浙江大学

本科实验报告

课程名称：计算机组成与设计

姓　　名：张晋恺

学　　院：竺可桢学院

系：所在系

专　　业：计算机科学与技术

学　　号：3230102400

指导教师：刘海风

2024 年 11 月 23 日

浙江大学实验报告

课程名称： 计算机组成与设计 实验类型： 综合

实验项目名称: Lab4-3

学生姓名： 张晋恺 专业： 计算机科学与技术 学号： 3230102400

同组学生姓名： 指导老师： 刘海风

实验地点： 东 4-512 实验日期： 2024 年 11 月 7 日

一、操作方法与实验步骤

本次实验需要设计一个支持 RISC-V 指令的单周期 SCPU，实现的指令集如下：

- R-type:add, sub, and, or, xor, sll, srl, sra, slt, sltu
- I-type: addi, andi, ori, xori, slli, srli, srai, slti, sltiu, lb, lh,lbu,lhu,lw, jalr
- S-Type: sb, sh, sw
- B-Type: beq, bne, blt, bge, bltu, bgeu
- J-Type: jal
- U-Type: lui, auipc

在开始之前，我梳理了一下 SCPU 的基本模块结构，如下图所示：

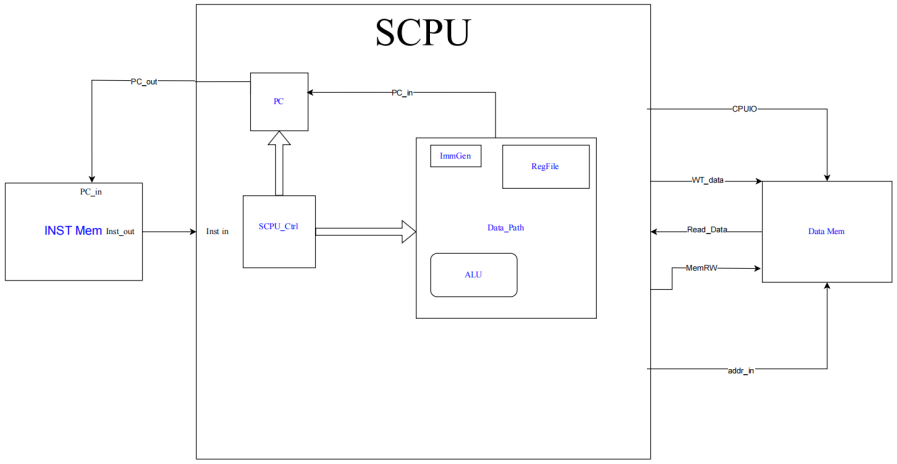


图 1: SCPU 基本模块结构

其中的 ALU 和 RegFile 模块我们已经在 lab1 中完成, 可以直接使用
总的来说。我设计的 SCPU 主要包括：

1. ScpuCtrl模块, 用于控制信号的生成
    - ALUCtrl生成 ALU 的控制信号
    - MainCtrl生成主控制信号
2. Datapath 模块, 用于数据通路的设计
    - ALU模块, 用于执行运算
    - RegFile模块, 用于寄存器的读写
    - ImmGen模块, 用于生成立即数
    - PC模块, 用于 PC 的更新
    - 以及其他逻辑组合电路
3. SCPU模块, 用于整合所有模块, 构成一个完整的 CPU

接下来我将按顺序介绍各个模块的设计思路和实现方法

各个模块的设计

ScpuCtrl 模块

我首先从最基本的 ALU 指令开始，依次扩展全部指令，在这个过程中依次为每个指令设计控制信号，最终得到了如下的控制信号表：

Selector output	ImmSel	ALUSrc_B	MemtoReg	Jump	Branch	RegWrite	MemRW	ALU_Control	CPU_MIO	signal	width
ALU_op	x	0	ALU	0	0	1	0	*	0	x	x
ALU_imm	I-Type	1	ALU	0	0	1	0	*	0	x	x
Load	I-Type	1	MEM	0	0	1	0	add	1	*	*
JALR	I-Type	1	PC+4	2'b11	0	1	0	add	0	x	x
Store	S-Type	1	x	0	0	0	1	add	1	*	*
Branch	B-Type	x	x	0	*	0	0	*	0	x	x
JAL	J-Type	x	PC+4	2'b10	0	1	0	null	0	x	x
LUI	U-Type	x	LUI	0	0	1	0	null	0	x	x
AUIPC	U-Type	x	AUIPC	0	0	1	0	null	0	x	x

