LAB 4 : Single-Cycle CPU

Group 17　109550135 范恩宇 109550156 曾偉杰

Part I.　Detailed description of the implementation :

1. Decoder.v:

First assign each outputs with a temporary signal , then deal with the cases through different “instr_i” . Then for different operations we asked to do in this lab , we set the required signals as true , and make it output corresponding ALUOp .

```
`timescale 1ns/1ps

module Decoder(
    input   [7-1:0]     instr_i,
    output              RegWrite,
    output              Branch,
    output              Jump,
    output              WriteBack1,
    output              WriteBack0,
    output              MemRead,
    output              MemWrite,
    output              ALUSrcA,
    output              ALUSrcB,
    output  [2-1:0]     ALUOp
);

/* Write your code HERE */
reg alusrc_A, alusrc_B, branch, regwrite, jump, write_back0, write_back1, mem_read, mem_write;
reg [1:0] aluop;
assign RegWrite = regwrite;
assign Branch = branch;
assign Jump = jump;
assign WriteBack1 = write_back1;
assign WriteBack0 = write_back0;
assign MemRead = mem_read;
assign MemWrite = mem_write;
assign ALUSrcA = alusrc_A;
assign ALUSrcB = alusrc_B;
assign ALUOp = aluop;
```

```
always@ (*) begin
    case(instr_i)
        7'b0110011: begin    //R-type
            regwrite <= 1;
            branch <= 0;
            jump <= 0;
            write_back1 <= 0;
            write_back0 <= 0;
            mem_read <= 0;
            mem_write <= 0;
            alusrc_A <= 0;
            alusrc_B <= 0;
            aluop <= 2'b10;
        end
        7'b0010011: begin    //addi
            regwrite <= 1;
            branch <= 0;
            jump <= 0;
            write_back1 <= 0;
            write_back0 <= 0;
            mem_read <= 0;
            mem_write <= 0;
            alusrc_A <= 0;
            alusrc_B <= 1;
            aluop <= 2'b00;
        end
```

```
57          7'b0000011: begin   //lw
58              regwrite <= 1;
59              branch <= 0;
60              jump <= 0;
61              write_back1 <= 0;
62              write_back0 <= 1;
63              mem_read <= 1;
64              mem_write <= 0;
65              alusrc_A <= 0;
66              alusrc_B <= 1;
67              aluop <= 2'b00;
68          end
69          7'b0100011: begin   //sw
70              regwrite <= 0;
71              branch <= 0;
72              jump <= 0;
73              write_back1 <= 0;
74              write_back0 <= 0;
75              mem_read <= 0;
76              mem_write <= 1;
77              alusrc_A <= 0;
78              alusrc_B <= 1;
79              aluop <= 2'b00;
80          end
```

```
81          7'b1100011: begin   //branch
82              regwrite <= 0;
83              branch <= 1;
84              jump <= 0;
85              write_back1 <= 0;
86              write_back0 <= 0;
87              mem_read <= 0;
88              mem_write <= 0;
89              alusrc_A <= 0;
90              alusrc_B <= 0;
91              aluop <= 2'b01;
92          end
93          7'b1101111: begin   //jal
94              regwrite <= 1;
95              branch <= 0;
96              jump <= 1;
97              write_back1 <= 1;
98              write_back0 <= 0;
99              mem_read <= 0;
100             mem_write <= 0;
101             alusrc_A <= 0;
102             alusrc_B <= 0;
103             aluop <= 2'b00;
104         end
```

```
105             7'b1100111: begin   //jalr
106                 regwrite <= 1;
107                 branch <= 0;
108                 jump <= 1;
109                 write_back1 <= 1;
110                 write_back0 <= 0;
111                 mem_read <= 0;
112                 mem_write <= 0;
113                 alusrc_A <= 1;
114                 alusrc_B <= 0;
115                 aluop <= 2'b00;
116             end
117             default: begin
118                 regwrite <= 0;
119                 branch <= 0;
120                 jump <= 0;
121                 write_back1 <= 0;
122                 write_back0 <= 0;
123                 mem_read <= 0;
124                 mem_write <= 0;
125                 alusrc_A <= 0;
126                 alusrc_B <= 0;
127                 aluop <= 2'b00;
128             end
129         endcase
130     end
131     endmodule
```

2. Imm_Gen.v :

We choose different case of options through the “opcode” . Then in each case section , we set “Imm_Gen_o[i]” as 1 or 0 in the range that shown on the slide . Since this is a file for sign-extension , the former actually means that we make it do sign-extension by Identifying the sign bit of “Imm_Gen_o” . And if sign bit is 1, we fill 1 into the remaining bits , otherwise we fill 0 .

