Report-Lab05

1. Detailed description of the implementation

   a. Adder.v

```
3   module Adder(
4       input  [32-1:0] src1_i,
5       input  [32-1:0] src2_i,
6       output [32-1:0] sum_o
7   );
```

```
9   /* Write your code HERE */
10
11  assign sum_o=src1_i+src2_i;
12
13  endmodule
```

In this section, we change the type of *sum_o*, we cancel the type of reg. Then, we just do the addition of source 1 and source 2 and get the value of the summation.

   b. ALU_Ctrl.v

```
8   /* Write your code HERE */
9
10  always @(*) begin
11      case(ALUOp)
12          //R-type: add sub and or xor slt
13          2'b10:begin
14              if(instr==4'b0000) begin //add
15                  ALU_Ctrl_o=4'b0010;
16              end
17              else if(instr==4'b1000) begin //sub
18                  ALU_Ctrl_o=4'b0110;
19              end
20              else if(instr==4'b0111) begin //and
21                  ALU_Ctrl_o=4'b0000;
22              end
23              else if(instr==4'b0110) begin //or
24                  ALU_Ctrl_o=4'b0001;
25              end
26              else if(instr==4'b0100) begin //xor
27                  ALU_Ctrl_o=4'b0111;
28              end
29              else if(instr==4'b0010) begin //slt
30                  ALU_Ctrl_o=4'b0011;
31              end
32          end
```

```
33          //addi nop slti slli
34          2'b11:begin
35              if(instr[2:0]==3'b000) begin //addi nop
36                  ALU_Ctrl_o=4'b0010;
37              end
38              else if(instr[2:0]==3'b010) begin //slti
39                  ALU_Ctrl_o=4'b0011;
40              end
41              else if(instr==4'b0001) begin //slli
42                  ALU_Ctrl_o=4'b1100;
43              end
44          end
45          //load store
46          2'b00:begin
47              ALU_Ctrl_o=4'b0010;
48          end
49          //beq
50          2'b01:begin
51              ALU_Ctrl_o=4'b0110;
52          end
53          //jal jalr
54          default:ALU_Ctrl_o=4'bxxxx;
55      endcase
56  end
57
58  endmodule
```

In this section, we use *ALUOp* to identify what type of instruction is going to be executed. Then, we use *instr* to assure what operation is definitely to be executed. Finally, we decide the value of *ALU_Ctrl_o*. As the same in every lab, we also need the default condition that the value of *ALU_Ctrl_o* is xxxx.

c. alu.v

```
/* Write your code HERE */

reg [32-1:0] set;

always @(*) begin
    if(!rst_n) begin
        result=0;
        zero=0;
    end
    else begin
        case(ALU_control)
            4'b0010: begin //add
                result=src1+src2;
            end
            4'b0110: begin //sub
                result=src1-src2;
            end
            4'b0000: begin //and
                result=src1&src2;
            end
            4'b0001: begin //or
                result=src1|src2;
            end
            4'b0111:begin //xor
                result=src1^src2;
            end
            4'b0011:begin //slt
                set=src1-src2;
                if(set[31]==1) begin
                    result=32'b00000000000000000000000000000001;
                end
                else begin
                    result=32'b00000000000000000000000000000000;
                end
            end
            4'b1100: begin //sll
                result=src1<<src2;
            end
            default: begin
                result=32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
            end
        endcase
    end
    zero=~(|result);
end

endmodule
```

In this section, we use *ALU_control* of identify what operation is going to be executed with the corresponding operands.