表 1: Control Signals

x 代表不考虑，可以置 0，或其他值；* 代表由 fun3 和 fun7 决定；

具体代码如下

MainCtrl.v

```
`timescale 1ns / 1ps

`include "SCPU_header.vh"

module MainCtrl(
    input [4:0]       OPcode,
    input [2:0]       Fun3,
    input             Fun7,
    output reg [2:0]  ImmSel,
    output reg        ALUSrc_B,
    output reg [2:0]  MemtoReg,
    output reg [1:0]  Jump,//jal-10 jalr-11
    output reg [4:0]  Branch,//beq-0001 bne-0010 blt,bltu-0100
    ↪  bge,bgeu-1000
    output reg        signal,//used for load/store instructions, 1 for
    ↪  unsigned, 0 for signed
    output reg [1:0]  width,
```

```
    output reg         RegWrite,
    output reg         MemRW,//0 for read, 1 for write
    output reg [1:0]   ALU_op,
    output reg         CPU_MIO//1 enable, 0 disable
    );



always @(*) begin
    case(OPcode)
    `OPCODE_ALU: begin // add, sub, and, or, xor, sll, srl, sra, slt, sltu
        ImmSel = 3'b000;
        ALUSrc_B = 0;
        MemtoReg = `MEM2REG_ALU;
        Jump = 2'b00;
        Branch = 4'b0000;
        RegWrite = 1;
        MemRW = 0;
        ALU_op = 2'b10;
        CPU_MIO = 0;
        signal = 0;
        width = 2'b00;
    end

    `OPCODE_ALU_IMM: begin // addi, andi, ori, xori, slti, sltiu, slli, srli, srai
        ImmSel = `IMM_SEL_I;
        ALUSrc_B = 1;
        MemtoReg = 3'b000;
        Jump = 2'b00;
        Branch = 4'b0000;
        RegWrite = 1;
        MemRW = 0;
        ALU_op = 2'b11;
        CPU_MIO = 0;
        signal = 0;
        width = 2'b00;
    end

    `OPCODE_LOAD: begin // lb, lh, lw, lbu, lhu
        ImmSel = `IMM_SEL_I;
        ALUSrc_B = 1;
        MemtoReg = `MEM2REG_MEM;
        Jump = 2'b00;
```

```
        Branch = 4'b0000;
        RegWrite = 1;
        MemRW = 0;
        ALU_op = 2'b00;
        CPU_MIO = 1;
        signal = Fun3[2];
        width = Fun3[1:0];
    end

    `OPCODE_JALR: begin // jalr
        ImmSel = `IMM_SEL_I;
        ALUSrc_B = 1;
        MemtoReg = `MEM2REG_PC_PLUS;
        Jump = 2'b11;
        Branch = 4'b0000;
        RegWrite = 1;
        MemRW = 0;
        ALU_op = 2'b00;
        signal = 0;
        width = 2'b00;
        CPU_MIO = 0;
    end

    `OPCODE_STORE: begin // sb, sh, sw  imm(rs1) <- rs2
        ImmSel = `IMM_SEL_S;
        ALUSrc_B = 1;
        MemtoReg = 3'b000;
        Jump = 2'b00;
        Branch = 4'b0000;
        RegWrite = 0;
        MemRW = 1;
        ALU_op = 2'b00;
        CPU_MIO = 1;
        signal = 0;
        width = Fun3[1:0];
    end

    `OPCODE_BRANCH: begin // beq, bne, blt, bge, bltu, bgeu
        ImmSel = `IMM_SEL_B;
        ALUSrc_B = 0;
        MemtoReg = 3'b000;
        Jump = 2'b00;
        case(Fun3)
        `FUNC_EQ: Branch = 4'b0001; // beq
        `FUNC_NE: Branch = 4'b0010; // bne
```

```
104         `FUNC_LT: Branch = 4'b0100; // blt
105         `FUNC_GE: Branch = 4'b1000; // bge
106         `FUNC_LTU: Branch = 4'b0100; // bltu
107         `FUNC_GEU: Branch = 4'b1000; // bgeu
108         default: Branch = 4'b0000;
109         endcase
110         RegWrite = 0;
111         MemRW = 0;
112         ALU_op = 2'b01;
113         CPU_MIO = 0;
114         signal = 0;
115         width = 2'b00;
116     end
117 
118     `OPCODE_JAL : begin // jal
119         ImmSel = `IMM_SEL_J;
120         ALUSrc_B = 0;
121         MemtoReg = `MEM2REG_PC_PLUS;
122         Jump = 2'b10;
123         Branch = 4'b0000;
124         RegWrite = 1;
125         MemRW = 0;
126         ALU_op = 2'b00;
127         CPU_MIO = 0;
128         signal = 0;
129         width = 2'b00;
130     end
131 
132     `OPCODE_LUI: begin // lui
133         ImmSel = `IMM_SEL_U;
134         ALUSrc_B = 0;
135         MemtoReg = `MEM2REG_LUI;
136         Jump = 2'b00;
137         Branch = 4'b0000;
138         RegWrite = 1;
139         MemRW = 0;
140         ALU_op = 2'b00;
141         CPU_MIO = 0;
142         signal = 0;
143         width = 2'b00;
144     end
145 
146     `OPCODE_AUIPC: begin // auipc
147         ImmSel = `IMM_SEL_U;
148         ALUSrc_B = 0;
```

```
149         MemtoReg = `MEM2REG_AUIPC;
150         Jump = 2'b00;
151         Branch = 4'b0000;
152         RegWrite = 1;
153         MemRW = 0;
154         ALU_op = 2'b00;
155         CPU_MIO = 0;
156         signal = 0;
157         width = 2'b00;
158     end
159 
160     default: begin
161         ImmSel = 3'b000;
162         ALUSrc_B = 0;
163         MemtoReg = 3'b000;
164         Jump = 2'b00;
165         Branch = 4'b0000;
166         RegWrite = 0;
167         MemRW = 0;
168         ALU_op = 2'b00;
169         CPU_MIO = 0;
170         signal = 0;
171         width = 2'b00;
172     end
173     endcase
174 end
175 
176 endmodule
```

MainCtrl 模块主要是根据指令的 opcode 和 fun3,fun7，来生成控制信号，其中的 fun3，7 和 opcode 是从指令中提取出来的，具体的提取方法在后面的 Datapath 模块中会介绍

对于每一种指令，我根据 OPcode 所对应的不同的操作，根据上面所总结出来的表格，生成对应的控制信号，接口说明如下：

1. input [4:0] opcode: 指令的 opcode
2. input [2:0] fun3: 指令的 fun3
3. input fun7: 指令的 fun7
4. output reg [2:0] ImmSel: 立即数的选择信号, 用于 ImmGen 模块, 有 5 种选择, 分别是 I-Type, S-Type, B-Type, U-Type, J-Type

5. output reg ALUSrc_B: ALU 的第二个操作数的选择信号, 有两种选择, 分别是 0 和 1，对应选择寄存器的值和立即数
6. output reg [2:0] MemtoReg: 写回寄存器的数据来源选择信号, 有 5 种选择, 分别是 ALU, MEM, PC+4,LUI, AUIPC
7. output reg [1:0] Jump: 跳转指令的类型选择信号, 有 3 种选择, 分别是 JALR(11), JAL(10), 其他 (00)
8. output reg [4:0] Branch: 分支指令的类型选择信号, 有 4 种选择, 分别是 beq, bne, blt, bge, bltu, bgeu
9. output reg signal: 用于 load/store 指令的信号,1 表示无符号,0 表示有符号
10. output reg [1:0] width: 用于 load 指令的信号, 表示读取的数据的宽度, 有三种选择, 分别是 1 和 2 和 4
11. output reg RegWrite: 是否写回寄存器的信号,1 表示写回,0 表示不写回
12. output reg MemRW: 内存读写的信号,1 表示写,0 表示读
13. output reg [1:0] ALU_op: ALU 的操作类型选择信号, 有 4 种选择, 分别是 add, 正常 ALU，立即数 ALU, 比较 ALU；
14. output reg CPU_MIO: 是否启用内存 IO 的信号,1 表示启用,0 表示不启用

ALUCtrl.v

```
`timescale 1ns / 1ps
`include "SCPU_header.vh"

module ALU_ctrl(
    input [2:0]       Fun3,
    input             Fun7,
    input [1:0]       ALU_op,
    output reg [3:0]  ALU_Control
    );

always @(*) begin
    case(ALU_op)
    2'b00: ALU_Control = `ALU_OP_ADD; // add for jal jalr SB-type
    2'b01: begin
        case(Fun3) //compare operations
        `FUNC_EQ: ALU_Control = `ALU_OP_XOR; // xor=0
        ↪  zero=1,state=(b[0]&zero)
```

```
        `FUNC_NE: ALU_Control = `ALU_OP_XOR; // xor!=0, zero=0
        ↪  ,state=(b[1]&(~zero))
        `FUNC_LT: ALU_Control = `ALU_OP_SLT; // slt state=(b[2]&res[1])
        `FUNC_GE: ALU_Control = `ALU_OP_SLT; // ge state=(b[3]&(~res[1]))
        `FUNC_LTU: ALU_Control = `ALU_OP_SLTU; // sltus
        ↪  state=(b[2]&res[1])
        `FUNC_GEU: ALU_Control = `ALU_OP_SLTU; // sltu
        ↪  state=(b[3]&(~res[1]))
        default: ALU_Control = `ALU_OP_EMPTY;
        endcase
    end
    // R-type normal ALU
    2'b10: begin
        case(Fun3)
        `FUNC_ADD: ALU_Control = Fun7 ? `ALU_OP_SUB : `ALU_OP_ADD; //
        ↪  sub/add
        `FUNC_SL: ALU_Control = `ALU_OP_SLL; // sll
        `FUNC_SLT: ALU_Control = `ALU_OP_SLT; // slt
        `FUNC_SLTU: ALU_Control = `ALU_OP_SLTU; // sltu
        `FUNC_XOR: ALU_Control = `ALU_OP_XOR; // xor
        `FUNC_SR: ALU_Control = Fun7 ? `ALU_OP_SRA : `ALU_OP_SRL; //
        ↪  srar/srl
        `FUNC_OR: ALU_Control = `ALU_OP_OR; // or
        `FUNC_AND: ALU_Control = `ALU_OP_AND; // and
        default: ALU_Control = `ALU_OP_EMPTY;
        endcase
    end
    // I-type ALU op Immidiate
    2'b11: begin
        case(Fun3)
        `FUNC_ADD: ALU_Control = `ALU_OP_ADD; // add
        `FUNC_SL: ALU_Control = `ALU_OP_SLL; // sll
        `FUNC_SLT: ALU_Control = `ALU_OP_SLT; // slt
        `FUNC_SLTU: ALU_Control = `ALU_OP_SLTU; // sltu
        `FUNC_XOR: ALU_Control = `ALU_OP_XOR; // xor
        `FUNC_SR: ALU_Control = Fun7 ? `ALU_OP_SRA : `ALU_OP_SRL; //
        ↪  sra/srl
        `FUNC_OR: ALU_Control = `ALU_OP_OR; // or
        `FUNC_AND: ALU_Control =  `ALU_OP_AND; // and
        default: ALU_Control = `ALU_OP_EMPTY;
    endcase
    end
    default: ALU_Control = `ALU_OP_EMPTY;
    endcase
end
```

```

endmodule
```