As for the range we didn’t do the former operation , we set it with different parts of “instr_i[x:y]” , based on the type of the command and its corresponding structure that is shown on the slide .

```
2   `timescale 1ns/1ps
3
4   module Imm_Gen(
5       input  [31:0] instr_i,
6       output reg[31:0] Imm_Gen_o
7   );
8
9   //Internal Signals
10  wire    [7-1:0] opcode;
11  wire    [2:0]   func3;
12  wire    [3-1:0] Instr_field;
13
14  assign opcode = instr_i[6:0];
15  assign func3  = instr_i[14:12];
16
17  integer i;
18
```

```
always@(*) begin
    case(opcode)
        7'b0010011: begin                       //addi
            for (i = 11; i < 32; i++) begin
                if (instr_i[31] == 1) begin
                    Imm_Gen_o[i] <= 1;
                end else begin
                    Imm_Gen_o[i] <= 0;
                end
            end
            Imm_Gen_o[11:0] <= instr_i[31:20];
        end
        7'b0000011: begin                       //lw
            for (i = 11; i < 32; i++) begin
                if (instr_i[31] == 1) begin
                    Imm_Gen_o[i] <= 1;
                end else begin
                    Imm_Gen_o[i] <= 0;
                end
            end
            Imm_Gen_o[11:0] <= instr_i[31:20];
        end
        7'b0100011: begin                       //sw
            for (i = 12; i < 32; i++) begin
                if (instr_i[31] == 1) begin
                    Imm_Gen_o[i] <= 1;
                end else begin
                    Imm_Gen_o[i] <= 0;
                end
            end
            Imm_Gen_o[11:5] <= instr_i[31:25];
            Imm_Gen_o[4:0] <= instr_i[11:7];
        end
```

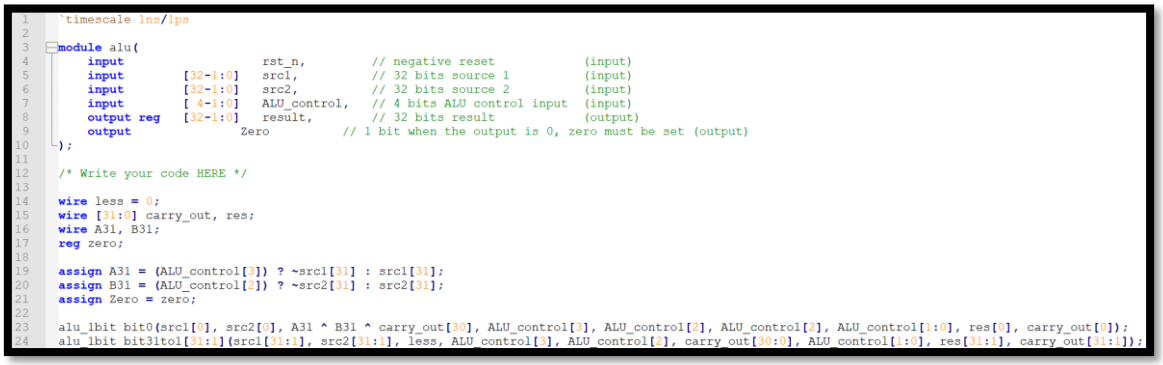
```
        7'b1100011: begin                       //branch
            for (i = 13; i < 32; i++) begin
                if (instr_i[31] == 1) begin
                    Imm_Gen_o[i] <= 1;
                end else begin
                    Imm_Gen_o[i] <= 0;
                end
            end
            Imm_Gen_o[12:0] <= {instr_i[31], instr_i[7], instr_i[30:25], instr_i[11:8], 1'b0};
        end
        7'b1101111: begin                       //jal
            for (i = 21; i < 32; i++) begin
                if (instr_i[31] == 1) begin
                    Imm_Gen_o[i] <= 1;
                end else begin
                    Imm_Gen_o[i] <= 0;
                end
            end
            Imm_Gen_o[20:0] <= {instr_i[31], instr_i[19:12], instr_i[20], instr_i[30:21], 1'b0};
        end
        7'b1100111: begin                       //jalr
            for (i = 11; i < 32; i++) begin
                if (instr_i[31] == 1) begin
                    Imm_Gen_o[i] <= 1;
                end else begin
                    Imm_Gen_o[i] <= 0;
                end
            end
            Imm_Gen_o[11:0] <= instr_i[31:20];
        end
    endcase
end
endmodule
```

