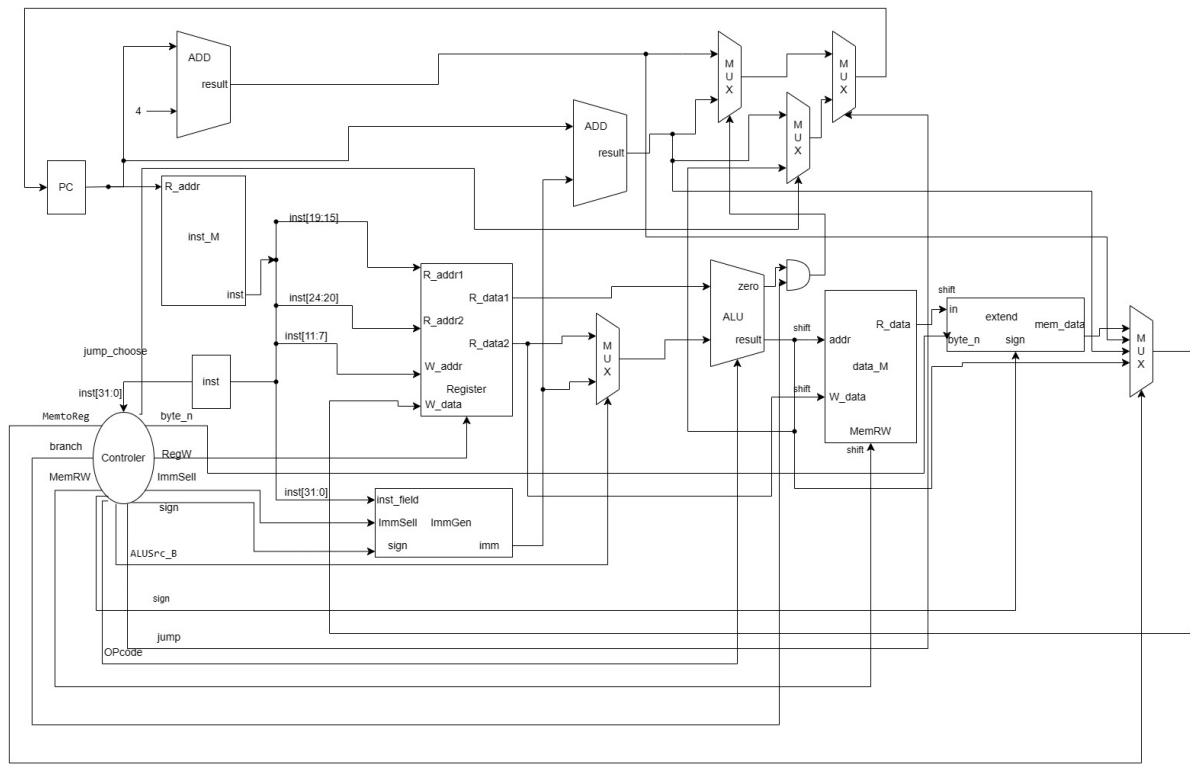


浙江大学实验报告

专业：计算机科学与技术 姓名：仇国智 学号：3220102181 日期：2024/4/10
课程名称：计算机组成与设计 实验名称:实现单周期 CPU-指令拓展 指导老师：刘海风 成绩：

操作方法与实验步骤

绘制Datapath图



源文件编写

将原来lab2的工程复制一份,删除 SCPU 模块,依据原理图,自行设计 SCPU 模块(下辖 ALU , Controller , Datapath , extend , ImmGEN , Regs 模块, ALU , Regs 已经在前面实验实现,可直接复制)

SCPU 模块

SCPU 模块主要连接 Controller 和 Datapath 模块,并且包含了 Inst 指令的寄存器(实际上应该把 Regs 和 PC 的寄存器都拉到 Datapath ,这样结构会更加清晰),还计算了 MemRW (实际上这个变量没有必要存在),同时对 Data_out 和 wea 进行了左位移偏移处理(因为 RAM 的地址是四字节对齐的,所以会对 Addr_out 造成截断, Data_out 和 wea 需要与截断后的 Addr_out 对齐,即代码 30 和 31 行).

```
1  `timescale 1ns / 1ps
2  `include "Lab4.vh"
3  module SCPU (
4      `RegFile_Regs_Outputs
5      input clk,
6      input rst,
7      input MIO_ready,
8      input [31:0] inst_in,
9      input [31:0] Data_in,
```

```

10     output CPU_MIO,
11     output MemRW,
12     output wire [31:0] PC_out,
13     output [31:0] Data_out,
14     output [31:0] Addr_out,
15     output wire [3:0] wea
16 );
17     wire [`IMM_SEL_WIDTH-1:0] ImmSel1;
18     wire ALUSrc_B;
19     wire [`MEM2REG_WIDTH-1:0] MemtoReg;
20     wire Jump;
21     wire Branch;
22     wire RegWrite;
23     wire sign;
24     wire jump_choose;
25     // 0 1byte, 1 2byte, 2 4byte
26     wire [1:0] byte_n;
27     assign MemRW = |wea;
28     wire [ 3:0] wea_temp;
29     wire [31:0] Data_out_temp;
30     assign wea = wea_temp << (Addr_out % 4);
31     assign Data_out = MemRW ? (Data_out_temp << ((Addr_out % 4) << 3)) :
Data_out_temp;
32     wire [`ALU_OP_WIDTH-1:0] ALU_Control;
33     reg [31:0] inst;
34     always @(posedge clk or posedge rst) begin
35         if (rst) begin
36             inst <= 32'h0;
37         end else begin
38             inst <= inst_in;
39         end
40     end
41     Controller v1 (
42         .OPcode(inst[6:2]),
43         .MIO_ready(MIO_ready),
44         .Fun7(inst[31:25]),
45         .Fun3(inst[14:12]),
46
47         .wea(wea_temp),
48         .CPU_MIO(CPU_MIO),
49         .ImmSel1(ImmSel1),
50         .ALUSrc_B(ALUSrc_B),
51         .MemtoReg(MemtoReg),
52         .Jump(Jump),
53         .Branch(Branch),
54         .RegWrite(RegWrite),
55         .ALU_Control(ALU_Control),
56         .sign(sign),
57         .byte_n(byte_n),
58         .jump_choose(jump_choose)
59     );
60     DataPath v2 (
61         `RegFile_Regs_Arguments
62         .clk(clk),
63         .rst(rst),
64         .inst_field(inst),

```

```

65         .Data_in(Data_in),
66         .ImmSell(ImmSell),
67         .ALUSrc_B(ALUSrc_B),
68         .MemtoReg(MemtoReg),
69         .Jump(Jump),
70         .Branch(Branch),
71         .RegWrite(RegWrite),
72         .ALU_Control(ALU_Control),
73         .sign(sign),
74         .byte_n(byte_n),
75         .jump_choose(jump_choose),
76
77         .PC_out (PC_out),
78         .Data_out(Data_out_temp),
79         .Addr_out(Addr_out)
80     );
81 endmodule

```

Controller 模块

Controller :根据指令不同要求解码即可,下面是各个输出属性的解释.代码很繁杂,或许可以通过将指令的不同输入组合然后存入一个 **ROM** 中,然后通过 **OPcode** 和 **Fun7** 和 **Fun3** 确定指令索引以读入 **ROM** 中的数据,但这样会降低代码可读性.