d. Decoder.v

```
/* Write your code HERE */

assign opcode=instr_i[6:0];
assign funct3=instr_i[14:12];

always @(*) begin
    if(opcode==7'b0110011) begin //R-type 00001,00010
        {RegWrite,Branch,Jump}=3'b100;
        {MemRead,MemWrite}=2'b00;
        {ALUSrc,MemtoReg}=2'b00;
        ALUOp=2'b10;
    end
    else if(opcode==7'b0010011) begin //I-type general 00101,00011
        {RegWrite,Branch,Jump}=3'b100;
        {MemRead,MemWrite}=2'b00;
        {ALUSrc,MemtoReg}=2'b10;
        ALUOp=2'b11;
    end
    else if(opcode==7'b0000011) begin //load 00111,10000
        {RegWrite,Branch,Jump}=3'b100;
        {MemRead,MemWrite}=2'b10;
        {ALUSrc,MemtoReg}=2'b11;
        ALUOp=2'b00;
    end
    else if(opcode==7'b0100011) begin //store 00100,01000
        {RegWrite,Branch,Jump}=3'b000;
        {MemRead,MemWrite}=2'b01;
        {ALUSrc,MemtoReg}=2'b10;
        ALUOp=2'b00;
    end
    else if(opcode==7'b1100011) begin //branch 00000,00101
        {RegWrite,Branch,Jump}=3'b010;
        {MemRead,MemWrite}=2'b00;
        {ALUSrc,MemtoReg}=2'b00;
        ALUOp=2'b01;
    end
    else if(opcode==7'b1101111) begin //jal 01001,00000
        {RegWrite,Branch,Jump}=3'b101;
        {MemRead,MemWrite}=2'b00;
        {ALUSrc,MemtoReg}=2'b00;
        ALUOp=2'b00;
    end
    else if(opcode==7'b1100111) begin //jalr 10101,00000
        {RegWrite,Branch,Jump}=3'b100;
        {MemRead,MemWrite}=2'b00;
        {ALUSrc,MemtoReg}=2'b10;
        ALUOp=2'b00;
    end
    else begin
        {RegWrite,Branch,Jump}=3'b000;
        {MemRead,MemWrite}=2'b00;
        {ALUSrc,MemtoReg}=2'b00;
        ALUOp=2'bxx;
    end
end

endmodule
```

In this section, we assign the value of *RegWrite*, *Branch*, *Jump*, *MemRead*, *MemWrite*, *ALUSrc*, *MemtoReg*, and *ALUOp* according to the datapath in the slide.

Signal\opcode	R-type	I-type	load	store	branch	jal	jalr	Default
RegWrite	1	1	1	0	0	1	1	0
Branch	0	0	0	0	1	0	0	0
Jump	0	0	0	0	0	1	0	0
MemRead	0	0	1	0	0	0	0	0
MemWrite	0	0	0	1	0	0	0	0
ALUSrc	0	1	1	1	0	0	1	0
MemtoReg	0	0	1	0	0	0	0	0
ALUOp	10	11	00	00	01	00	00	xx

e. ForwardingUnit.v

```
12  /* Write your code HERE */
13
14  always @(*) begin
15      //FoewardA
16      if(EXEMEM_RegWrite&&(EXEMEM_RD!=5'b00000)&&(EXEMEM_RD==IDEXE_RS1)) begin
17          ForwardA=2'b10;
18      end
19      else if(MEMWB_RegWrite&&(MEMWB_RD!=5'b00000)&&
20              ~(EXEMEM_RegWrite&&(EXEMEM_RD!=5'b00000)&&(EXEMEM_RD==IDEXE_RS1))&&
21              (MEMWB_RD==IDEXE_RS1)) begin
22          ForwardA=2'b01;
23      end
24      else begin
25          ForwardA=2'b00;
26      end
27      //ForwardB
28      if(EXEMEM_RegWrite&&(EXEMEM_RD!=5'b00000)&&(EXEMEM_RD==IDEXE_RS2)) begin
29          ForwardB=2'b10;
30      end
31      else if(MEMWB_RegWrite&&(MEMWB_RD!=5'b00000)&&
32              ~(EXEMEM_RegWrite&&(EXEMEM_RD!=5'b00000)&&(EXEMEM_RD==IDEXE_RS2))&&
33              (MEMWB_RD==IDEXE_RS2)) begin
34          ForwardB=2'b01;
35      end
36      else begin
37          ForwardB=2'b00;
38      end
39  end
40
41  endmodule
```

In this section, we deal with two MUXs, which is *ForwardA* and *ForwardB* respectively.

