register.v” , we set both “~rst_i” and “flush” with the same result in the beginning . Apparently , that is not correct , so we eventually separate them into two parts .