- **wea** : 写入地址选择
- **ImmSell** : 立即数模式选择
- **MemtoReg** : 写入寄存器数据来源选择
- **ALU_Control** : ALU控制信号
- **ALUSrc_B** : ALU第二个操作数来源选择
- **Jump** : 是否跳转
- **Branch** : 是否分支
- **RegWrite** : 是否写入寄存器
- **sign** : 字节/半字读入时是否进行有符号扩展
- **byte_n** : 读入字节数
- **jump_choose** : 跳转地址选择(PC+立即数/ALU输出)

```

1  `include "Lab4.vh"
2  module Controller (
3      input wire [4:0] OPcode,
4      input wire MIO_ready,
5      input wire [6:0] Fun7,
6      input wire [2:0] Fun3,
7      output reg CPU_MIO,
8      output reg [3:0] wea,
9      output reg [`IMM_SEL_WIDTH-1:0] ImmSell,
10     output reg [`MEM2REG_WIDTH-1:0] MemtoReg,
11     output reg [`ALU_OP_WIDTH-1:0] ALU_Control,
12     output reg ALUSrc_B,
13     output reg Jump,
14     output reg Branch,

```

```

15     output reg Regwrite,
16     output reg sign,
17     output reg [1:0] byte_n,
18     output reg jump_choose
19 );
20 always @(*) begin
21     case (OPCode)
22     `OPCODE_ALU: begin
23         CPU_MIO <= 1'b0;
24         wea <= `WEA_READ;
25         ImmSel1 <= 3'b0;
26         MemtoReg <= `MEM2REG_ALU;
27         case (Fun3)
28             `FUNC_ADD: ALU_Control <= Fun7[5] ? `ALU_OP_SUB :
`ALU_OP_ADD;
29             `FUNC_SL: ALU_Control <= `ALU_OP_SLL;
30             `FUNC_SLT: ALU_Control <= `ALU_OP_SLT;
31             `FUNC_SLTU: ALU_Control <= `ALU_OP_SLTU;
32             `FUNC_XOR: ALU_Control <= `ALU_OP_XOR;
33             `FUNC_OR: ALU_Control <= `ALU_OP_OR;
34             `FUNC_AND: ALU_Control <= `ALU_OP_AND;
35             `FUNC_SR: ALU_Control <= Fun7[5] ? `ALU_OP_SRA : `ALU_OP_SRL;
36             default: ALU_Control <= 4'b0;
37         endcase
38         ALUSrc_B <= 1'b0;
39         Jump <= 1'b0;
40         Branch <= 1'b0;
41         Regwrite <= 1'b1;
42         sign <= 1'b0;
43         byte_n <= `WORD;
44         jump_choose <= `JUMP_PC_IMM;
45     end
46     `OPCODE_ALU_IMM: begin
47         CPU_MIO <= 1'b0;
48         wea <= `WEA_READ;
49         ImmSel1 <= `IMM_SEL_I;
50         MemtoReg <= `MEM2REG_ALU;
51         case (Fun3)
52             `FUNC_ADD: ALU_Control <= `ALU_OP_ADD;
53             `FUNC_SL: ALU_Control <= `ALU_OP_SLL;
54             `FUNC_SLT: ALU_Control <= `ALU_OP_SLT;
55             `FUNC_SLTU: ALU_Control <= `ALU_OP_SLTU;
56             `FUNC_XOR: ALU_Control <= `ALU_OP_XOR;
57             `FUNC_OR: ALU_Control <= `ALU_OP_OR;
58             `FUNC_AND: ALU_Control <= `ALU_OP_AND;
59             `FUNC_SR: ALU_Control <= Fun7[5] ? `ALU_OP_SRA : `ALU_OP_SRL;
60             default: ALU_Control <= 4'b0;
61         endcase
62         ALUSrc_B <= 1'b1;
63         Jump <= 1'b0;
64         Branch <= 1'b0;
65         Regwrite <= 1'b1;
66         sign <= 1'b1;
67         byte_n <= `WORD;
68         jump_choose <= `JUMP_PC_IMM;
69     end

```

```

70     `OPCODE_LOAD: begin
71         CPU_MIO <= 1'b1;
72         wea <= `WEA_READ;
73         ImmSel1 <= `IMM_SEL_I;
74         MemtoReg <= `MEM2REG_MEM;
75         ALU_Control <= 4'b0;
76         ALUSrc_B <= 1'b1;
77         Jump <= 1'b0;
78         Branch <= 1'b0;
79         Regwrite <= 1'b1;
80         sign <= ~(Fun3 == `FUNC_BYTE_UNSIGNED || Fun3 ==
`FUNC_HALF_UNSIGNED);
81         case (Fun3)
82             `FUNC_BYTE, `FUNC_BYTE_UNSIGNED: byte_n <= `BYTE;
83             `FUNC_HALF, `FUNC_HALF_UNSIGNED: byte_n <= `HALF;
84             `FUNC_WORD: byte_n <= `WORD;
85             default: byte_n <= `WORD;
86         endcase
87         jump_choose <= `JUMP_PC_IMM;
88     end
89     `OPCODE_STORE: begin
90         CPU_MIO <= 1'b1;
91         case (Fun3)
92             `FUNC_BYTE: wea <= `WEA_BYTE;
93             `FUNC_HALF: wea <= `WEA_HALF;
94             `FUNC_WORD: wea <= `WEA_WORD;
95             default: wea <= `WEA_READ;
96         endcase
97         ImmSel1 <= `IMM_SEL_S;
98         MemtoReg <= `MEM2REG_MEM;
99         ALU_Control <= 4'b0;
100        ALUSrc_B <= 1'b1;
101        Jump <= 1'b0;
102        Branch <= 1'b0;
103        Regwrite <= 1'b0;
104        sign <= 1'b1;
105        byte_n <= `WORD;
106        jump_choose <= `JUMP_PC_IMM;
107    end
108    `OPCODE_BRANCH: begin
109        CPU_MIO <= 1'b0;
110        wea <= `WEA_READ;
111        ImmSel1 <= `IMM_SEL_B;
112        MemtoReg <= `MEM2REG_ALU;
113        case (Fun3)
114            `FUNC_EQ: ALU_Control <= `ALU_OP_SUB;
115            `FUNC_NE: ALU_Control <= `ALU_OP_EQ;
116            `FUNC_LT: ALU_Control <= `ALU_OP_SGE;
117            `FUNC_GE: ALU_Control <= `ALU_OP_SLT;
118            `FUNC_LTU: ALU_Control <= `ALU_OP_SGEU;
119            `FUNC_GEU: ALU_Control <= `ALU_OP_SLTU;
120            default: ALU_Control <= 4'b0;
121        endcase
122        ALUSrc_B <= 1'b0;
123        Jump <= 1'b0;
124        Branch <= 1'b1;