3. alu.v :

We construct the complete ALU with 32 1-bit ALUs . For each 1-bit ALU , we add one full adder , two 2x1 multiplexers and one 4x1 multiplexer just like we did in lab2 . Then the result and carry out will be outputs of the 4x1 multiplexer and full adder respectively .

For the connected 1-bit alus , first let “A31” and “B31” be “ALU_control[3]” and “ALU_control[2]” determined by “src1[31]” and “src2[31]” respectively . As for the first 1-bit alu , set “less” = A31 ^ B31 ^ carry_out[30] , “cin” = ALU_control[2] . Then connect the 1-bit alus with the carryout of their former 1-bit alu as cin , and make the result of each 1-bit alu as the corresponding part of full result .

Finally for the running process , result will be the ones produced by the alus , but setting xor , sll , sra with self-defined operation . In the end , “zero” will be 1 once “result” equals to 1 .



```
`timescale 1ns/1ps

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              Zero          // 1 bit when the output is 0, zero must be set (output)
);

/* Write your code HERE */

wire less = 0;
wire [31:0] carry_out, res;
wire A31, B31;
reg zero;

assign A31 = (ALU_control[3]) ? ~src1[31] : src1[31];
assign B31 = (ALU_control[2]) ? ~src2[31] : src2[31];
assign Zero = zero;

alu_1bit bit0(src1[0], src2[0], A31 ^ B31 ^ carry_out[30], ALU_control[3], ALU_control[2], ALU_control[2], ALU_control[1:0], res[0], carry_out[0]);
alu_1bit bit31to1[31:1](src1[31:1], src2[31:1], less, ALU_control[3], ALU_control[2], carry_out[30:0], ALU_control[1:0], res[31:1], carry_out[31:1]);
```

```
always@ (*) begin

    case(ALU_control)
        4'b0011: result <= src1 ^ src2;     //xor
        4'b0100: result <= src1 << src2;    //sll
        4'b0101: result <= src1 >>> src2;   //sra
        default: result <= res;
    endcase

    if (result == 0) begin
        zero <= 1;
    end else begin
        zero <= 0;
    end

end
endmodule

module alu_1bit(
    input               src1,       //1 bit source 1  (input)
    input               src2,       //1 bit source 2  (input)
    input               less,       //1 bit less      (input)
    input               Ainvert,    //1 bit A_invert  (input)
    input               Binvert,    //1 bit B_invert  (input)
    input               cin,        //1 bit carry in  (input)
    input      [2-1:0]  operation,  //2 bit operation (input)
    output reg          result,     //1 bit result    (output)
    output reg          cout        //1 bit carry out (output)
    );
```

```
56    wire A, B, res;
57
58    MUX2to1 A_invert(
59        .src1(src1),
60        .src2(~src1),
61        .select(Ainvert),
62        .result(A)
63    );
64    MUX2to1 B_invert(
65        .src1(src2),
66        .src2(~src2),
67        .select(Binvert),
68        .result(B)
69    );
70    MUX4to1 op(
71        .src1(A & B),
72        .src2(A | B),
73        .src3(A ^ B ^ cin),
74        .src4(less),
75        .select(operation),
76        .result(res)
77    );
78    always@(*) begin
79        result <= res;
80        cout <= (A & B) | (A & cin) | (B & cin);
81    end
82    endmodule
```

```
84  module MUX4to1(
85      input           src1,
86      input           src2,
87      input           src3,
88      input           src4,
89      input   [2-1:0] select,
90      output reg      result
91      );
92
93      always @(*) begin
94          case(select[1:0])
95              2'b00: result = src1;
96              2'b01: result = src2;
97              2'b10: result = src3;
98              2'b11: result = src4;
99          endcase
100     end
101 endmodule
102
103 module MUX2to1(
104     input      src1,
105     input      src2,
106     input      select,
107     output reg result
108     );
109     always @(*) begin
110         if (select) begin
111             result <= src2;
112         end else begin
113             result <= src1;
114         end
115     end
116 endmodule
```

