浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	汪珉凯
学　　院：	计算机科学与技术学院
专　　业：	计算机
指导教师：	刘海风
报告日期：	2024 年 5 月 4 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：SCPU 的设计与实现

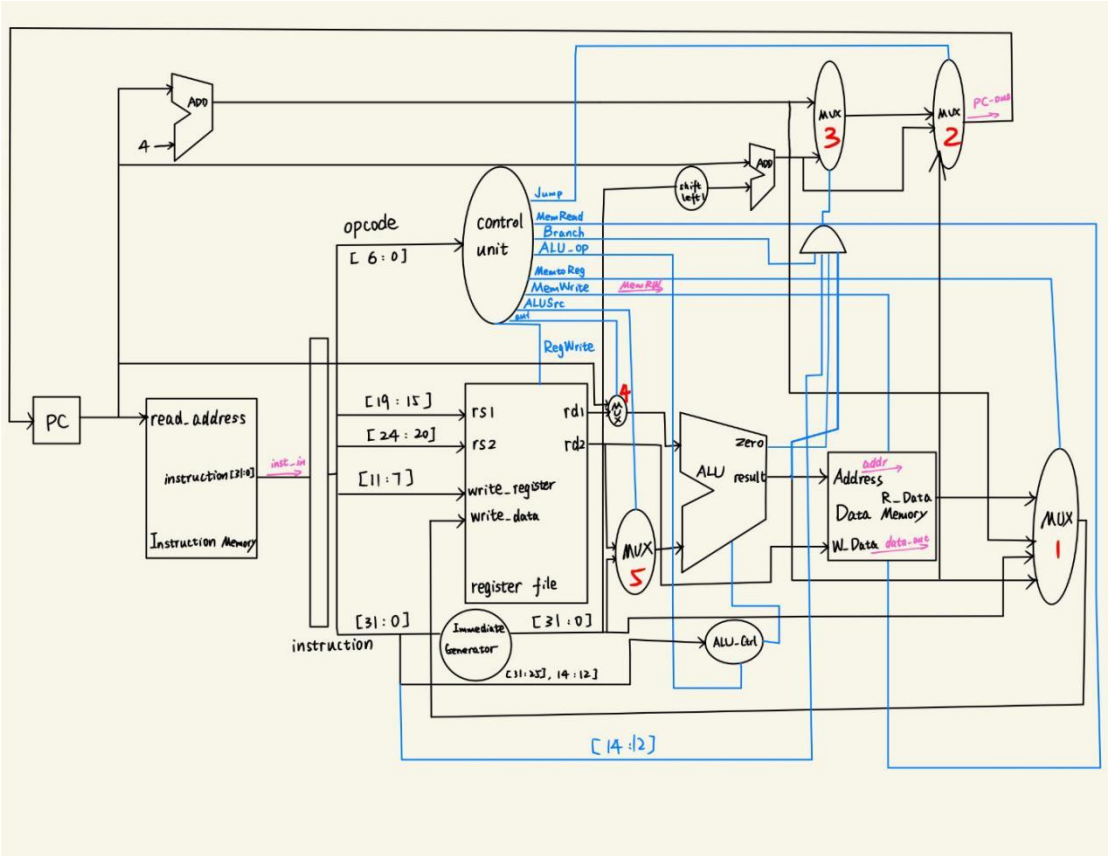
学生姓名：汪珉凯 学号：3220100975 同组学生姓名：赵桉、马扬松

实验地点：紫金港东四 509 室 实验日期：2024 年 5 月 4 日

我选择直接完成 Lab4.3

一、操作方法与实验步骤

1.设计 Datapath 通路图：



可以看到，我设计的 SCPU 主要由 ControlUnit(产生控制信号) / ALU_Ctrl(产生 ALU 的控制信号) / Imm_Gen(生成立即数) / RegFiles(寄存器堆) / ALU(计算) / PC(指令编号) 以及五个 MUX 组成。