```

```

125         RegWrite <= 1'b0;
126         sign <= 1'b1;
127         byte_n <= `WORD;
128     end
129     `OPCODE_JAL: begin
130         CPU_MIO <= 1'b0;
131         wea <= `WEA_READ;
132         ImmSel1 <= `IMM_SEL_J;
133         jump_choose <= `JUMP_PC_IMM;
134         MemtoReg <= `MEM2REG_PC_PLUS;
135         ALU_Control <= `ALU_OP_ADD;
136         ALUSrc_B <= 1'b1;
137         Jump <= 1'b1;
138         Branch <= 1'b0;
139         RegWrite <= 1'b1;
140         sign <= 1'b1;
141         byte_n <= `WORD;
142     end
143     `OPCODE_JALR: begin
144         CPU_MIO <= 1'b0;
145         wea <= `WEA_READ;
146         ImmSel1 <= `IMM_SEL_I;
147         MemtoReg <= `MEM2REG_PC_PLUS;
148         ALU_Control <= `ALU_OP_ADD;
149         ALUSrc_B <= 1'b1;
150         Jump <= 1'b1;
151         Branch <= 1'b0;
152         RegWrite <= 1'b1;
153         sign <= 1'b1;
154         byte_n <= `WORD;
155         jump_choose <= `JUMP_ALU;
156     end
157     `OPCODE_LUI: begin
158         CPU_MIO <= 1'b0;
159         wea <= `WEA_READ;
160         ImmSel1 <= `IMM_SEL_U;
161         MemtoReg <= `MEM2REG_ALU;
162         ALU_Control <= `ALU_OP_R2;
163         ALUSrc_B <= 1'b1;
164         Jump <= 1'b0;
165         Branch <= 1'b0;
166         RegWrite <= 1'b1;
167         sign <= 1'b1;
168         byte_n <= `WORD;
169         jump_choose <= `JUMP_PC_IMM;
170     end
171     `OPCODE_AUIPC: begin
172         CPU_MIO <= 1'b0;
173         wea <= `WEA_READ;
174         ImmSel1 <= `IMM_SEL_U;
175         MemtoReg <= `MEM2REG_IMM_PC;
176         ALU_Control <= `ALU_OP_ADD;
177         ALUSrc_B <= 1'b1;
178         Jump <= 1'b0;
179         Branch <= 1'b0;
180         RegWrite <= 1'b1;

```

```

181         sign <= 1'b1;
182         byte_n <= `WORD;
183         jump_choose <= `JUMP_PC_IMM;
184     end
185     `OPCODE_PASS: begin
186         CPU_MIO <= 1'b0;
187         wea <= `WEA_READ;
188         ImmSel1 <= `IMM_SEL_I;
189         MemtoReg <= `MEM2REG_ALU;
190         ALU_Control <= 4'b0;
191         ALUSrc_B <= 1'b0;
192         Jump <= 1'b0;
193         Branch <= 1'b0;
194         Regwrite <= 1'b0;
195         sign <= 1'b1;
196         byte_n <= `WORD;
197         jump_choose <= `JUMP_PC_IMM;
198     end
199 endcase
200 end
201 endmodule

```

extend 模块

extend :根据是否有符号 **sign** 和读取字节数 **byte_n** 来生成读取数据的32位扩展形式

```

1  `include "Lab4.vh"
2  module extend (
3      input wire [1:0] byte_n,
4      input wire [31:0] in,
5      input wire sign,
6      output [31:0] mem_data
7  );
8      assign mem_data=byte_n==`WORD?in:
9          byte_n==`HALF?(sign?{{16{in[15]}},in[15:0]}:
10         {16'b0,in[15:0]}):
11         byte_n==`BYTE?(sign?{{24{in[7]}},in[7:0]}:{24'b0,in[7:0]}):
12         :32'b0;
13 endmodule

```

1. **ImmGEN**:根据指令类型 **ImmSel1** 进行立即数扩展.

```

1  `include "Lab4.vh"
2  module ImmGen (
3      input wire [`IMM_SEL_WIDTH-1:0] ImmSel1,
4      input wire [31:0] inst_field,
5      input wire sign,
6      output wire [31:0] Imm
7  );
8      wire [31:0] I_Imm, S_Imm, B_Imm, J_Imm, U_Imm;
9      assign I_Imm = sign ? {{20{inst_field[31]}}, inst_field[31:20]} : {20'b0,
10     inst_field[31:20]};
11     assign S_Imm = sign?{{20{inst_field[31]}}, inst_field[31:25],
12     inst_field[11:7]}:{20'b0, inst_field[31:25], inst_field[11:7]};

```

```

11     assign B_Imm = sign?{{19{inst_field[31]}}}, inst_field[31], inst_field[7],
inst_field[30:25], inst_field[11:8], 1'b0}:{19'b0, inst_field[31],
inst_field[7], inst_field[30:25], inst_field[11:8], 1'b0};

12
13     assign J_Imm = sign?{{11{inst_field[31]}}}, inst_field[19:12],
inst_field[20], inst_field[30:21], 1'b0:
{10'b0,inst_field[31],inst_field[19:12], inst_field[20], inst_field[30:21],
1'b0};

14     assign U_Imm = {inst_field[31:12], 12'b0};
15     assign Imm = (ImmSel1 == `IMM_SEL_I) ? I_Imm :
16                 (ImmSel1 == `IMM_SEL_S) ? S_Imm :
17                 (ImmSel1 == `IMM_SEL_B) ? B_Imm :
18                 (ImmSel1 == `IMM_SEL_J) ? J_Imm :
19                 (ImmSel1 == `IMM_SEL_U) ? U_Imm : 32'b0;
20 endmodule

```

ALU 模块

ALU:修改原 ALU 模块增添了若干运算.

```

1  `timescale 1ns / 1ps
2  module ALU (
3      input  [31:0] A,
4      input  [31:0] B,
5      input  [ 3:0] ALU_operation,
6      output [31:0] res,
7      output          zero
8  );
9      wire signed [31:0] A_s = $signed(A);
10     wire signed [31:0] B_s = $signed(B);
11     wire [31:0] A_u = $unsigned(A);
12     wire [31:0] B_u = $unsigned(B);
13     wire [31:0] result0 = A_s + B_s;
14     wire [31:0] result1 = A_s - B_s;
15     wire [31:0] result2 = A << B[4:0];
16     wire [31:0] result3 = (A_s < B_s) ? 32'b1 : 32'b0;
17     wire [31:0] result4 = (A_u < B_u) ? 32'b1 : 32'b0;
18     wire [31:0] result5 = A ^ B;
19     wire [31:0] result6 = A >> B[4:0];
20     wire [31:0] result7 = A_s >>> B_s[4:0];
21     wire [31:0] result8 = A | B;
22     wire [31:0] result9 = A & B;
23     wire [31:0] result10 = ~|result1;
24     wire [31:0] result11 = ~|result3;
25     wire [31:0] result12 = ~|result4;
26     wire [31:0] result13 = B;
27     assign res = (ALU_operation==4'b0000)?result0:
28                 (ALU_operation==4'b0001)?result1:
29                 (ALU_operation==4'b0010)?result2:
30                 (ALU_operation==4'b0011)?result3:
31                 (ALU_operation==4'b0100)?result4:
32                 (ALU_operation==4'b0101)?result5:
33                 (ALU_operation==4'b0110)?result6:
34                 (ALU_operation==4'b0111)?result7:
35                 (ALU_operation==4'b1000)?result8:

```



```

36         (ALU_operation==4'b1001)?result9:
37         (ALU_operation==4'b1010)?result10:
38         (ALU_operation==4'b1011)?result11:
39         (ALU_operation==4'b1100)?result12:
40         (ALU_operation==4'b1101)?result13:
41         32'b0;
42     assign zero = ~(|res) ? 1'b1 : 1'b0;
43 endmodule

```

Datapath 模块

Datapath:将各个部件连接起来,实际上CPU的核心部件,决定了CPU运作的逻辑,代码 1 到 68 行主要是部件之间的简单连接,下面分析一下复杂逻辑部分(其中要注意 extend 模块的 in 即内存读入需要进行便宜操作以便和地址对齐,原因同上).