First, for ForwardA, we need to process three conditions. The first is at the part of MEM. We need to assure that *EXEMEM_RegWrite* is not zero, and the value of *EXEMEM_RD* is equal to that of *IDEXE_RS1*. Then, the value of *ForwardA* is 10. The second condition is at the part of WB. There are some constraints in “if” more than that of the first condition. The former of the constraints is the negative part of that in the MEM part. Similarly, the value of

EXEMEM_RD is equal to that of *IDEXE_RS1*. Then, the value of *ForwardA* is 01. The other condition is default so that *ForwardA* is 00.

On the other hand, for *ForwardB*, we also need to process three conditions which are much similar to that of *ForwardA*. *IDEXE_RS2* is used to replace *IDEXE_RS1*.

f. Hazard_detection.v

In this part, we have two conditions to process.

```
    /* 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
```

The first is that *IDEXE_memRead* is not zero and the condition that *IDEXE_regRd* equaling to *IFID_regRS* or *IFID_regRt*. Then, *PC_write* and *IFID_write* are both 0 and *control_output_select* is 1. In this way, all the above actions are stopped.

The other condition is the counterpart. Hence, the value of above registers is contrary.

g. Imm_Gen.v

```
/* Write your code HERE */

wire [7-1:0] opcode;
assign opcode=instr_i[6:0];

always @(*) begin
    //I-type: addi, Load, JALR
    if((opcode==7'b0010011)||(opcode==7'b0000011)||(opcode==7'b1100111)) begin
        Imm_Gen_o={{20{instr_i[31]}},instr_i[31:20]};
    end
    //S-type: Store
    else if (opcode==7'b0100011) begin
        Imm_Gen_o={{20{instr_i[31]}},instr_i[31:25],instr_i[11:7]};
    end
    //B-type: Branch
    else if (opcode==7'b1100011) begin
        Imm_Gen_o={{20{instr_i[31]}},instr_i[7],instr_i[30:25],instr_i[11:8],1'b0};
    end
    //J-type: JAL
    else if (opcode==7'b1101111) begin
        Imm_Gen_o={{12{instr_i[31]}},instr_i[19:12],instr_i[20],instr_i[30:21],1'b0};
    end
end

endmodule
```

At first, we process I-type, addi, load and jalr. We generate the value of *Imm_Gen_o* by

sign extension of the thirty first bit of *instr_i* and the rest is the thirty first bit to the 20th bit of *instr_i*.

Secondly, we process S-type, store. Similarly, we generated the value of *Imm_Gen_o* by sign extension of the thirty first bit of *instr_i*, and the other two sections are the thirty first bit to the 25th bit of *instr_i* and the eleventh bit to the seventh bit of *insr_i*.

For the last two types, Branch and JAL, we do the similar things as the above. However, we need to plus '0' at the last bit of *Imm_Gen_o*, since we have to double the value. i.e. 010(2) -> 100(4)

h. MUX_2to1.v

```
 9    /* Write your code HERE */
10
11    always @(*) begin
12        data_o=(!select_i)?data0_i:data1_i;
13    end
14
15    endmodule
```

In this section, *select_i* is a controller to decide the value of *data_o*. If *select_i* is not zero, *data_o* is *data0_i*. otherwise, *data_o* is *data1_i*.

i. MUX_3to1.v

```
10   /* Write your code HERE */
11
12   always @(*) begin
13       if(select_i==2'b00) begin
14           data_o=data0_i;
15       end
16       else if(select_i==2'b01) begin
17           data_o=data1_i;
18       end
19       else if(select_i==2'b10) begin
20           data_o=data2_i;
21       end
22       else begin
23       end
24   end
25
26   endmodule
```

In this section, the code is similar to *MUX_3to1.v*. Therefore, if *select_i* is 00, *data_o* is *data0_i*. If *select_i* is 01, *data_o* is *data1_i*. The last part, if *select_i* is 10, *data_o* is *data2_i*.

j. Shift_Left_1.v

```
7     /* Write your code HERE */
8
9     assign data_o=data_i<<1;
10
11    endmodule
```

In this section, we just use the operand "<<" to let the bits of *data_o* shift left for one bit.