4. ALU_Ctrl.v :

In case of making the output “ALU_Ctrl_o” be covered by undesired data , we first declare a register “ctl” and assign it to the original output “ALU_Ctrl_o”.

Then the main part is using ALUOP signal and function 3 to choose different operations , according to the lecture slides . 2’b00 is for S-type and addi , 2’b01 is for B-type , and 2’b10 is for R-type , then use function 3 to decide different operations in R-type . In part of R-type , although the slide doesn’t ask us to implement operations such like subtraction,and,or … directly , they’re still required for some commands , thus we also implement then . For subtraction of R-type , since (I30+fun3)[3] equals to alu_control[2] and (I30 + fun3 = fun7) , we use “ALU_Ctrl_o[2] = (instr[3]==1 ? 1:0) to distinguish it from addition .

```
1
2   `timescale 1ns/1ps
3   /*instr[30,14:12]*/
4   module ALU_Ctrl(
5       input       [4-1:0] instr,
6       input       [2-1:0] ALUOp,
7       output      [4-1:0] ALU_Ctrl_o
8   );
9   wire [2:0] func3;
10  assign func3 = instr[2:0];
11  /* Write your code HERE */
12  reg [4-1:0] ctl;
13  assign ALU_Ctrl_o = ctl;
14
15  always @(*) begin
16      if(ALUOp == 2'b00) begin            //lw,sw,addi
17          ctl <= 4'b0010;
18      end else if(ALUOp == 2'b01) begin   //beq
19          ctl <= 4'b0110;
20      end else if(ALUOp == 2'b10) begin   //R-type
21          case(func3)
22              3'b000:begin                //add
23                  ctl <= 4'b0010;
24                  ctl[2] <= (instr[3] == 1 ? 1 : 0);//sub
25              end
26              3'b111: begin               //and
27                  ctl <= 4'b0000;
28              end
29              3'b110: begin               //or
30                  ctl <= 4'b0001;
31              end
32              3'b010: begin               //slt
33                  ctl <= 4'b0111;
34              end
35              default: begin
36                  ctl <= 4'b0000;
37              end
38          endcase
39      end
40  end
41
42  endmodule
```