- 通过 ALUSrc_B 选择 ALU 的第二个操作数

```

1 | assign adder_2=ALUSrc_B?imm:Rs2_data;

```

- 进行下一条指令地址 PC_out 的选择
 - PC_temp:用于存储临时的程序计数器值.
 - PC_add_4:等于 PC_temp 加 4,即不进行跳转,正常下一条指令
 - PC_imm:通过 PC 相对偏移量进行跳转
 - Branch_final:等于 Branch 信号和 ALU 模块计算为零 zero 信号的逻辑与,这用于判断是否需要进行条件分支跳转.
 - PC_branch:根据 Branch_final 的值选择 PC_imm 或 PC_add_4
 - PC_jump:根据 Jump 信号的值选择 PC_branch 或者根据 jump_choose 的值选择 ALU_out 或 PC_imm.这用于处理无条件跳转的情况.

```

1 | wire[31:0] PC_temp;
2 | wire[31:0] PC_add_4=PC_temp+4;
3 | wire[31:0] PC_imm=imm+PC_temp;
4 | wire Branch_final=Branch&zero;
5 | wire[31:0] PC_branch=Branch_final?PC_imm:PC_add_4;
6 | wire[31:0] PC_jump=Jump?((jump_choose==`JUMP_ALU)?
    ALU_out:PC_imm):PC_branch;

```

- 根据 MemtoReg 选择写入寄存器的源数据

```

1 | assign w_data=MemtoReg==`MEM2REG_MEM?mem_out:
2 |         MemtoReg==`MEM2REG_ALU?ALU_out:
3 |         MemtoReg==`MEM2REG_PC_PLUS?PC_add_4:
4 |         MemtoReg==`MEM2REG_IMM_PC?PC_imm:32'b0;

```

下面是完整代码

```

1 | `include "Lab4.vh"
2 | module DataPath (
3 |     `RegFile_Regs_Outputs
4 |     input wire clk,

```

```

5     input wire rst,
6     input wire [31:0] inst_field,
7     input wire [31:0] Data_in,
8     input wire [`IMM_SEL_WIDTH-1:0] ImmSel1,
9     input wire ALUSrc_B,
10    input wire [`MEM2REG_WIDTH-1:0] MemtoReg,
11    input wire Jump,
12    input wire Branch,
13    input wire RegWrite,
14    input wire [`ALU_OP_WIDTH-1:0] ALU_Control,
15    input wire sign,
16    input wire[1:0] byte_n,
17    input wire jump_choose,
18    output wire [31:0] PC_out,
19    output wire [31:0] Data_out,
20    output wire [31:0] Addr_out
21 );
22 reg[31:0] PC=32'h4;
23 wire[31:0] imm;
24 ImmGen U1 (
25     .ImmSel1(ImmSel1),
26     .inst_field(inst_field),
27     .sign(1'b1),
28     .Imm(imm)
29 );
30 wire [31:0] Rs1_data, Rs2_data;
31 wire [4:0] Rs1_addr, Rs2_addr, W_addr;
32 wire[31:0]W_data;
33 assign Rs1_addr = inst_field[19:15];
34 assign Rs2_addr = inst_field[24:20];
35 assign W_addr = inst_field[11:7];
36 Regs U2 (
37     `RegFile_Regs_Arguments
38     .clk(clk),
39     .rst(rst),
40     .Rs1_addr(Rs1_addr),
41     .Rs2_addr(Rs2_addr),
42     .Wt_addr(W_addr),
43     .Wt_data(W_data),
44     .RegWrite(RegWrite),
45     .Rs1_data(Rs1_data),
46     .Rs2_data(Rs2_data)
47 );
48 wire[31:0] adder_2;
49 wire [31:0] ALU_out;
50 wire zero;
51 ALU U3 (
52     .A(Rs1_data),
53     .B(adder_2),
54     .ALU_operation(ALU_Control),
55     .res(ALU_out),
56     .zero(zero)
57 );
58 assign Addr_out=ALU_out;
59 assign Data_out=Rs2_data;
60 wire [31:0] mem_out;

```

```

61
62 extend U4(
63     .byte_n(byte_n),
64     .in(Data_in>>((Addr_out%4)<<3)),
65     .sign(sign),
66     .mem_data(mem_out)
67 );
68
69 assign adder_2=ALUSrc_B?imm:Rs2_data;
70
71 wire[31:0] PC_temp;
72 wire[31:0] PC_add_4=PC_temp+4;
73 wire[31:0] PC_imm=imm+PC_temp;
74 wire Branch_final=Branch&zero;
75 wire[31:0] PC_branch=Branch_final?PC_imm:PC_add_4;
76 wire[31:0] PC_jump=Jump?((jump_choose==`JUMP_ALU)?ALU_out:PC_imm):PC_branch;
77 always@(posedge clk or posedge rst)
78 begin
79     if(rst)
80     begin
81         PC<=32'hFFFFFFC;
82     end
83     else
84     begin
85         PC<=PC_jump;
86     end
87 end
88 assign PC_temp=PC;
89 assign PC_out=PC_jump;
90 assign w_data=MementoReg==`MEM2REG_MEM?mem_out:
91     MementoReg==`MEM2REG_ALU?ALU_out:
92     MementoReg==`MEM2REG_PC_PLUS?PC_add_4:
93     MementoReg==`MEM2REG_IMM_PC?PC_imm:32'b0;
94
95 endmodule

```

头文件宏定义

下面是一些我修改过的头文件宏定义

```

1  /* WHAT'S THIS HEADERFILE FOR? */
2  /*
3      * Reffered to code written by PanZiyue, TA of 2020_CO
4      * Macro for opcode/func3 for RV32I
5      * declaration, inputs/outputs, assignment for debug signals(RegFile)
6  */
7  /* wea */
8  `define WEA_READ 4'b0000
9  `define WEA_BYTE 4'b0001
10 `define WEA_HALF 4'b0011
11 `define WEA_WORD 4'b1111
12 /*JUMP CHOOSE*/
13 `define JUMP_PC_IMM 1'b0
14 `define JUMP_ALU 1'b1
15 /* Byte/Half/word */

```

```

16 `define BYTE 2'b00
17 `define HALF 2'b01
18 `define WORD 2'b10
19 /* ALU Operation(Using in Lab1) */
20 `define ALU_OP_WIDTH 4
21
22 `define ALU_OP_ADD `ALU_OP_WIDTH'd0
23 `define ALU_OP_SUB `ALU_OP_WIDTH'd1
24 `define ALU_OP_SLL `ALU_OP_WIDTH'd2
25 `define ALU_OP_SLT `ALU_OP_WIDTH'd3
26 `define ALU_OP_SLTU `ALU_OP_WIDTH'd4
27 `define ALU_OP_XOR `ALU_OP_WIDTH'd5
28 `define ALU_OP_SRL `ALU_OP_WIDTH'd6
29 `define ALU_OP_SRA `ALU_OP_WIDTH'd7
30 `define ALU_OP_OR `ALU_OP_WIDTH'd8
31 `define ALU_OP_AND `ALU_OP_WIDTH'd9
32 `define ALU_OP_EQ `ALU_OP_WIDTH'd10
33 `define ALU_OP_SGE `ALU_OP_WIDTH'd11
34 `define ALU_OP_SGEU `ALU_OP_WIDTH'd12
35 `define ALU_OP_R2 `ALU_OP_WIDTH'd13
36
37 /*-----*/
38
39 /* Inst decoding(Using in Lab4/5) */
40 /* Opcode(5-bits) */
41 // R-Type
42 `define OP_CODE_ALU 5'b01100
43 // I-Type
44 `define OP_CODE_ALU_IMM 5'b00100
45 `define OP_CODE_LOAD 5'b00000
46 `define OP_CODE_JALR 5'b11001
47 `define OP_CODE_ENV 5'b11100
48 `define OP_CODE_JALR 5'b11001
49 // S-Type
50 `define OP_CODE_STORE 5'b01000
51 // B-Type
52 `define OP_CODE_BRANCH 5'b11000
53 // J-Type
54 `define OP_CODE_JAL 5'b11011
55 // U-Type
56 `define OP_CODE_LUI 5'b01101
57 `define OP_CODE_AUIPC 5'b00101
58
59 // not use
60 `define OP_CODE_PASS 5'b00000
61
62 /* Func3(3-bits) */
63 // R-Type & I-Type(ALU)
64 // For R-Type, SUB if inst[30] else ADD
65 `define FUNC_ADD 3'd0
66 // Shift Left (Logical)
67 `define FUNC_SL 3'd1
68 `define FUNC_SLT 3'd2
69 `define FUNC_SLTU 3'd3
70 `define FUNC_XOR 3'd4
71 // Shift Right Arith if inst[30] else Logical