k. IFID_register.v

```
15  /* Write your code HERE */
16
17  always @(posedge clk_i) begin
18      if(!rst_i) begin
19          address_o<=32'b00000000000000000000000000000000;
20          instr_o<=32'b00000000000000000000000000000000;
21          pc_add4_o<=32'b00000000000000000000000000000000;
22      end
23      else if(!IFID_write) begin
24      end
25      else if(flush) begin
26          address_o<=32'b00000000000000000000000000000000;
27          instr_o<=32'b00000000000000000000000000000000;
28          pc_add4_o<=32'b00000000000000000000000000000000;
29      end
30      else begin
31          address_o<=address_i;
32          instr_o<=instr_i;
33          pc_add4_o<=pc_add4_i;
34      end
35  end
36
37  endmodule
```

In this section, we process the four conditions. First, if *rst_i* is 0, which means we need to reset. Hence, the values are all 32-bit 0. Secondly, if the value of *IFID_write* is 0, which means we don't need to write anything. Hence, we do nothing. Third, if the value of *flush* is 1, we also let all the values are 32-bit 0. The last, the value of *address_o* is *address_i*. The value of *instr_o* is *instr_i*. The value of *pc_add4_o* is *pc_add4_i*.

l. IDEXE_register.v

```
/* Write your code HERE */

always @(posedge clk_i) begin
    if(!rst_i) begin
        instr_o<=32'b00000000000000000000000000000000;
        WB_o<=3'b000;
        Mem_o<=2'b00;
        Exe_o<=3'b000;
        data1_o<=32'b00000000000000000000000000000000;
        data2_o<=32'b00000000000000000000000000000000;
        immgen_o<=32'b00000000000000000000000000000000;
        alu_ctrl_input<=4'b0000;
        WBreg_o<=5'b00000;
        pc_add4_o<=32'b00000000000000000000000000000000;
    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
```

In this section, if the value of *rst_i* is 0, we reset all the values of the above parameters to be 0. On the other hand, the value of the parameters which are output is given by the corresponding parameters which are input.

m. EXEMEM_register.v

```
/* Write your code HERE */

always @(posedge clk_i) begin
    if(!rst_i) begin
        instr_o<=32'b00000000000000000000000000000000;
        WB_o<=3'b000;
        Mem_o<=2'b00;
        zero_o<=1'b0;
        alu_ans_o<=32'b00000000000000000000000000000000;
        rtdata_o<=32'b00000000000000000000000000000000;
        WBreg_o<=5'b00000;
        pc_add4_o<=32'b00000000000000000000000000000000;
    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
```

In the section, we do the similar things to that of the last section.

n. MEMWB_register.v

```
/* Write your code HERE */

always @(posedge clk_i) begin
    if(!rst_i) begin
        WB_o<=3'b000;
        DM_o<=32'b00000000000000000000000000000000;
        alu_ans_o<=32'b00000000000000000000000000000000;
        WBreg_o<=5'b00000;
        pc_add4_o<=32'b00000000000000000000000000000000;
    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
```

In this section, we also do the similar things to that of the last two sections.

o. Pipeline_CPU.v

```
wire [31:0] IFID_Instr;
```

At first, we add a new parameter named IFID_Instr.

```
// IF
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'b00000000000000000000000000000100),
    .sum_o(PC_Add4)
);

Instr_Memory IM(
    .addr_i(PC_o),
    .instr_o(IFID_Instr)
);
```

In Adder, we do the operation of PC+4. Hence, the value of src2_i is 4.

```
112
113   IFID_register IFtoID(
114       .clk_i(clk_i),
115       .rst_i(rst_i),
116       .address_i(PC_o),
117       .instr_i(IFID_Instr),
118       .pc_add4_i(PC_Add4),
119       .IFID_write(IFID_Write),
120       .flush(IFID_Flush),
121       .address_o(IFID_PC_o),
122       .instr_o(IFID_Instr_o),
123       .pc_add4_o(IFID_PC_Add4_o)
124   );
125
126   // ID
127
128   assign Branch_zero=(RSdata_o!=RTdata_o)?1'b0:1'b1;
129   assign MUXPCSrc=Jump|(Branch&Branch_zero);
130   assign IFID_Flush=MUXPCSrc;
```