5. Simple Single CPU.v :

Basically consists of all the 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 whole cpu run correctly .

```
1   `timescale 1ns/1ps
2   module Simple_Single_CPU(
3       input clk_i,
4       input rst_i
5   );
6
7   //Internal Signales
8   //Control Signales
9   wire RegWrite;
10  wire Branch;
11  wire Jump;
12  wire WriteBack1;
13  wire WriteBack0;
14  wire MemRead;
15  wire MemWrite;
16  wire ALUSrcA;
17  wire ALUSrcB;
18  wire [2-1:0] ALUOp;
19  wire PCSrc;
20  //ALU Flag
21  wire Zero;
22
23  //Datapath
24  wire [32-1:0] pc_i;
25  wire [32-1:0] pc_o;
26  wire [32-1:0] instr;
27  wire [32-1:0] RegWriteData;
28  wire [32-1:0] Imm_Gen_o;
29  wire [32-1:0] Imm_4 = 4;
30  wire [4-1:0] ALUControlOut;
31  wire [4-1:0] ALUControlIn;
32  wire [32-1:0] ALUresult;
33  wire [32-1:0] RSdata_o;
34  wire [32-1:0] RTdata_o;
35  wire [32-1:0] ALUsrc2;
36  wire [32-1:0] PCsrc1;
37  wire [32-1:0] PCsrc2;
38  wire [32-1:0] PCRegsrc1;
39  wire [32-1:0] MemData_o;
40  wire [32-1:0] WriteBacksrc2;
41  assign ALUControlIn[3] = instr[30];
42  assign ALUControlIn[2:0] = instr[14:12];
43  assign PCSrc = (Branch & Zero) | Jump;
```

```
45  ProgramCounter PC(
46      .clk_i(clk_i),
47      .rst_i(rst_i),
48      .pc_i(pc_i),
49      .pc_o(pc_o)
50  );
51
52  Adder Adder_PCPlus4(
53      .src1_i(pc_o),
54      .src2_i(Imm_4),
55      .sum_o(PCsrc1)
56  );
57
58  Instr_Memory IM(
59      .addr_i(pc_o),
60      .instr_o(instr)
61  );
```

```
63  Reg_File RF(
64      .clk_i(clk_i),
65      .rst_i(rst_i),
66      .RSaddr_i(instr[19:15]),
67      .RTaddr_i(instr[24:20]),
68      .RDaddr_i(instr[11:7]),
69      .RDdata_i(RegWriteData),
70      .RegWrite_i(RegWrite),
71      .RSdata_o(RSdata_o),
72      .RTdata_o(RTdata_o)
73  );
74
75
76  Decoder Decoder(
77      .instr_i(instr[6:0]),
78      .RegWrite(RegWrite),
79      .Branch(Branch),
80      .Jump(Jump),
81      .WriteBack1(WriteBack1),
82      .WriteBack0(WriteBack0),
83      .MemRead(MemRead),
84      .MemWrite(MemWrite),
85      .ALUSrcA(ALUSrcA),
86      .ALUSrcB(ALUSrcB),
87      .ALUOp(ALUOp)
88  );
```