ALUCtrl 模块主要是根据 fun3 和 fun7, 以及不同的指令类型，生成 ALU 的操作类型选择信号；

对于 ALU 型操作 (ALU 和 ALUI)，我们只需要根据 fun3 来生成对应的操作类型，fun7 来判断是否是有符号操作；

对于地址计算类型，我们的 ALU 操作类型一直都是 ADD；

对于 Branch 类型，我们需要根据 fun3 来判断是哪一种比较操作；

ScpuCtrl.v

```
`timescale 1ns / 1ps
module ScpuCtrl(
    input [4:0]       OPcode,
    input [2:0]       Fun3,
    input             Fun7,
    input             MIO_ready,
    output reg [2:0]  ImmSel,
    output reg        ALUSrc_B,
    output reg [2:0]  MemtoReg,
    output reg [1:0]  Jump,//11 for jalr 10 for jal
    output reg [3:0] Branch,
    output reg        RegWrite,
    output reg        MemRW,
    output reg [3:0]  ALU_Control,
    output reg        CPU_MIO,
    output reg        signal,
    output reg [1:0]  width
);

wire [1:0] ALU_op_tmp;
wire [2:0] ImmSel_tmp;
wire ALUSrc_B_tmp;
wire [2:0] MemtoReg_tmp;
wire [1:0] Jump_tmp;
wire [3:0] Branch_tmp;
wire RegWrite_tmp;
wire MemRW_tmp;
wire [3:0] ALU_Control_tmp;
wire CPU_MIO_tmp;
wire signal_tmp;
wire [1:0] width_tmp;

```

```
33  MainCtrl MainCtrl_inst(
34      .OPcode(OPcode),
35      .Fun3(Fun3),
36      .Fun7(Fun7),
37      .ImmSel(ImmSel_tmp),
38      .ALUSrc_B(ALUSrc_B_tmp),
39      .MemtoReg(MemtoReg_tmp),
40      .Jump(Jump_tmp),
41      .Branch(Branch_tmp),
42      .RegWrite(RegWrite_tmp),
43      .MemRW(MemRW_tmp),
44      .ALU_op(ALU_op_tmp),
45      .CPU_MIO(CPU_MIO_tmp),
46      .signal(signal_tmp),
47      .width(width_tmp)
48  );
49
50
51
52  ALU_ctrl ALU_ctrl_inst(
53      .Fun3(Fun3),
54      .Fun7(Fun7),
55      .ALU_op(ALU_op_tmp),
56      .ALU_Control(ALU_Control_tmp)
57  );
58
59
60  always @(*) begin
61      ImmSel = ImmSel_tmp;
62      ALUSrc_B = ALUSrc_B_tmp;
63      MemtoReg = MemtoReg_tmp;
64      Jump = Jump_tmp;
65      Branch = Branch_tmp;
66      RegWrite = RegWrite_tmp;
67      MemRW = MemRW_tmp;
68      ALU_Control = ALU_Control_tmp;
69      CPU_MIO = CPU_MIO_tmp;
70      signal = signal_tmp;
71      width = width_tmp;
72  end
73
74  endmodule
75
76
```