In this section, we initialize the value of Branch_zero, MUXPCSrc and IFID_Flush.

```
132   Hazard_detection Hazard_detection_obj(
133       .IFID_regRs(IFID_Instr_o[19:15]),
134       .IFID_regRt(IFID_Instr_o[24:20]),
135       .IDEXE_regRd(IDEXE_Instr_11_7_o),
136       .IDEXE_memRead(IDEXE_Mem_o[1]),
137       .PC_write(PC_write),
138       .IFID_write(IFID_Write),
139       .control_output_select(MUXControl)
140   );
141
142   MUX_2to1 MUX_control(
143       .data0_i({24'b000000000000000000000000,Jump,MemtoReg,RegWrite,MemRead,MemWrite,ALUOp,ALUSrc}),
144       .data1_i(32'b00000000000000000000000000000000),
145       .select_i(MUXControl),
146       .data_o(MUX_control_o)
147   );
148
149   Decoder Decoder(
150       .instr_i(IFID_Instr_o),
151       .Branch(Branch),
152       .ALUSrc(ALUSrc),
153       .RegWrite(RegWrite),
154       .ALUOp(ALUOp),
155       .MemRead(MemRead),
156       .MemWrite(MemWrite),
157       .MemtoReg(MemtoReg),
158       .Jump(Jump)
159   );
```

In MUX_2to1, we have two parameters data0_i and data1_i. data0_i consists of 24-bit 0 and the other totally 8-bit signals. Data1_i is 32-bit 0. If the value of MUXControl is zero, the control signal would be passed.

```
161   Reg_File RF(
162       .clk_i(clk_i),
163       .rst_i(rst_i),
164       .RSaddr_i(IFID_Instr_o[19:15]),
165       .RTaddr_i(IFID_Instr_o[24:20]),
166       .RDaddr_i(MEMWB_Instr_11_7_o),
167       .RDdata_i(MUXMemtoReg_o),
168       .RegWrite_i(MEMWB_WB_o[0]),
169       .RSdata_o(RSdata_o),
170       .RTdata_o(RTdata_o)
171   );
172
173   Imm_Gen ImmGen(
174       .instr_i(IFID_Instr_o),
175       .Imm_Gen_o(Imm_Gen_o)
176   );
177
178   Shift_Left_1 SL1(
179       .data_i(Imm_Gen_o),
180       .data_o(SL1_o)
181   );
182
183   Adder Branch_Adder(
184       .src1_i(Imm_Gen_o),
185       .src2_i(IFID_PC_o),
186       .sum_o(PC_Add_Immediate)
187   );
```

In this section, we fill in the corresponding parameter.

```
189   IDEXE_register IDtoEXE(
190       .clk_i(clk_i),
191       .rst_i(rst_i),
192       .instr_i(IFID_Instr_o),
193       .WB_i(MUX_control_o[7:5]),
194       .Mem_i(MUX_control_o[4:3]),
195       .Exe_i(MUX_control_o[2:0]),
196       .data1_i(RSdata_o),
197       .data2_i(RTdata_o),
198       .immgen_i(Imm_Gen_o),
199       .alu_ctrl_instr({IFID_Instr_o[30],IFID_Instr_o[14:12]}),
200       .WBreg_i(IFID_Instr_o[11:7]),
201       .pc_add4_i(IFID_PC_Add4_o),
202       .instr_o(IDEXE_Instr_o),
203       .WB_o(IDEXE_WB_o),
204       .Mem_o(IDEXE_Mem_o),
205       .Exe_o(IDEXE_Exe_o),
206       .data1_o(IDEXE_RSdata_o),
207       .data2_o(IDEXE_RTdata_o),
208       .immgen_o(IDEXE_ImmGen_o),
209       .alu_ctrl_input(IDEXE_Instr_30_14_12_o),
210       .WBreg_o(IDEXE_Instr_11_7_o),
211       .pc_add4_o(IDEXE_PC_add4_o)
212   );
```