```

```

72 `define FUNC_SR          3'd5
73 `define FUNC_OR          3'd6
74 `define FUNC_AND        3'd7
75
76 // I-Type(Load) & S-Type
77 `define FUNC_BYTE        3'd0
78 `define FUNC_HALF        3'd1
79 `define FUNC_WORD        3'd2
80 `define FUNC_BYTE_UNSIGNED 3'd4
81 `define FUNC_HALF_UNSIGNED 3'd5
82
83 // B-Type
84 `define FUNC_EQ          3'd0
85 `define FUNC_NE          3'd1
86 `define FUNC_LT          3'd4
87 `define FUNC_GE          3'd5
88 `define FUNC_LTU         3'd6
89 `define FUNC_GEU         3'd7
90 /*-----*/
91
92 // JAR
93 `define FUNC_JALR        3'd0
94 /* ImmSel signals */
95 // NOTE: You may add terms in Lab4-3 to implement more inst.
96 `define IMM_SEL_WIDTH 3
97
98 `define IMM_SEL_I `IMM_SEL_WIDTH'd0
99 `define IMM_SEL_S `IMM_SEL_WIDTH'd1
100 `define IMM_SEL_B `IMM_SEL_WIDTH'd2
101 `define IMM_SEL_J `IMM_SEL_WIDTH'd3
102 `define IMM_SEL_U `IMM_SEL_WIDTH'd4
103 /*-----*/
104
105 /* Mem2Reg signals */
106 // NOTE: You may add terms in Lab4-3 to implement more inst.
107 `define MEM2REG_WIDTH 2
108
109 `define MEM2REG_ALU `MEM2REG_WIDTH'd0
110 `define MEM2REG_MEM `MEM2REG_WIDTH'd1
111 `define MEM2REG_PC_PLUS `MEM2REG_WIDTH'd2
112 `define MEM2REG_IMM_PC `MEM2REG_WIDTH'd3
113 // `define MEM2REG_IMM `MEM2REG_WIDTH'd4
114 `define YOUR_REGS MY_REGS

```

仿真

SCPU 模块仿真

仿真激励代码 `SCPU_tb` 如下:

```

1 module SCPU_tb ();
2     reg clk;
3     reg rst;
4     reg [31:0] inst_in;
5     reg [31:0] Data_in;

```

```

6  wire MemRW;
7  wire [31:0] PC_out; // Next PC
8  wire [31:0] Data_out; //Rs2 or Imm
9  wire [31:0] Addr_out; //ALU_out
10 wire [3:0] wea;
11 SCPU SCPU (
12     .clk(clk),
13     .rst(rst),
14     .inst_in(inst_in),
15     .Data_in(Data_in),
16     .MemRW(MemRW),
17     .PC_out(PC_out),
18     .Data_out(Data_out),
19     .Addr_out(Addr_out),
20     .wea(wea)
21 );
22 always begin
23     #5 clk = ~clk;
24 end
25 //      # U型指令
26 //      lui x1, 0x625          # 加载上位立即数到0x00625000
27 //      auipc x2, 0xA38        # 加载到程序计数器上位立即数到0x00A38000
28
29 //      # 基础算术逻辑指令
30 //      add x3, x1, x2          # x3 = x1 + x2
31 //      sub x4, x3, x1          # x4 = x3 - x1
32 //      xor x5, x4, x3          # x5 = x4 ^ x3
33 //      or  x6, x5, x4          # x6 = x5 | x4
34 //      and x7, x6, x5          # x7 = x6 & x5
35 //      sll x8, x7, x1          # x8 = x7 << x1
36 //      srl x9, x8, x2          # x9 = x8 >> x2
37 //      sra x10, x9, x1         # x10 = x9 >> x1 (arithmetic)
38 //      slt x11,x10,x9
39 //      sltu x12,x11,x10
40
41 //      # 立即操作指令
42 //      addi x11, x10, 0x10 # x11 = x10 + 0x10
43 //      xori x12, x11, 0xFF # x12 = x11 ^ 0xFF
44 //      ori  x13, x12, 0x1F # x13 = x12 | 0x1F
45 //      andi x14, x13, 0x3F # x14 = x13 & 0x3F
46 //      slli x15, x14, 0x2  # x15 = x14 << 0x2
47 //      srli x16, x15, 0x2  # x16 = x15 >> 0x2
48 //      srai x17, x16, 0x2  # x17 = x16 >> 0x2 (arithmetic)
49 //      slti x18,x17,-1
50 //      sltiu x18,x17,-1
51
52 //      # 分支跳转与链接指令
53 //      jal x18, 76          # 跳转到标签end, 并将返回地址保存到x18
54 //      jalr x19, x18, 2     # 通过x18跳到返回地址, 保存下一条指令地址到x19
55
56 //      # 内存加载与存储指令
57 //      lb x20,5(x1)         # 从x1 + arr加载一个字节到x20
58 //      lh x21, 5(x1)        # 加载半字
59 //      lw x22, 4(x1)        # 加载字
60 //      lbu x23, 5(x1)       # 无符号加载字节
61 //      lhu x24, 5(x1)       # 无符号加载半字

```

```

62      //      sb x20, 5(x1)          # 将x20的最低字节存回内存
63      //      sh x21, 5(x1)          # 存储半字
64      //      sw x22, 4(x1)          # 存储字
65
66      //      # 分支指令
67      //      beq x21, x22, 28
68      //      bne x21, x22, 28
69      //      blt x21, x22, 24
70      //      bge x21, x22, 16
71      //      bltu x21, x22, 16
72      //      bgeu x21, x22, 8
73      // 006250B7
74      // 00A38117
75      // 002081B3
76      // 40118233
77      // 003242B3
78      // 0042E333
79      // 005373B3
80      // 00139433
81      // 002454B3
82      // 4014D533
83      // 009525B3
84      // 00A5B633
85      // 01050593
86      // 0FF5C613
87      // 01F66693
88      // 03F6F713
89      // 00271793
90      // 0027D813
91      // 40285893
92      // FFF8A913
93      // FFF8B913
94      // 04C0096F
95      // 002909E7
96      // 00508A03
97      // 00509A83
98      // 0040AB03
99      // 0050CB83
100     // 0050DC03
101     // 014082A3
102     // 015092A3
103     // 0160A223
104     // 016A8E63
105     // 016A9E63
106     // 016ACC63
107     // 016AD863
108     // 016AE863
109     // 016AF463
110     initial begin
111         clk = 0;
112         rst = 0;
113         inst_in = 0;
114         Data_in = 0;
115         #10 rst = 1;
116         #10 rst = 0;
117         // # U型指令