ScpuCtrl 模块主要是将 MainCtrl 和 ALUCtrl 模块整合在一起，根据不同的指令类型，生成对应的控制信号输出出去；

至此，ScpuCtrl 模块的设计完成，接下来我们将介绍 Datapath 模块的设计

Datapath 模块

Datapath 模块主要是将各个模块整合在一起，构成一个完整的数据通路，我自己设计的数据通路如下图所示：

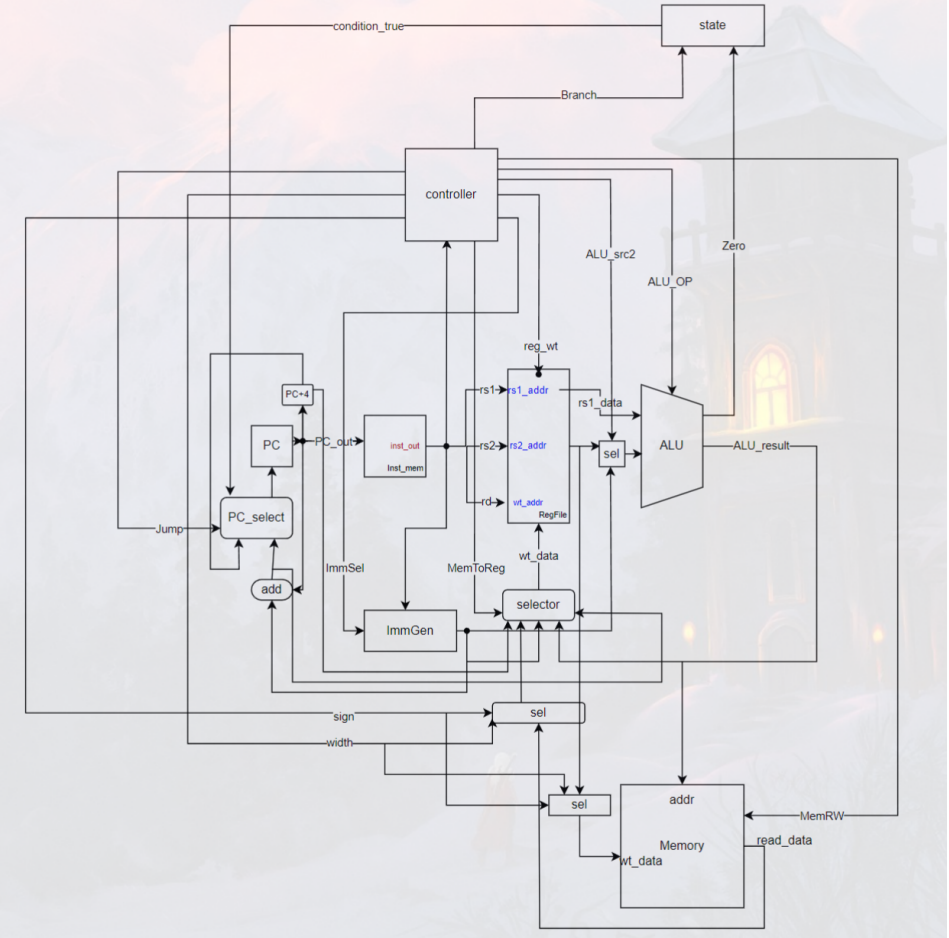


图 2: SCPU 数据通路

Verilog 代码如下：

ALU.v 和 **RegFile.v** 的代码在 lab1 中已经给出，这里不再赘述

ImmGen.v

```
`timescale 1ns / 1ps
`include "SCPU_header.vh"
module Immgen(
    input [2:0]   ImmSel,
    input [31:0]  inst_field,
    output reg [31:0] Imm_out
    );

always@(*) begin
    case(ImmSel)
         // I-type
        `IMM_SEL_I: Imm_out = {{20{inst_field[31]}}, inst_field[31:20]};
        // S-type
        `IMM_SEL_S: Imm_out = {{20{inst_field[31]}}, inst_field[31:25],
        ↪  inst_field[11:7]};
        // B-type
        `IMM_SEL_B: Imm_out = {{19{inst_field[31]}}, inst_field[31],
        ↪  inst_field[7], inst_field[30:25], inst_field[11:8], 1'b0};
        // J-type
        `IMM_SEL_J: Imm_out = {{11{inst_field[31]}}, inst_field[31],
        ↪  inst_field[19:12], inst_field[20], inst_field[30:21], 1'b0};
        // U-type
        `IMM_SEL_U: Imm_out = {inst_field[31:12], 12'b0};
        default: Imm_out = 32'b0;
    endcase
end

endmodule
```

这是立即数生成模块，根据输入的指令和 ImmSel，生成对应的立即数并输出

PC.v

```
`timescale 1ns / 1ps
module PC(
    input clk,
    input rst,
    input [31:0] PC_in,
    input ctrl,
    output reg [31:0] PC_out
    );
```

```
9
10      always @(posedge clk or posedge rst) begin
11          if(rst) begin
12              PC_out <= 32'b0;
13          end
14          else if(ctrl) begin
15              PC_out <= PC_in;
16          end
17      end
18
19  endmodule
```

这是 PC 模块，根据输入的 PC，生成下一个 PC 并输出，其实就相当于一个 D 触发器，这也是我们 SCPU 中唯一的一个时序逻辑

Datapath.v

```
1   `timescale 1ns / 1ps
2
3   `include "SCPU_header.vh"
4
5   module Datapath(
6       input clk,
7       input rst,
8       input [31:0] inst_field,
9       input [31:0] data_in,
10      input [3:0] ALU_Control,
11      input [2:0] ImmSel,
12      input [2:0] MemtoReg,
13      input [3:0] Branch,
14      input [1:0] Jump,
15      input ALUSrc_B,
16      input RegWrite,
17      input signal,
18      input [1:0] width,
19      `RegFile_Regs_Outputs
20      output reg [3:0] RAM_wt_bits,
21      output reg [31:0] Data_out,
22      output reg [31:0] ALU_out,
23      output reg [31:0] PC_out
24  );
25
26
27  reg [31:0] Wt_data;
28  wire [31:0] A, B;
```

```
wire [31:0] ALUSrc_B_Imm;
wire [31:0] ALUSrc_B_Reg;
wire [31:0] ALUSrc_A_Reg;
wire [31:0] Imm_out;
wire [31:0] ALU_res;
wire [31:0] PC_in;
wire [31:0] PC_res;
wire [4:0] rd, rs1, rs2;
wire Branch_state;
wire zero;

Immgen U1(.inst_field(inst_field), .ImmSel(ImmSel), .Imm_out(Imm_out));


assign rd = inst_field[11:7];
assign rs1 = inst_field[19:15];
assign rs2 = inst_field[24:20];


Regs U2(.clk(clk),
 .rst(rst),
 .Rs1_addr(rs1),
 .Rs2_addr(rs2),
 .Wt_addr(rd),
.Wt_data(Wt_data),
.RegWrite(RegWrite),
`RegFile_Regs_Arguments
.Rs1_data(ALUSrc_A_Reg),
.Rs2_data(ALUSrc_B_Reg)
);
//ALU input
assign ALUSrc_B_Imm = Imm_out;
assign A = ALUSrc_A_Reg;
assign B = ALUSrc_B ? ALUSrc_B_Imm : ALUSrc_B_Reg;
ALU U3(.A(A), .B(B), .ALU_operation(ALU_Control), .res(ALU_res),
 ↪  .zero(zero));
assign Branch_state = (Branch[0] & zero) | (Branch[1] & ~zero) |
 ↪  (Branch[2] & ALU_res[0]) | (Branch[3] & ~ALU_res[0]);
PC U4(.clk(clk), .rst(rst), .PC_in(PC_in), .ctrl(1'b1), .PC_out(PC_res));
assign PC_in = Jump[1] ? (Jump[0] ? ALU_res : Imm_out + PC_res) :
 ↪  (Branch_state ? (Imm_out + PC_res) : (PC_res + 4));
always @(*) begin
   ALU_out = ALU_res;
   PC_out = PC_res;
   //sb sh sw
```

```
71      case({signal, width, ALU_res[1:0]})
72          {`FUNC_BYTE, `MOD_ZERO}: begin
73              Data_out = {24'b0, ALUSrc_B_Reg[7:0]};
74              RAM_wt_bits = 4'b0001;
75          end
76          {`FUNC_BYTE, `MOD_ONE}: begin
77              Data_out = {16'b0, ALUSrc_B_Reg[7:0], 8'b0};
78              RAM_wt_bits = 4'b0010;
79          end
80          {`FUNC_BYTE, `MOD_TWO}: begin
81              Data_out = {8'b0, ALUSrc_B_Reg[7:0], 16'b0};
82              RAM_wt_bits = 4'b0100;
83          end
84          {`FUNC_BYTE, `MOD_THREE}: begin
85              Data_out = {ALUSrc_B_Reg[7:0], 24'b0};
86              RAM_wt_bits = 4'b1000;
87          end
88          {`FUNC_HALF, `MOD_ZERO}: begin
89              Data_out = {16'b0, ALUSrc_B_Reg[15:0]};
90              RAM_wt_bits = 4'b0011;
91          end
92          {`FUNC_HALF, `MOD_ONE}: begin
93              Data_out = {8'b0, ALUSrc_B_Reg[15:0], 8'b0};
94              RAM_wt_bits = 4'b0110;
95          end
96          {`FUNC_HALF, `MOD_TWO}: begin
97              Data_out = {ALUSrc_B_Reg[15:0], 16'b0};
98              RAM_wt_bits = 4'b1100;
99          end
100         {`FUNC_WORD, `MOD_ZERO}: begin
101             Data_out = ALUSrc_B_Reg;
102             RAM_wt_bits = 4'b1111;
103         end
104         default: begin
105             Data_out = 32'b0;
106             RAM_wt_bits = 4'b0000;
107         end
108     endcase
109
110     //lb lh lw lbu lhu
111     case(MemtoReg)
112      `MEM2REG_ALU: Wt_data = ALU_res;
113      `MEM2REG_MEM: begin
114          case({signal, width, ALU_res[1:0]})
115              {`FUNC_BYTE, `MOD_ZERO}: Wt_data = {{24{data_in[7]}},
                 ↪  data_in[7:0]};
```

```
            {`FUNC_BYTE, `MOD_ONE}: Wt_data = {{24{data_in[15]}},
             ↪  data_in[15:8]};
            {`FUNC_BYTE, `MOD_TWO}: Wt_data = {{24{data_in[23]}},
             ↪  data_in[23:16]};
            {`FUNC_BYTE, `MOD_THREE}: Wt_data = {{24{data_in[31]}},
             ↪  data_in[31:24]};
            {`FUNC_HALF, `MOD_ZERO}: Wt_data = {{16{data_in[15]}},
             ↪  data_in[15:0]};
            {`FUNC_HALF, `MOD_ONE}: Wt_data = {{16{data_in[23]}},
             ↪  data_in[23:8]};
            {`FUNC_HALF, `MOD_TWO}: Wt_data = {{16{data_in[31]}},
             ↪  data_in[31:16]};
            {`FUNC_WORD, `MOD_ZERO}: Wt_data = data_in;

            //unsigned lbu lhu
            {`FUNC_BYTE_UNSIGNED, `MOD_ZERO}: Wt_data = {{24{1'b0}},
             ↪  data_in[7:0]};
            {`FUNC_BYTE_UNSIGNED, `MOD_ONE}: Wt_data = {{24{1'b0}},
             ↪  data_in[15:8]};
            {`FUNC_BYTE_UNSIGNED, `MOD_TWO}: Wt_data = {{24{1'b0}},
             ↪  data_in[23:16]};
            {`FUNC_BYTE_UNSIGNED, `MOD_THREE}: Wt_data = {{24{1'b0}},
             ↪  data_in[31:24]};
            {`FUNC_HALF_UNSIGNED, `MOD_ZERO}: Wt_data = {{16{1'b0}},
             ↪  data_in[15:0]};
            {`FUNC_HALF_UNSIGNED, `MOD_ONE}: Wt_data = {{16{1'b0}},
             ↪  data_in[23:8]};
            {`FUNC_HALF_UNSIGNED, `MOD_TWO}: Wt_data = {{16{1'b0}},
             ↪  data_in[31:16]};

            default: Wt_data = 32'b0;
        endcase
    end
    `MEM2REG_PC_PLUS: Wt_data = PC_res + 4;
    `MEM2REG_LUI: Wt_data = Imm_out;
    `MEM2REG_AUIPC: Wt_data = Imm_out + PC_res;
    default: Wt_data = 32'b0;
   endcase
end
endmodule
```