In this section, we fill in the corresponding parameter.

```
214  // EXE
215  MUX_2to1 MUX_ALUSrc(
216      .data0_i(ALUSrc2_o),
217      .data1_i(IDEXE_ImmGen_o),
218      .select_i(IDEXE_Exe_o[0]),
219      .data_o(MUXALUSrc_o)
220  );
221
222  ForwardingUnit FWUnit(
223      .IDEXE_RS1(IDEXE_Instr_o[19:15]),
224      .IDEXE_RS2(IDEXE_Instr_o[24:20]),
225      .EXEMEM_RD(EXEMEM_Instr_11_7_o),
226      .MEMWB_RD(MEMWB_Instr_11_7_o),
227      .EXEMEM_RegWrite(EXEMEM_WB_o[0]),
228      .MEMWB_RegWrite(MEMWB_WB_o[0]),
229      .ForwardA(ForwardA),
230      .ForwardB(ForwardB)
231  );
232
233  MUX_3to1 MUX_ALU_src1(
234      .data0_i(IDEXE_RSdata_o),
235      .data1_i(MUXMemtoReg_o),
236      .data2_i(EXEMEM_ALUResult_o),
237      .select_i(ForwardA),
238      .data_o(ALUSrc1_o)
239  );
```

In this section, we fill in the corresponding parameter

```
241  MUX_3to1 MUX_ALU_src2(
242      .data0_i(IDEXE_RTdata_o),
243      .data1_i(MUXMemtoReg_o),
244      .data2_i(EXEMEM_ALUResult_o),
245      .select_i(ForwardB),
246      .data_o(ALUSrc2_o)
247  );
248
249  ALU_Ctrl ALU_Ctrl(
250      .instr(IDEXE_Instr_30_14_12_o),
251      .ALUOp(IDEXE_Exe_o[2:1]),
252      .ALU_Ctrl_o(ALU_Ctrl_o)
253  );
254
255  alu alu(
256      .rst_n(rst_i),
257      .src1(ALUSrc1_o),
258      .src2(MUXALUSrc_o),
259      .ALU_control(ALU_Ctrl_o),
260      .result(ALUResult),
261      .zero(ALU_zero)
262  );
```

In this section, we fill in the corresponding parameter

```
264   EXEMEM_register EXEtoMEM(
265       .clk_i(clk_i),
266       .rst_i(rst_i),
267       .instr_i(IDEXE_Instr_o),
268       .WB_i(IDEXE_WB_o),
269       .Mem_i(IDEXE_Mem_o),
270       .zero_i(ALU_zero),
271       .alu_ans_i(ALUResult),
272       .rtdata_i(ALUSrc2_o),
273       .WBreg_i(IDEXE_Instr_11_7_o),
274       .pc_add4_i(IDEXE_PC_add4_o),
275       .instr_o(EXEMEM_Instr_o),
276       .WB_o(EXEMEM_WB_o),
277       .Mem_o(EXEMEM_Mem_o),
278       .zero_o(EXEMEM_Zero_o),
279       .alu_ans_o(EXEMEM_ALUResult_o),
280       .rtdata_o(EXEMEM_RTdata_o),
281       .WBreg_o(EXEMEM_Instr_11_7_o),
282       .pc_add4_o(EXEMEM_PC_Add4_o)
283   );
```