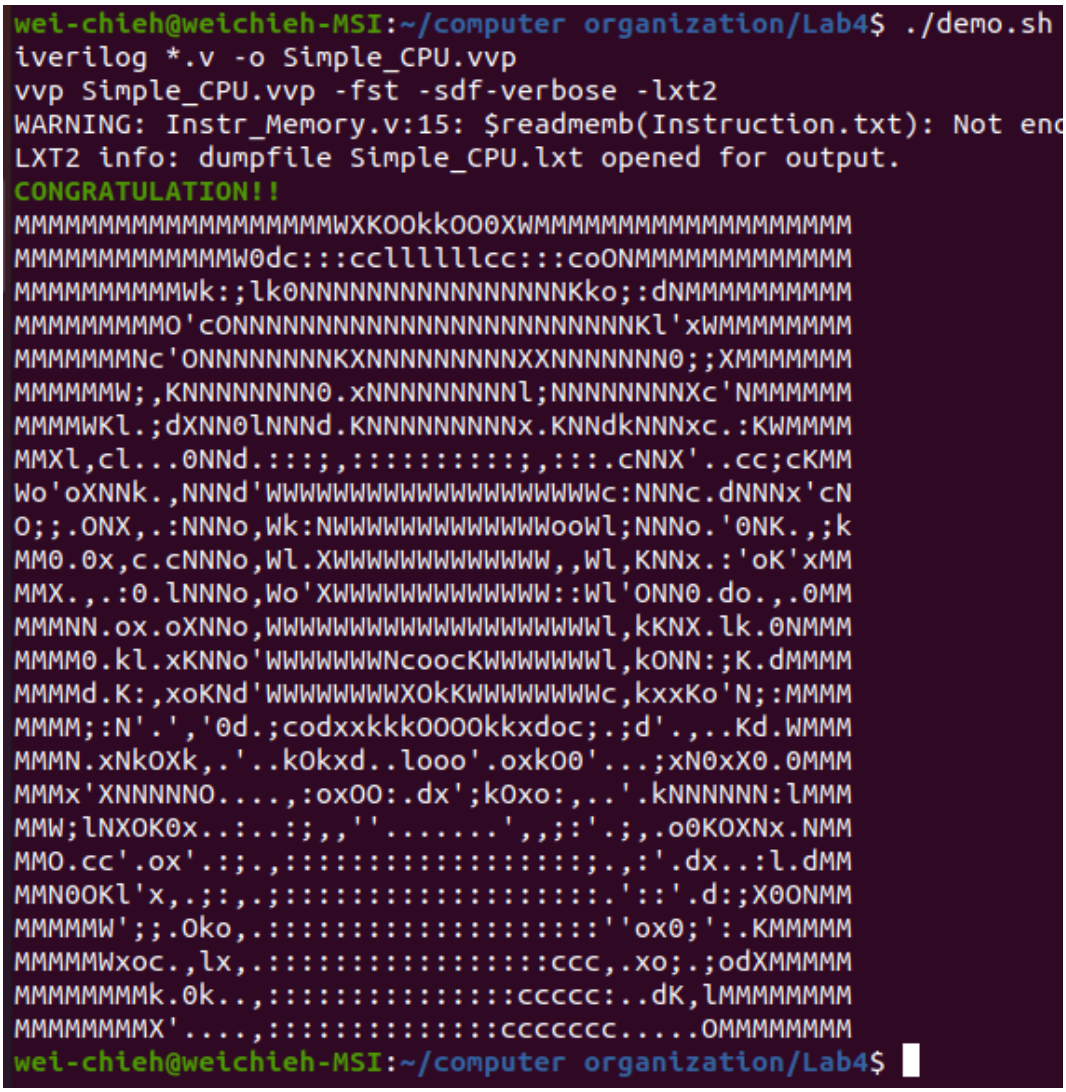
```
90   Imm_Gen ImmGen(
91       .instr_i(instr),
92       .Imm_Gen_o(Imm_Gen_o)
93   );
94
95
96   ALU_Ctrl ALU_Ctrl(
97       .instr(ALUControlIn),
98       .ALUOp(ALUOp),
99       .ALU_Ctrl_o(ALUControlOut)
100  );
101
102  MUX_2to1 MUX_ALUSrcA(
103      .data0_i(pc_o),
104      .data1_i(RSdata_o),
105      .select_i(ALUSrcA),
106      .data_o(PCRegsrc1)
107  );
```

```
109  Adder Adder_PCReg(
110      .src1_i(PCRegsrc1),
111      .src2_i(Imm_Gen_o),
112      .sum_o(PCsrc2)
113  );
114
115  MUX_2to1 MUX_PCSrc(
116      .data0_i(PCsrc1),
117      .data1_i(PCsrc2),
118      .select_i(PCSrc),
119      .data_o(pc_i)
120  );
121
122  MUX_2to1 MUX_ALUSrcB(
123      .data0_i(RTdata_o),
124      .data1_i(Imm_Gen_o),
125      .select_i(ALUSrcB),
126      .data_o(ALUsrc2)
127  );
```

```
129     alu alu(
130         .rst_n(rst_n),
131         .src1(RSdata_o),
132         .src2(ALUsrc2),
133         .ALU_control(ALUControlOut),
134         .Zero(Zero),
135         .result(ALUresult)
136     );
137
138     Data_Memory Data_Memory(
139         .clk_i(clk_i),
140         .addr_i(ALUresult),
141         .data_i(RTdata_o),
142         .MemRead_i(MemRead),
143         .MemWrite_i(MemWrite),
144         .data_o(MemData_o)
145     );
```

```
147     MUX_2to1 MUX_WriteBack0(
148         .data0_i(ALUresult),
149         .data1_i(MemData_o),
150         .select_i(WriteBack0),
151         .data_o(WriteBacksrc2)
152     );
153
154     MUX_2to1 MUX_WriteBack1(
155         .data0_i(WriteBacksrc2),
156         .data1_i(PCsrc1),
157         .select_i(WriteBack1),
158         .data_o(RegWriteData)
159     );
160
161     endmodule
```

Part II. Implementation results :



Part III. Problems encountered and solutions :

Although Lab 4 is similar to Lab 3 , there are much more details that we need to be cautious of . The most significant problem we met is that we were once unable to read new pc signals , making we have no idea about how to debug . We thought that it may result from the nop of the commands . After trying to include the conditions of it in our code , we eventually got the desired result . In addition , the structure of this cpu is much more complicated than the last one , we do have to be more careful and patient to check the original diagram and connect the datapaths as the diagram asks we to do .