Datapath 模块主要是将 ALU, RegFile, ImmGen, PC, ScpuCtrl 模块整合在一起，构成一个完整的数据通路

具体的代码解释如下

SCPU 数据通路模块解析

这段代码实现了一个简化的 SCPU（Simple CPU）数据通路模块（`Datapath`）。模块功能包括指令解码、寄存器文件操作、ALU 运算、程序计数器（PC）的更新，以及存储器访问相关逻辑。

输入信号

- `clk` 和 `rst`：时钟和复位信号，控制数据通路的时序。
- `inst_field`：当前的指令字段（32 位），提供操作码及操作数相关信息。
- `data_in`：存储器读取的数据输入。
- `ALU_Control`：控制 ALU 运算类型的信号。
- `ImmSel`：决定立即数（`Imm_out`）的生成方式。
- `MemtoReg`：决定写回寄存器的数据来源。
- `Branch`：决定分支跳转逻辑。
- `Jump`：决定跳转方式。
- `ALUSrc_B`：决定 ALU 的第二操作数来源（寄存器值或立即数）。
- `RegWrite`：控制寄存器文件的写入使能。
- `signal` 和 `width`：用于存储器读写操作的类型和宽度。

输出信号

- `RAM_wt_bits`：存储器写入时的字节使能信号。
- `Data_out`：写入存储器的数据。
- `ALU_out`：ALU 计算结果。
- `PC_out`：当前的程序计数器值。

功能模块及信号流动

1. 立即数生成（Immgen）

- 负责根据 `ImmSel` 选择立即数类型，并生成适当的立即数。
- 输入：指令字段（`inst_field`）和立即数选择信号（`ImmSel`）。
- 输出：`Imm_out`，立即数值。

2. 寄存器文件（Regs）

- `Regs` 模块实现了寄存器文件的操作。
- 输入：
  - `Rs1_addr` 和 `Rs2_addr`：从 `inst_field` 中解析出的源寄存器地址。
  - `Wt_addr`：目标寄存器地址。
  - `Wt_data`：待写入数据。
  - `RegWrite`：写使能信号。
- 输出：
  - `Rs1_data` 和 `Rs2_data`，分别传递到 ALU 的操作数 A 和 B。

3. ALU 计算（ALU）

- `ALU` 模块实现算术与逻辑运算。
- 输入：
  - `A`：第一操作数（寄存器值）。
  - `B`：第二操作数（寄存器值或立即数）。
  - `ALU_operation`：决定运算类型（如加法、减法、与、或等）。
- 输出：
  - `res`（`ALU_res`）：运算结果。
  - `zero`：零标志，用于分支控制。

4. **分支和跳转逻辑**

- 根据 Branch 和 Jump 信号控制 PC 更新：
  - 普通顺序：PC_res + 4。
  - 条件分支：Imm_out + PC_res。
  - 跳转：Imm_out 或 ALU_res。

5. **存储器访问**

- **写入存储器：**
  - 根据 signal 和 width，生成不同的字节使能信号（RAM_wt_bits）以及写入数据（Data_out）。
  - 支持 sb（字节存储）、sh（半字存储）、sw（字存储）。
- **读取存储器：**
  - 根据 MemtoReg 决定写回寄存器的数据：
    - 从存储器读取的 data_in。
    - ALU_res。
    - 特殊数据（如 PC + 4、Imm_out）。