1.2SCPU 具体实现

1.2.1.ControlUnit：

```
`timescale 1ns / 1ps
module Control_Unit(
    input [6:0] opcode,
    input [2:0] funct3,
    output ALUSrc,//0->use rs2 ; 1->use imm
    output [1:0] Jump,//00->PC+4/Branch ; 01->jal ; 10->jalr //this is the last mux before PC
    output Branch,
    output [2:0] MemRead, //001->lb ; 010->lh ; 011->lw ; 101->lbu ; 110->lhu
    output [1:0] MemtoReg,//00->ALU_result ; 01->Memory ; 10->PC+4 ; 11->imm
    output [1:0] MemWrite,//01->sb ; 10->sh ; 11->sw
    output RegWrite,
    output [1:0] ALU_op,//00->ld/sd/auipc ; 01->beq ; 10->R_type
    output aui//1->auipc ; 0->others
    );
reg ALUSrc0,RegWrite0,Branch0,aui0;
reg[1:0]  Jump0,ALU_op0,MemtoReg0,MemWrite0;
reg[2:0]  MemRead0;
assign RegWrite=RegWrite0;
assign MemRead=MemRead0;
assign MemWrite=MemWrite0;
assign Branch=Branch0;
assign aui=aui0;
assign Jump=Jump0;
assign ALU_op=ALU_op0;
assign ALUSrc=ALUSrc0;
assign MemtoReg=MemtoReg0;
always @ (*) begin
    case(opcode)
        7'b0110011://R add
            begin
                ALUSrc0<=0;
                MemtoReg0<=2'b00;
                RegWrite0<=1;
                MemRead0<=3'b0;
                MemWrite0<=2'b0;
                Branch0<=0;
                Jump0<=2'b00;
                ALU_op0<=2'b10;
                aui0<=0;
            end
        7'b0010011://I  addi
            begin
```

```
            ALUSrc0<=1;
            MemtoReg0<=2'b00;
            RegWrite0<=1;
            MemRead0<=3'b0;
            MemWrite0<=2'b0;
            Branch0<=0;
            Jump0<=2'b00;
            ALU_op0<=2'b10;
            aui0<=0;
        end
    7'b0000011://I  ld
        begin
            ALUSrc0<=1;
            MemtoReg0<=2'b01;
            RegWrite0<=1;
            // MemRead0<=1;
            MemWrite0<=2'b0;
            Branch0<=0;
            Jump0<=2'b00;
            ALU_op0<=2'b00;
            aui0<=0;
            case(funct3)
                3'b000:MemRead0<=3'b001;
                3'b001:MemRead0<=3'b010;
                3'b010:MemRead0<=3'b011;
                3'b100:MemRead0<=3'b101;
                3'b101:MemRead0<=3'b110;
            endcase
        end
    7'b0100011://S sd
        begin
            ALUSrc0<=1;
            RegWrite0<=0;
            MemRead0<=3'b0;
            Branch0<=0;
            Jump0<=2'b00;
            ALU_op0<=2'b00;
            aui0<=0;
            case(funct3)
                3'b000:MemWrite0<=2'b01;
                3'b001:MemWrite0<=2'b10;
                3'b010:MemWrite0<=2'b11;
            endcase
        end
```

```
        7'b1100011://B beq
            begin
                ALUSrc0<=0;
                RegWrite0<=0;
                MemRead0<=3'b0;
                MemWrite0<=2'b0;
                Branch0<=1;
                Jump0<=2'b00;
                ALU_op0<=2'b01;
                aui0<=0;
            end
        7'b1101111://J jal
            begin
                MemtoReg0<=2'b10;
                RegWrite0<=1;
                MemRead0<=3'b0;
                MemWrite0<=2'b0;
                Branch0<=0;
                Jump0<=2'b01;
                aui0<=0;
            end
        7'b1100111://J jalr
            begin
                ALUSrc0<=1;
                MemtoReg0<=2'b10;
                RegWrite0<=1;
                MemRead0<=3'b0;
                MemWrite0<=2'b0;
                Branch0<=0;
                Jump0<=2'b10;
                ALU_op0<=2'b00;
                aui0<=0;
            end
        7'b0110111://U lui
            begin
                MemtoReg0<=2'b11;
                RegWrite0<=1;
                MemRead0<=3'b0;
                MemWrite0<=2'b0;
                Branch0<=0;
                Jump0<=2'b00;
                aui0<=0;
            end
        7'b0010111:
```

```
            begin
                aui0<=1;
                ALUSrc0<=1;
                ALU_op0<=2'b00;
                MemWrite0<=2'b0;
                MemRead0<=3'b0;
                Branch0<=0;
                Jump0<=2'b00;
                MemtoReg0<=2'b00;
                RegWrite0<=1;
            end

    endcase
end
endmodule
```

Control_Unit 模块实现的功能是：根据指令的 7 位 opcode 和 3 位 function3，实现控制信号的产生。我利用了 case 语句，产生了以下 9 个控制信号，分别具有不同含义：