```

```

118 // lui x1, 1573          # 加载上位立即数到0x00625000
119 inst_in = 32'h006250B7;
120 // MemRW = 0;
121 // PC_out = h00000004;
122 // Data_out = h00000000;
123 // Addr_out = h00625000;
124 // wea = 4'b0000;
125
126 // auipc x2, 0xA38        # 加载到程序计数器上位立即数到0x00A38000
127 #10 inst_in = 32'h00A38117;
128 // MemRW = 0;
129 // PC_out = h00000008;
130 // Data_out = h00000000;
131 // Addr_out = h00A38000;
132 // wea = 4'b0000;
133
134 // # 基础算术逻辑指令
135 // add x3, x1, x2         # x3 = x1 + x2
136 #10 inst_in = 32'h002081B3;
137 // MemRW = 0;
138 // PC_out = h0000000C;
139 // Data_out = h00a38004;
140 // Addr_out = h0105d004;
141 // wea = 4'b0000;
142
143 // sub x4, x3, x1         # x4 = x3 - x1
144 #10 inst_in = 32'h40118233;
145 // MemRW = 0;
146 // PC_out = h00000010;
147 // Data_out = h00625000;
148 // Addr_out = h00a38004;
149 // wea = 4'b0000;
150
151 // xor x5, x4, x3         # x5 = x4 ^ x3
152 #10 inst_in = 32'h003242B3;
153 // MemRW = 0;
154 // PC_out = h00000014;
155 // Data_out = h0105d004;
156 // Addr_out = h01a65000;
157 // wea = 4'b0000;
158
159 // or x6, x5, x4          # x6 = x5 | x4
160 #10 inst_in = 32'h0042E333;
161 // MemRW = 0;
162 // PC_out = h00000018;
163 // Data_out = h00a38004;
164 // Addr_out = h01a7d004;
165 // wea = 4'b0000;
166
167 // and x7, x6, x5         # x7 = x6 & x5
168 #10 inst_in = 32'h005373B3;
169 // MemRW = 0;
170 // PC_out = h0000001C;
171 // Data_out = h01a65000;
172 // Addr_out = h01a65000;
173 // wea = 4'b0000;

```



```

174
175 // sll x8, x7, x1      # x8 = x7 << x1
176 #10 inst_in = 32'h00139433;
177 // MemRW = 0;
178 // PC_out = h00000020;
179 // Data_out = h00625000;
180 // Addr_out = h01a65000;
181 // wea = 4'b0000;
182
183 // srl x9, x8, x2      # x9 = x8 >> x2
184 #10 inst_in = 32'h002454B3;
185 // MemRW = 0;
186 // PC_out = h00000024;
187 // Data_out = h00a38004;
188 // Addr_out = h001a6500;
189 // wea = 4'b0000;
190
191 // sra x10, x9, x1     # x10 = x9 >> x1 (arithmetic)
192 #10 inst_in = 32'h4014D533;
193 // MemRW = 0;
194 // PC_out = h00000028;
195 // Data_out = h00625000;
196 // Addr_out = h001a6500;
197 // wea = 4'b0000;
198
199 // slt x11,x10,x9      # x11 = x10 < x9
200 #10 inst_in = 32'h009525B3;
201 // MemRW = 0;
202 // PC_out = h0000002C;
203 // Data_out = h001a6500;
204 // Addr_out = h00000000;
205 // wea = 4'b0000;
206
207 // sltu x12,x11,x10    # x12 = x11 < x10
208 #10 inst_in = 32'h00A5B633;
209 // MemRW = 0;
210 // PC_out = h00000030;
211 // Data_out = h001a6500;
212 // Addr_out = h00000001;
213 // wea = 4'b0000;
214
215 // # 立即操作指令
216 // addi x11, x10, 0x10 # x11 = x10 + 0x10
217 #10 inst_in = 32'h01050593;
218 // MemRW = 0;
219 // PC_out = h00000034;
220 // Data_out = h00000000;
221 // Addr_out = h001a6510;
222 // wea = 4'b0000;
223
224 // xori x12, x11, 0xFF # x12 = x11 ^ 0xFF
225 #10 inst_in = 32'h0FF5C613;
226 // MemRW = 0;
227 // PC_out = h00000038;
228 // Data_out = h00000000;
229 // Addr_out = h001a65ef;

```

```

230 // wea = 4'b0000;
231
232 // ori x13, x12, 0x1F # x13 = x12 | 0x1F
233 #10 inst_in = 32'h01F66693;
234 // MemRW = 0;
235 // PC_out = h0000003c;
236 // Data_out = h00000000;
237 // Addr_out = h001a65ff;
238 // wea = 4'b0000;
239
240 // andi x14, x13, 0x3F # x14 = x13 & 0x3F
241 #10 inst_in = 32'h03F6F713;
242 // MemRW = 0;
243 // PC_out = h00000040;
244 // Data_out = h00000000;
245 // Addr_out = h0000003f;
246 // wea = 4'b0000;
247
248 // slli x15, x14, 0x2 # x15 = x14 << 0x2
249 #10 inst_in = 32'h00271793;
250 // MemRW = 0;
251 // PC_out = h00000044;
252 // Data_out = h00a38004;
253 // Addr_out = h000000fc;
254 // wea = 4'b0000;
255
256 // srli x16, x15, 0x2 # x16 = x15 >> 0x2
257 #10 inst_in = 32'h0027D813;
258 // MemRW = 0;
259 // PC_out = h00000048;
260 // Data_out = h00a38004;
261 // Addr_out = h0000003f;
262 // wea = 4'b0000;
263
264 // srai x17, x16, 0x2 # x17 = x16 >> 0x2 (arithmetic)
265 #10 inst_in = 32'h40285893;
266 // MemRW = 0;
267 // PC_out = h0000004c;
268 // Data_out = h00a38004;
269 // Addr_out = h0000000f;
270 // wea = 4'b0000;
271
272 // slti x18,x17,-1 # x18 = x17 < -1
273 #10 inst_in = 32'hFFF8A913;
274 // MemRW = 0;
275 // PC_out = h00000050;
276 // Data_out = h00000000;
277 // Addr_out = h00000000;
278 // wea = 4'b0000;
279
280 // sltiu x18,x17,-1 # x18 = x17 < -1
281 #10 inst_in = 32'hFFF8B913;
282 // MemRW = 0;
283 // PC_out = h00000054;
284 // Data_out = h00000000;
285 // Addr_out = h00000001;

```

```

286 // wea = 4'b0000;
287
288 // # 分支跳转与链接指令
289 // jal x18, 76 # 跳转到标签end, 并将返回地址保存到x18
290 #10 inst_in = 32'h04C0096F;
291 // MemRW = 0;
292 // PC_out = h000000a0;
293 // Data_out = h001a6500;
294 // Addr_out = h0000004c;
295 // wea = 4'b0000;
296
297 // jalr x19, x18, 2 # 通过x18跳到返回地址, 保存下一条指令地址到x19
298 #10 inst_in = 32'h002909E7;
299 // MemRW = 0;
300 // PC_out = h0000005a;
301 // Data_out = h00a38004;
302 // Addr_out = h0000005a;
303 // wea = 4'b0000;
304
305 // # 内存加载与存储指令
306 // lb x20, 5(x1) # 从x1 + arr加载一个字节到x20
307 #10 inst_in = 32'h00508A03;
308 Data_in = 32'h0000FF01;
309 // MemRW = 1;
310 // PC_out = h0000005e;
311 // Data_out = h01a65000;
312 // Addr_out = h00625005;
313 // wea = 4'b0000;
314
315 // lh x21, 5(x1) # 加载半字
316 #10 inst_in = 32'h00509A83;
317 // MemRW = 1;
318 // PC_out = h00000062;
319 // Data_out = h01a65000;
320 // Addr_out = h00625005;
321 // wea = 4'b0000;
322
323 // lw x22, 4(x1) # 加载字
324 #10 inst_in = 32'h0040AB03;
325 // MemRW = 1;
326 // PC_out = h00000066;
327 // Data_out = h00a38004;
328 // Addr_out = h00625004;
329 // wea = 4'b0000;
330
331 // lbu x23, 5(x1) # 无符号加载字节
332 #10 inst_in = 32'h0050CB83;
333 // MemRW = 1;
334 // PC_out = h0000006a;
335 // Data_out = h01a65000;
336 // Addr_out = h00625005;
337 // wea = 4'b0000;
338
339 // lhu x24, 5(x1) # 无符号加载半字
340 #10 inst_in = 32'h0050DC03;
341 // MemRW = 1;