6. PC **更新（PC）**

- PC 模块实现了程序计数器的更新逻辑，支持以下操作：
  - 顺序执行。
  - 条件分支。
  - 无条件跳转。

写回逻辑分析

根据 MemtoReg 决定最终写入寄存器的数据（Wt_data）：

1. MEM2REG_ALU：写回 ALU_res。
2. MEM2REG_MEM：写回存储器读取的数据，根据 signal 和 width 控制。
3. MEM2REG_PC_PLUS：写回 PC + 4，用于跳转指令（如 jal）。

4. `MEM2REG_LUI`：写回立即数（`lui` 指令）。

5. `MEM2REG_AUIPC`：写回 `PC + Imm_out`（`auipc` 指令）。

二、实验结果与分析

ScpuCtrl 模块的仿真

仿真代码如下

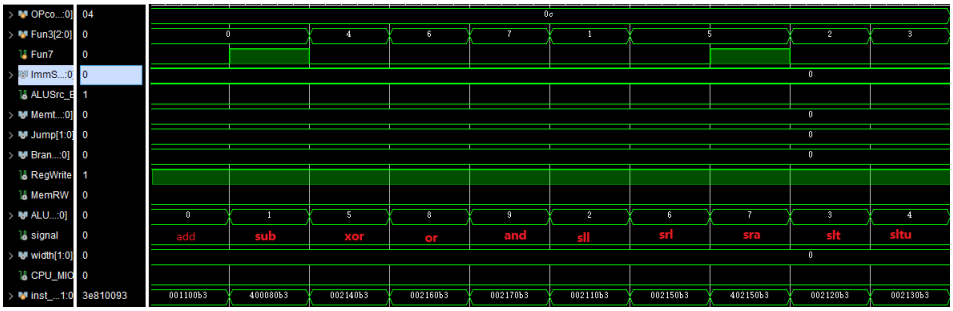
```
`timescale 1ns/1ps

// `include "SCPU_header.vh"

module SCPU_ctrl_tb();
  reg [4:0]     OPcode;
  reg [2:0]     Fun3;
  reg           Fun7;
  reg           MIO_ready;
  wire [2:0]    ImmSel;
  wire          ALUSrc_B;
  wire [2:0]    MemtoReg;
  wire [1:0]    Jump;
  wire [3:0]    Branch;
  wire          RegWrite;
  wire          MemRW;
  wire [3:0]    ALU_Control;
  wire          signal;// 0: signed, 1: unsigned
  wire [1:0]    width;// 0: byte, 1: half-word, 2: word
  wire          CPU_MIO;

  ScpuCtrl m0 (
    .OPcode(OPcode),
    .Fun3(Fun3),
    .Fun7(Fun7),
    .MIO_ready(MIO_ready),
    .ImmSel(ImmSel),
    .ALUSrc_B(ALUSrc_B),
    .MemtoReg(MemtoReg),
    .Jump(Jump),
    .Branch(Branch),
    .RegWrite(RegWrite),
    .MemRW(MemRW),
    .ALU_Control(ALU_Control),
    .CPU_MIO(CPU_MIO),
```

```
    .signal(signal),
    .width(width)
  );

  reg [31:0] inst_for_test;

`define LET_INST_BE(inst) \
  inst_for_test = inst; \
  OPcode = inst_for_test[6:2]; \
  Fun3 = inst_for_test[14:12]; \
  Fun7 = inst_for_test[30]; \
  #5

  initial begin
    #5;
    MIO_ready = 0;
    #5;
    // R-type
    `LET_INST_BE(32'h001100B3);   //add x1, x2, x1
    `LET_INST_BE(32'h400080B3);   //sub x1, x1, x0
    `LET_INST_BE(32'h002140B3);   //xor x1, x2, x2
    `LET_INST_BE(32'h002160B3);   //or x1, x2, x2
    `LET_INST_BE(32'h002170B3);   //and x1, x2, x2
    `LET_INST_BE(32'h002110B3);   //sll x1, x2, x2
    `LET_INST_BE(32'h002150B3);   //srl x1, x2, x2
    `LET_INST_BE(32'h402150B3);   //sra x1, x2, x2
    `LET_INST_BE(32'h002120B3);   //slt x1, x2, x2
    `LET_INST_BE(32'h002130B3);   //sltu x1, x2, x2

    // I-type
    `LET_INST_BE(32'h3E810093);   //addi x1, x2, 1000
    `LET_INST_BE(32'h00A14093);   //xori x1, x2, 10
    `LET_INST_BE(32'h00116093);   //ori x1, x2, 1
    `LET_INST_BE(32'h00017093);   //andi x1, x2, 0
    `LET_INST_BE(32'h01411093);   //slli x1, x2, 20
    `LET_INST_BE(32'h00515093);   //srli x1, x2, 5
    `LET_INST_BE(32'h41815093);   //srai x1, x2, 24
    `LET_INST_BE(32'hFFF12093);   //slti x1, x2, -1
    `LET_INST_BE(32'h3FF13093);   //sltiu x1, x2, 1023

    `LET_INST_BE(32'h0E910083);   //lb x1, 233(x2)
    `LET_INST_BE(32'h01411083);   //lh x1, 20(x2)
    `LET_INST_BE(32'h00812083);   //lw x1, 8(x2)
    `LET_INST_BE(32'h0E914083);   //lbu x1, 233(x2)
    `LET_INST_BE(32'h01415083);   //lhu x1, 20(x2)
```

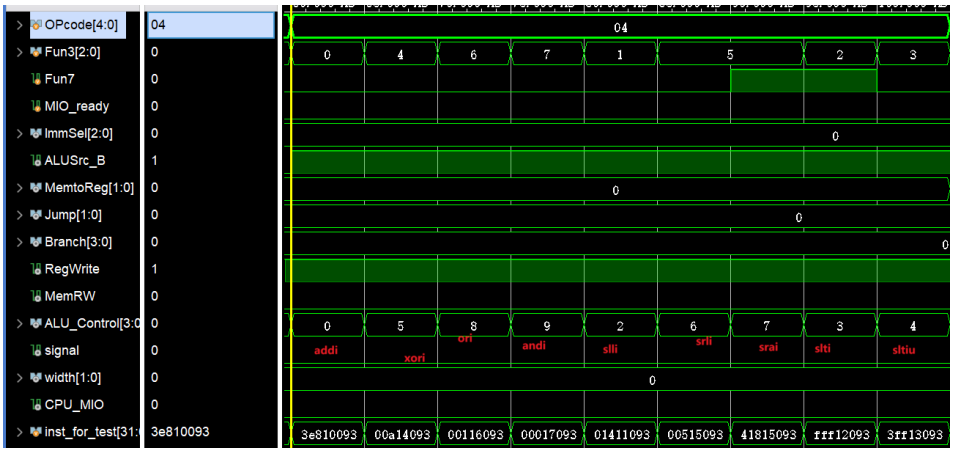
```
81
82      `LET_INST_BE(32'h004100E7);    //jalr x1, x0, 4
83
84      // S-type
85      `LET_INST_BE(32'hFE110DA3);    //sb x1, -5(x2)
86      `LET_INST_BE(32'h00211023);    //sh x2, 0(x2)
87      `LET_INST_BE(32'h00C0A823);    //sw x12, 16(x1)
88
89      // B-type
90      `LET_INST_BE(32'hFE108AE3);    //beq x1, x1, -12
91      `LET_INST_BE(32'h00211463);    //bne x2, x2, 8
92      `LET_INST_BE(32'h0031C463);    //blt x3, x3, 8
93      `LET_INST_BE(32'hFE425CE3);    //bge x4, x4, -8
94      `LET_INST_BE(32'h0031E463);    //bltu x3, x3, 8
95      `LET_INST_BE(32'hFE427CE3);    //bgeu x4, x4, -8
96
97      // J-type
98      `LET_INST_BE(32'hF9DFF06F);    //jal x0, -100
99      `LET_INST_BE(32'h3FE000EF);    //jal x1, 1023
100
101     // U-type
102     `LET_INST_BE(32'h000000B7);    //lui x1, 0
103     `LET_INST_BE(32'h000640B7);    //lui x1, 0
104     `LET_INST_BE(32'h00000097);    //auipc x1, 0
105     `LET_INST_BE(32'h00064097);    //auipc x1, 100
106
107     #50; $finish();
108   end
109 endmodule
```