In this section, we fill in the corresponding parameter

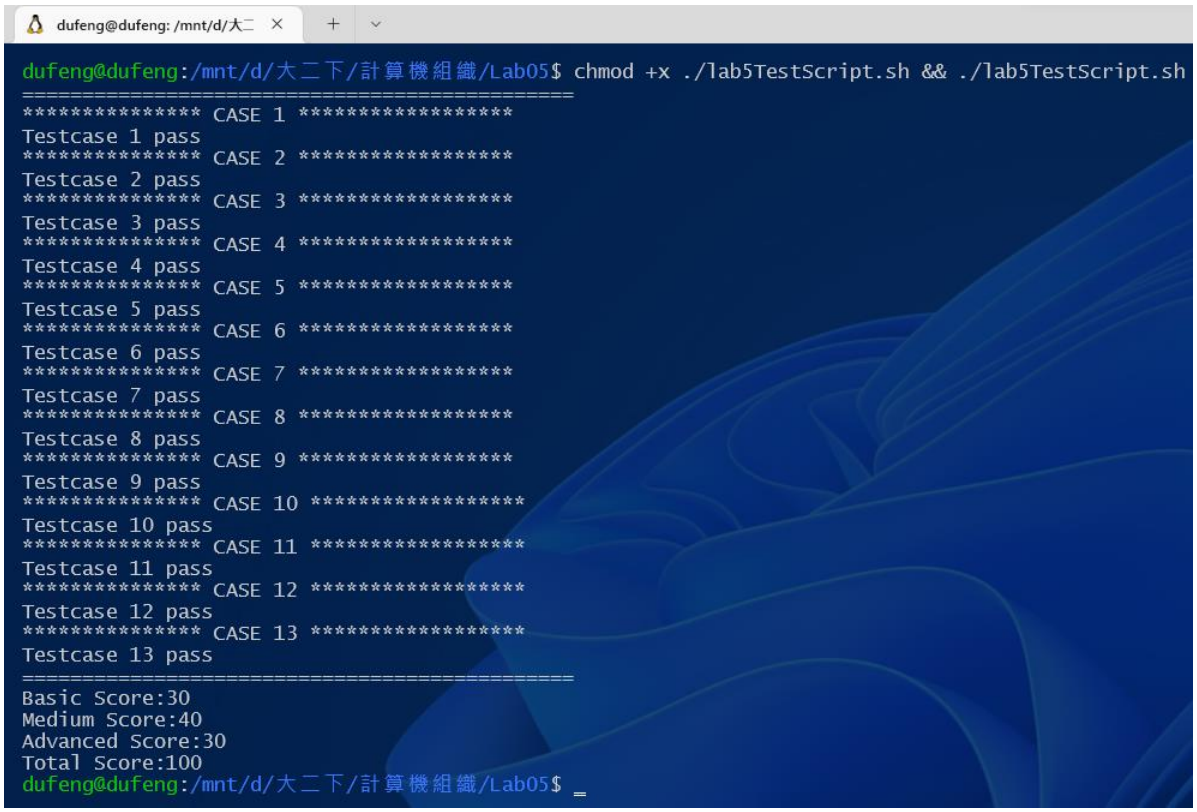
```
285   // MEM
286   Data_Memory Data_Memory(
287       .clk_i(clk_i),
288       .addr_i(EXEMEM_ALUResult_o),
289       .data_i(EXEMEM_RTdata_o),
290       .MemRead_i(EXEMEM_Mem_o[1]),
291       .MemWrite_i(EXEMEM_Mem_o[0]),
292       .data_o(DM_o)
293   );
294
295   MEMWB_register MEMtoWB(
296       .clk_i(clk_i),
297       .rst_i(rst_i),
298       .WB_i(EXEMEM_WB_o),
299       .DM_i(DM_o),
300       .alu_ans_i(EXEMEM_ALUResult_o),
301       .WBreg_i(EXEMEM_Instr_11_7_o),
302       .pc_add4_i(EXEMEM_PC_Add4_o),
303       .WB_o(MEMWB_WB_o),
304       .DM_o(MEMWB_DM_o),
305       .alu_ans_o(MEMWB_ALUresult_o),
306       .WBreg_o(MEMWB_Instr_11_7_o),
307       .pc_add4_o(MEMWB_PC_Add4_o)
308   );
```

In this section, we fill in the corresponding parameter

```
310   // WB
311   MUX_3to1 MUX_MemtoReg(
312       .data0_i(MEMWB_ALUresult_o),
313       .data1_i(MEMWB_DM_o),
314       .data2_i(MEMWB_PC_Add4_o),
315       .select_i((MEMWB_WB_o[2]==1'b1)?2'b10:((MEMWB_WB_o[1]==1'b1)?2'b01:2'b00)),
316       .data_o(MUXMemtoReg_o)
317   );
318
319   endmodule
```

In this section, we use the value of *MEMWB_WB_o* to decide the value of *select_i*. If the second bit of *MEMWB_WB_o* is 1, the value of *select_i* is 10. If the first bit of *MEMWB_WB_o* is 1, the value of *select_i* is 01. For the other cases, the value of *select_i* is 00.

2. Implementation results



3. Problems encountered and solutions

Since this Lab is larger than before, although I have used gtkwave to debug, it still was very hard to find which part got in troubles. It needed us to check each command is what kind of type one by one and to distinguish it would use which module or register. But after review each part, it truly let us be more familiar with the structure of pipeline cpu.