```

```

342 // PC_out = h0000006e;
343 // Data_out = h01a65000;
344 // Addr_out = h00625005;
345 // wea = 4'b0000;
346
347 // sb x20, 5(x1)          # 将x20的最低字节存回内存
348 #10 inst_in = 32'h014082A3;
349 // MemRW = 1;
350 // PC_out = h00000072;
351 // Data_out = hfffffff00;
352 // Addr_out = h00625005;
353 // wea = 4'b0010;
354
355 // sh x21, 5(x1)          # 存储半字
356 #10 inst_in = 32'h015092A3;
357 // MemRW = 1;
358 // PC_out = h00000076;
359 // Data_out = 0000ff00;
360 // Addr_out = h00625005;
361 // wea = 4'b0110;
362
363 // sw x22, 4(x1)          # 存储字
364 #10 inst_in = 32'h0160A223;
365 // MemRW = 1;
366 // PC_out = h0000007a;
367 // Data_out = h0000ff01;
368 // Addr_out = h00625004;
369 // wea = 4'b1111;
370
371 // # 分支指令
372 // beq x21, x22, 28
373 #10 inst_in = 32'h016A8E63;
374 // MemRW = 0;
375 // PC_out = h0000007e;
376 // Data_out = h0000ff01;
377 // Addr_out = hffff01fe;
378 // wea = 4'b0000;
379
380 // bne x21, x22, 28
381 #10 inst_in = 32'h016A9E63;
382 // MemRW = 0;
383 // PC_out = h0000009a;
384 // Data_out = h0000ff01;
385 // Addr_out = h00000000;
386 // wea = 4'b0000;
387
388 // blt x21, x22, 24
389 #10 inst_in = 32'h016ACC63;
390 // MemRW = 0;
391 // PC_out = h000000b2;
392 // Data_out = h0000ff01;
393 // Addr_out = h00000000;
394 // wea = 4'b0000;
395
396 // bge x21, x22, 16
397 #10 inst_in = 32'h016AD863;

```

```

398 // MemRW = 0;
399 // PC_out = h000000b6;
400 // Data_out = h0000ff01;
401 // Addr_out = h00000001;
402 // wea = 4'b0000;
403
404 // bltu x21, x22, 16
405 #10 inst_in = 32'h016AE863;
406 // MemRW = 0;
407 // PC_out = h000000c6;
408 // Data_out = h0000ff01;
409 // Addr_out = h00000000;
410 // wea = 4'b0000;
411
412 // bgeu x21, x22, 8
413 #10 inst_in = 32'h016AF463;
414 // MemRW = 0;
415 // PC_out = h000000ca;
416 // Data_out = h0000ff01;
417 // Addr_out = h00000001;
418 #10;
419 $finish();
420
421 end
422 endmodule

```

仿真结果如下



相对时间	指令 (inst_in)	MemRW	PC_out	Data_out	Addr_out	wea
240	32'h00509A83	1	h00000062	h01a65000	h00625005	0000
250	32'h0040AB03	1	h00000066	h00a38004	h00625004	0000
260	32'h0050CB83	1	h0000006a	h01a65000	h00625005	0000
270	32'h0050DC03	1	h0000006e	h01a65000	h00625005	0000
280	32'h014082A3	1	h00000072	hffffff00	h00625005	0010
290	32'h015092A3	1	h00000076	0000ff00	h00625005	0110
300	32'h0160A223	1	h0000007a	h0000ff01	h00625004	1111
310	32'h016A8E63	0	h0000007e	h0000ff01	hffff01fe	0000
320	32'h016A9E63	0	h0000009a	h0000ff01	h00000000	0000
330	32'h016ACC63	0	h000000b2	h0000ff01	h00000000	0000
340	32'h016AD863	0	h000000b6	h0000ff01	h00000001	0000
350	32'h016AE863	0	h000000c6	h0000ff01	h00000000	0000
360	32'h016AF463	0	h000000ca	h0000ff01	h00000001	0000

datapath 模块仿真

如下是仿真代码

```

1  // Testbench for DataPath module
2  `timescale 1ns / 1ps
3  `include "../sources_1/new/Lab4.vh"
4  module DataPath_tb;
5
6      // Inputs
7      reg clk = 1'b0;
8      reg rst;
9      reg [31:0] inst_field;
10     reg [31:0] Data_in;
11     reg [`IMM_SEL_WIDTH-1:0] ImmSel1;
12     reg ALUSrc_B;
13     reg [`MEM2REG_WIDTH-1:0] MemtoReg;
14     reg Jump;
15     reg Branch;
16     reg Regwrite;
17     reg [`ALU_OP_WIDTH-1:0] ALU_Control;
18     reg sign;
19     reg [1:0] byte_n;
20     reg jump_choose;
21
22     // Outputs
23     wire [31:0] PC_out;
24     wire [31:0] Data_out;
25     wire [31:0] Addr_out;

```

```

26
27 // Instantiate the Unit Under Test (UUT)
28 DataPath uut (
29     .clk(clk),
30     .rst(rst),
31     .inst_field(inst_field),
32     .Data_in(Data_in),
33     .ImmSel1(ImmSel1),
34     .ALUSrc_B(ALUSrc_B),
35     .MemtoReg(MemtoReg),
36     .Jump(Jump),
37     .Branch(Branch),
38     .RegWrite(RegWrite),
39     .ALU_Control(ALU_Control),
40     .sign(sign),
41     .byte_n(byte_n),
42     .jump_choose(jump_choose),
43     .PC_out(PC_out),
44     .Data_out(Data_out),
45     .Addr_out(Addr_out)
46 );
47
48 // Initialize all inputs
49 initial begin
50     rst = 0;
51     #10 rst = 1;
52     #10 rst = 0;
53     // lui x1 1573
54     inst_field <= 32'h006250B7;
55     Data_in <= 32'h00000000;
56     ImmSel1 <= `IMM_SEL_U;
57     MemtoReg <= `MEM2REG_ALU;
58     ALU_Control <= `ALU_OP_R2;
59     ALUSrc_B <= 1'b1;
60     Jump <= 1'b0;
61     Branch <= 1'b0;
62     RegWrite <= 1'b1;
63     sign <= 1'b1;
64     byte_n <= `WORD;
65     jump_choose <= `JUMP_PC_IMM;
66     #15;
67     // auipc x2, 0xA38
68     inst_field <= 32'h00A38117;
69     Data_in <= 32'h00000000;
70     ImmSel1 <= `IMM_SEL_U;
71     MemtoReg <= `MEM2REG_IMM_PC;
72     ALU_Control <= `ALU_OP_ADD;
73     ALUSrc_B <= 1'b1;
74     Jump <= 1'b0;
75     Branch <= 1'b0;
76     RegWrite <= 1'b1;
77     sign <= 1'b1;
78     byte_n <= `WORD;
79     jump_choose <= `JUMP_PC_IMM;
80     #10;
81