对每一种指令, 我们分别分析其仿真结果

ALU 指令



(a) R-type ALU 指令

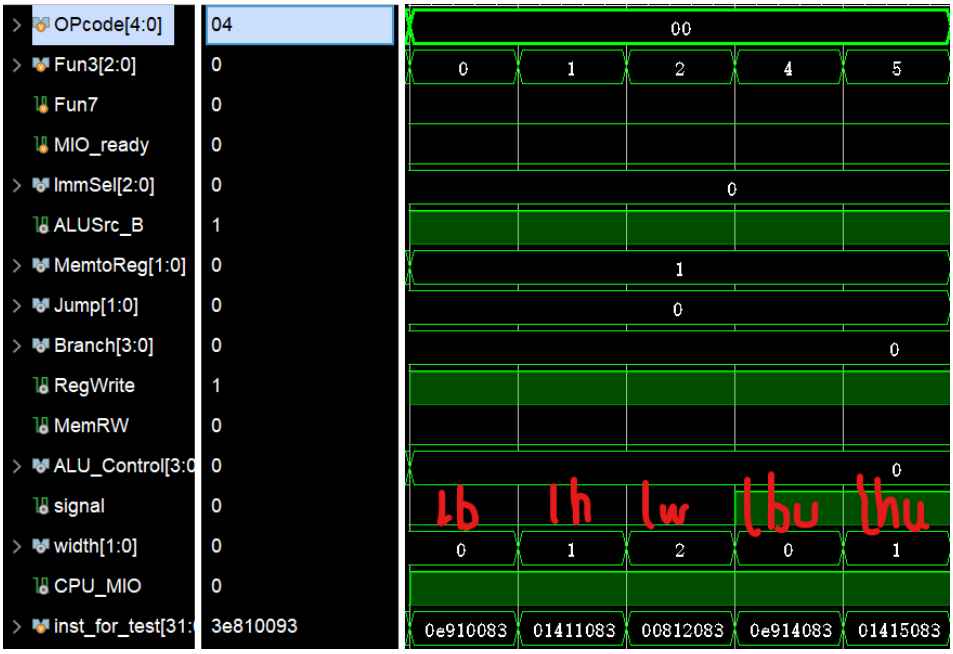


(b) I-type ALU 指令

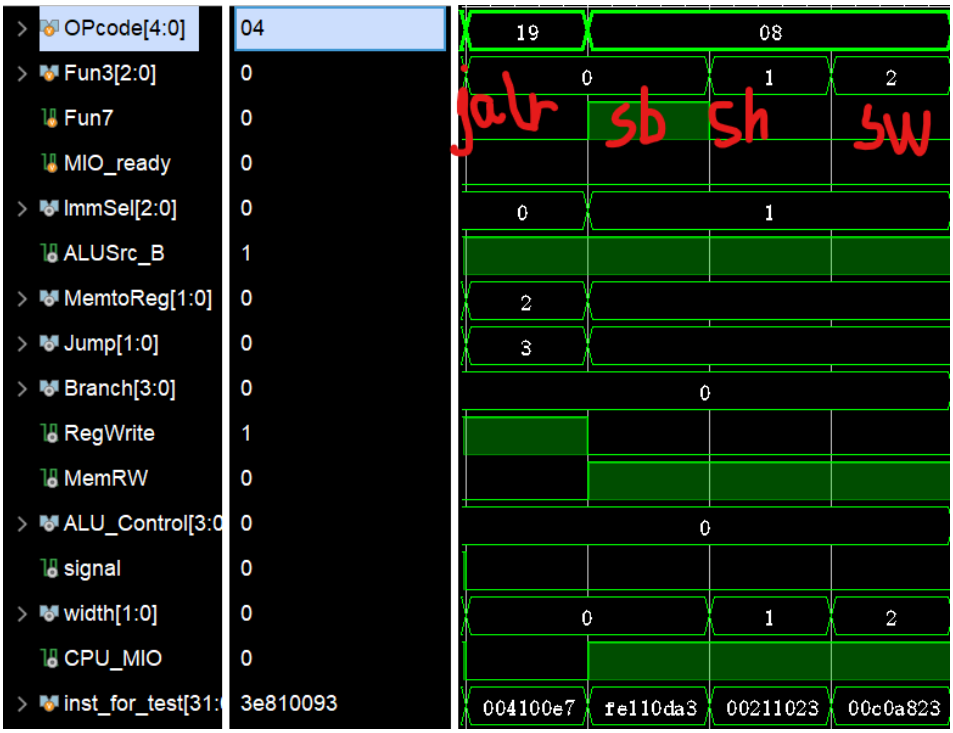
图 3: ALU 指令仿真结果

可以看到，对于不同的指令，其相关控制信号的输出与我们的表格是一致的。相应的 ALU 操作也是正确的。

访存指令和 Jalr 指令



(a) Load 指令



(b) JALR-Store 指令

图 4: 访存指令仿真结果

可以看到，对于不同的指令，其相关控制信号的输出与我们的表格是一致, 同时对于不同的字节，半字，字的访存指令，其写入数据的长度和符号控制也正常运行，符合我们的预期。

BJ 指令

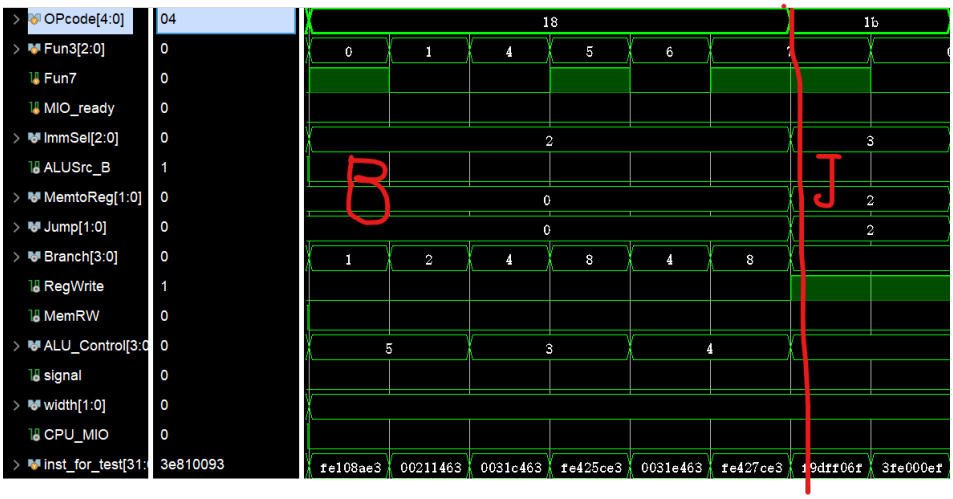


图 5: Branch 指令仿真结果

Branch 指令和 Jump 指令的控制信号也是正确的，可以看到，对于 Branch 指令，的不同操作，对应的码值也是正确的。

U-type 指令

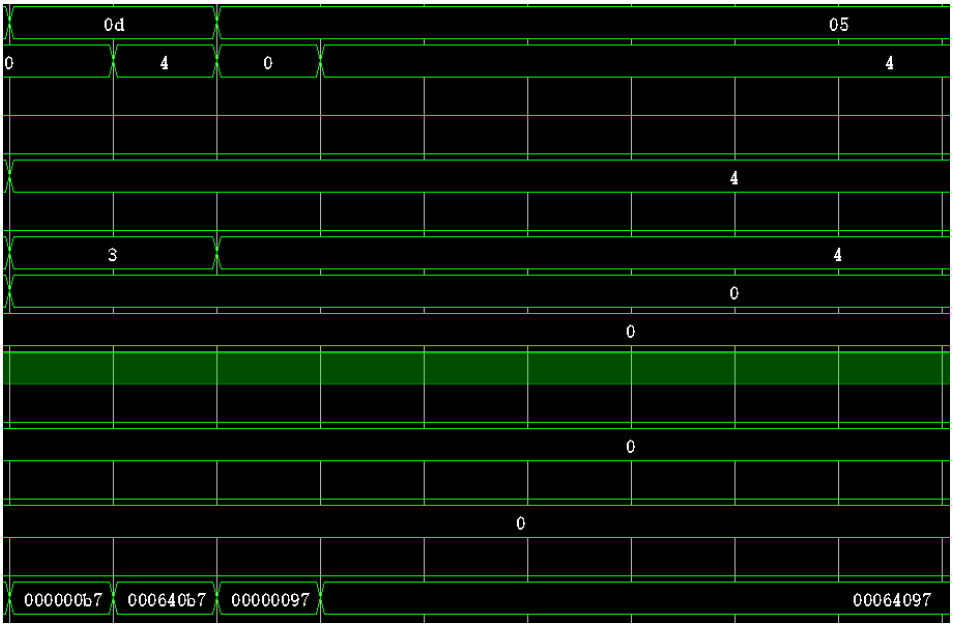


图 6: U-type 指令仿真结果

对于 U-type 指令，只有 lui 和 auipc 两种指令，其控制信号的区别在与写回寄存器的选择。

SCPU 的总体仿真

为了验证我们的 CPU 的正确性，我们对其进行了总体仿真，新建了一个 testbench 文件，代码如下

```
`timescale 1ns / 1ps
module testbench(
    input clk,
    input rst
    );

     /* SCPU 中接出 */
    wire [3:0] wt_bit;
    wire [31:0] Addr_out;
    wire [31:0] Data_out;
    wire        CPU_MIO;
    wire        MemRW;
```

```
    wire [31:0] PC_out;
    /* RAM 接出 */
    wire [31:0] douta;
    /* ROM 接出 */
    wire [31:0] spo;

MyScpu inst0(
    .clk(clk),
    .rst(rst),
    .Data_in(douta),
    .MIO_ready(CPU_MIO),
    .inst_in(spo),
    .RAM_wt_bits(wt_bit),
    .Addr_out(Addr_out),
    .Data_out(Data_out),
    .CPU_MIO(CPU_MIO),
    .MemRW(MemRW),
    .PC_out(PC_out)
);


RAM_B u1(
    .clka(~clk),
    .wea({4{MemRW}} & wt_bit),
    .addra(Addr_out[11:2]),
    .dina(Data_out),
    .douta(douta)
);


ROM_in_testbench u2(
    .addr(PC_out[11:2]),
    .spo(spo),
);

endmodule
```

其中的 ROM 模块是根据实验文档中给出的验收代码生成的，如果我们的 CPU 能够正确运行，那么其最后的 x31 寄存器的值会是 0x666；

由于仿真波形过长，我们只展示最后 x31 为 0x666 的部分

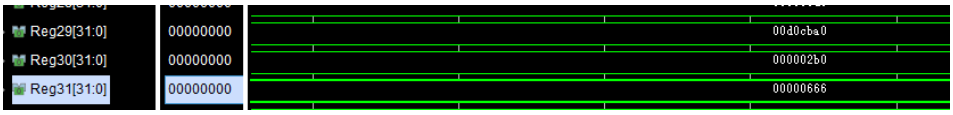


图 7: SCPU 总体仿真结果

可以看到，我们的 CPU 能够正确运行，最后的 x31 寄存器的值为 0x666；

SCPU 的综合与实现

将 lab2 中的 SCPU 替换为我们实现的 SCPU，对 CSSTE 文件进行修改，我们将所有 debug 信号从 datapath 和 ctrl 模块中输出，以便于我们的观察;

最后上板的结果如下

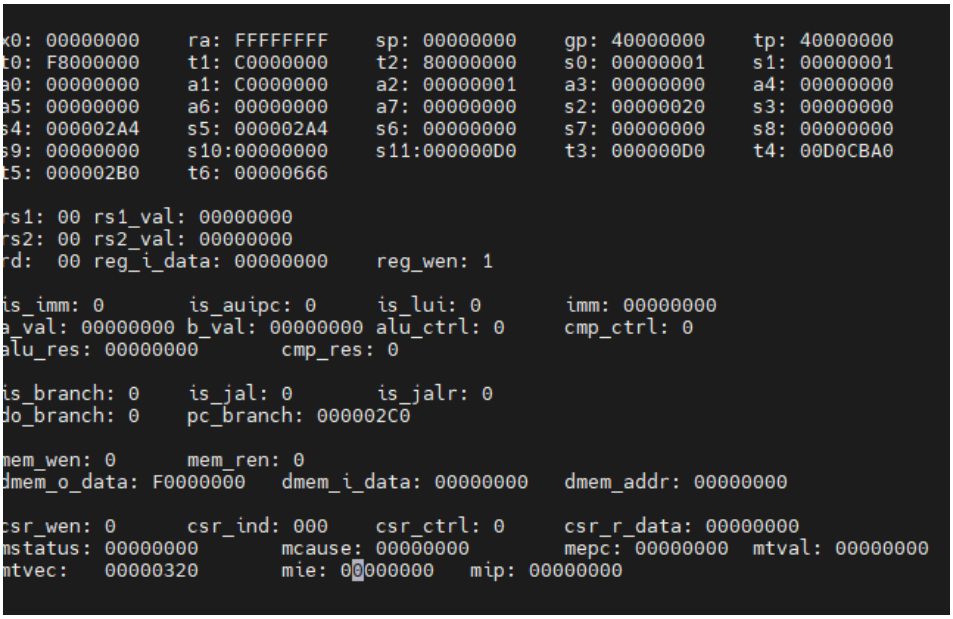


图 8: SCPU 上板结果

可以看到，我们的 CPU 能够正确运行，最后的 x31 寄存器的值为 0x666；最后进入 dummy 循环；

三、讨论与实验心得

在本次实验一开始，我就想着不按照 4-1 和 4-2 的顺序做，首先我去智云课堂上看了姜老师的课，跟着她从 ALU 指令开始，一步步给 datapath 增加新的指令，这样的过程结束后，我对整个 SCPU 的结构有了清晰的认识，也对各种指令的执行过程以及控制信号有了更深的理解。然后我再开始画 datapath，感觉十分顺利，然后再完成了控制模块，到这里最后的工作只是按照 datapath 进行简单的连线了，十分顺利就写完了代码；

但是我并没有一下子就结束，由于线路很多，我连错了一些地方，以及在定义数据类型的时候，我把一些 reg 定义成了 wire，导致了一些赋值上的小错误，这些错误让我花了很多时间去 debug，但是最后还是解决了。

也提升了我对于 verilog 的熟练度，当我最后终于在仿真波形里面看到了 0x666 的时候，我感到了很大的成就感，也让我对接下来的中断处理有了更多的信心。