```

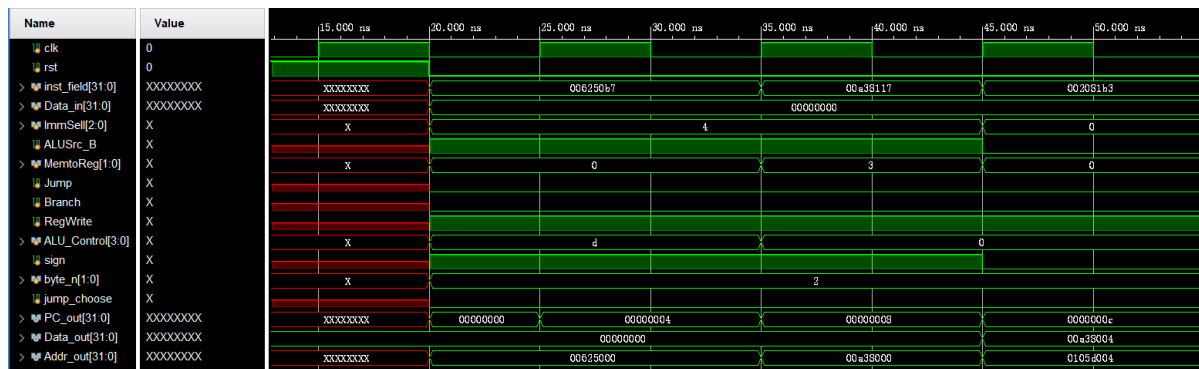


```

82 // add x3, x1, x2
83 inst_field <= 32'h002081B3;
84 Data_in <= 32'h00000000;
85 ImmSel1 <= 3'b0;
86 MemtoReg <= `MEM2REG_ALU;
87 ALUSrc_B <= 1'b0;
88 Jump <= 1'b0;
89 Branch <= 1'b0;
90 RegWrite <= 1'b1;
91 sign <= 1'b0;
92 byte_n <= `WORD;
93 jump_choose <= `JUMP_PC_IMM;
94 ALU_Control <= `ALU_OP_ADD;
95 #10;
96
97 $finish;
98
99 end
100 // Clock generation
101 always #5 clk = ~clk; // Generate a clock with a period of 10ns
102
103 endmodule

```

以下为仿真图像:



该输入为 SCPU 仿真的前三条代码,仿真图像符合前表

controller 模块仿真

以下为仿真代码

```

1 `timescale 1ns / 1ps
2 module Controller_tb ();
3     wire [4:0] OPcode;
4     reg MIO_ready;
5     wire [7:0] Fun7;
6     wire [2:0] Fun3;
7     reg [31:0] inst_field;
8     wire CPU_MIO;
9     wire [3:0] wea;
10    wire [2:0] ImmSel1;
11    wire [1:0] MemtoReg;
12    wire [3:0] ALU_Control;
13    wire ALUSrc_B;
14    wire Jump;
15    wire Branch;

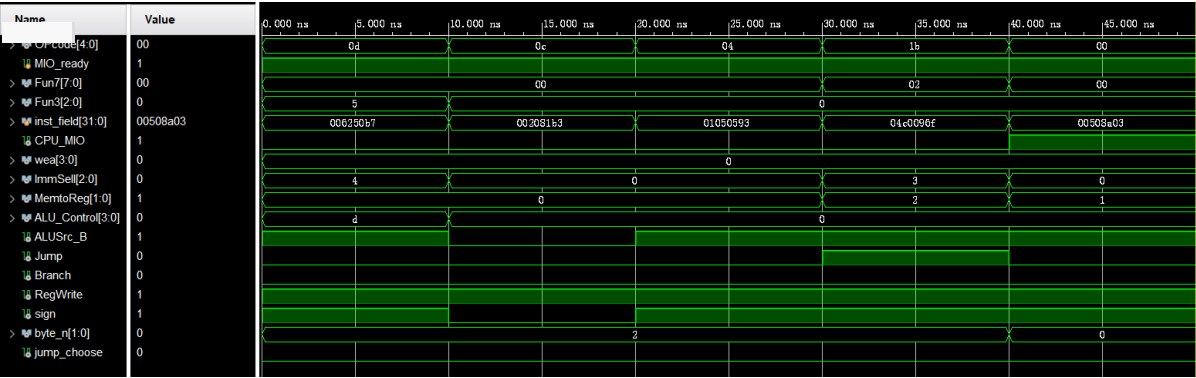
```

```

16  wire RegWrite;
17  wire sign;
18  wire [1:0] byte_n;
19  wire jump_choose;
20  Controller uut (
21      .OPcode(OPcode),
22      .MIO_ready(MIO_ready),
23      .Fun7(Fun7),
24      .Fun3(Fun3),
25      .CPU_MIO(CPU_MIO),
26      .wea(wea),
27      .ImmSel1(ImmSel1),
28      .MemtoReg(MemtoReg),
29      .ALU_Control(ALU_Control),
30      .ALUSrc_B(ALUSrc_B),
31      .Jump(Jump),
32      .Branch(Branch),
33      .RegWrite(RegWrite),
34      .sign(sign),
35      .byte_n(byte_n),
36      .jump_choose(jump_choose)
37  );
38  assign OPcode = inst_field[6:2];
39  assign Fun7   = inst_field[31:25];
40  assign Fun3   = inst_field[14:12];
41  initial begin
42      // lui x1 1573
43      MIO_ready = 1'b1;
44      inst_field = 32'h006250B7;
45      #10;
46      // add x3, x1, x2
47      inst_field = 32'h002081B3;
48      #10;
49      // addi x11, x10, 0x10
50      inst_field = 32'h01050593;
51      #10;
52      // jal x18, 76
53      inst_field = 32'h04C0096F;
54      #10;
55      // lb x20, 5(x1)
56      inst_field = 32'h00508A03;
57      #10;
58      $finish;
59  end
60  endmodule

```

以下为仿真图像：



仿真第一条指令为 `lui` ,可以看到输出 `wea` 为0不写入, `ImmSel1` 为4是U型指令, `RegWrite` 为1, `MemtoReg` 为0将 `ALU` 结果写入寄存器, `ALUSrc_B` 为1选取立即数作为第二个操作数, `ALU_Control1` 为d表示 `ALU` 的操作为保留第二操作数输出, `Jump` 和 `Branch` 均为0表示不会跳转,剩余信号不重要;仿真第二条指令为 `add` ,可以看到输出 `wea` 为0不写入, `RegWrite` 为1, `MemtoReg` 为0将 `ALU` 结果写入寄存器, `ALUSrc_B` 为0选取寄存器作为第二个操作数, `ALU_Control1` 为0表示 `ALU` 的操作为将第一第二操作数相加输出, `Jump` 和 `Branch` 均为0表示不会跳转,剩余信号不重要;仿真第三条指令为 `add` ,可以看到输出 `wea` 为0不写入, `ImmSel1` 为1是I型指令, `RegWrite` 为1, `MemtoReg` 为0将 `ALU` 结果写入寄存器, `ALUSrc_B` 为1选取立即数作为第二个操作数, `ALU_Control1` 为0表示 `ALU` 的操作为将第一第二操作数相加输出, `Jump` 和 `Branch` 均为0表示不会跳转,剩余信号不重要;其余指令照前分析均符合预期.

实验结果与分析

可以看到x31为666说明通过验收代码.