ALUSrc：用于区分 ALU 的计算数据来源，0 表示来源于寄存器，1 表示来源于立即数。

[1:0] Jump：用于区分 Jump 类指令，00 表示不是 Jump，01 表示 jal 指令，10 表示 jalr 指令。

Branch：用于区分 Branch 类指令，0 表示非 Branch，1 表示 Branch

[2:0] MemRead：用于区分不同 load 类型，000 表示非 load 类，001 表示 lb，010 表示 lh，011 表示 lw，101 表示 lbu，110 表示 lhu。

[1:0] MemtoReg：用于区分写入 RegFile 的是数据来源，00 表示来源于 ALU 的计算结果，01 表示来源于 Memory，10 表示来源与 PC+4，11 表示来源于立即数。

[1:0] MemWrite：用于区分 store 类操作：00 表示非 s 类指令，01 表示 sb 指令，10 表示 sh 指令，11 表示 sw 指令。RegWrite,

[1:0] ALU_op：用于区分 ALU 的功能，00 表示是 load 类/store 类/auipc 指令， 01 表示 Branch 类指令，10 表示 R 类指令。

Aui：用于区分 auipc 指令，0 表示非 auipc，1 表示是 auipc 指令。

1.2.2.ALU_Ctrl：

```
`timescale 1ns / 1ps
module ALU_Ctrl(
    input [6:0] opcode,
    input [1:0] ALU_op,
    input [2:0] fun3,
    input [6:0] fun7,
    output [3:0] ALU_operation
    );
```

```
reg [3:0] ALU_operation0;
assign ALU_operation=ALU_operation0;
    always @ (*) begin
        case(ALU_op)
            2'b00:ALU_operation0<=4'b0000;//如果是 ld/sd/auipc 操作，直接做 ADD
            2'b01://如果是 branch 类操作
                case(fun3)
                    3'd0,3'd1:ALU_operation0<=4'd1;//sub->beq/bne
                    3'd4,3'd5:ALU_operation0<=4'd3;//slt->blt/bge
                    3'd6,3'd7:ALU_operation0<=4'd4;//sltu->bltu/bgeu
                endcase
            2'b10:
                case(fun3)
                    3'd0:
                        if(opcode==7'b0110011 && fun7[5]==1'b1)begin
                            ALU_operation0<=4'd1;//SUB
                        end
                        else begin
                            ALU_operation0<=4'd0;
                        end
                    3'd1:ALU_operation0<=4'd2;//SLL
                    3'd2:ALU_operation0<=4'd3;//SLT
                    3'd3:ALU_operation0<=4'd4;//SLTU
                    3'd4:ALU_operation0<=4'd5;//XOR
                    3'd5:
                        case(fun7[5])
                            1'b0:ALU_operation0<=4'd6;//SRL
                            1'b1:ALU_operation0<=4'd7;//SRA
                        endcase
                    3'd6:ALU_operation0<=4'd8;//OR
                    3'd7:ALU_operation0<=4'd9;//AND
                endcase
        endcase
    end
endmodule
```

ALU_Ctrl 模块实现的功能是：产生控制 ALU 的信号 ALU_operation，我还是利用了 case 语句实现。首先根据 ALU_op 分类：

若 ALU_op 为 00，代表是 load、store、auipc 类指令，ALU 一定执行 add 操作。

若 ALU_op 为 01，代表是 branch 类指令，需要继续根据 fun3 的具体内容进一步分类：如果是 beq、bne 操作，则 ALU 进行 SUB 操作；如果是 blt、bge 操作，则 ALU 进行 slt 操作；如果是 bltu、bgeu 操作，则 ALU 进行 sltu 操作。

若 ALU_op 为 10，代表是 R 类指令，继续细分即可。

1.2.3.Imm_Gen:

```
`timescale 1ns / 1ps


module Imm_Gen(
    input [31:0] instruction,
    output [31:0] imm
    );
reg [31:0] imm0;
assign imm=imm0;
always @(*) begin
    case(instruction[6:0])
        7'b0010011://I type
            begin
                if(instruction[14:12] == 3'd1 || instruction[14:12] == 3'd5)begin
                    imm0[31:5]<={27{instruction[31]}};
                    imm0[4:0]<=instruction[24:20];
                end
                else begin
                    imm0[31:12]<={20{instruction[31]}};
                    imm0[11:0]<=instruction[31:20];
                end
            end
        7'b0000011,7'b1100111://I type
            begin
                imm0[31:12]<={20{instruction[31]}};
                imm0[11:0]<=instruction[31:20];
            end
        7'b0100011://S type
            begin
                imm0[31:12]<={20{instruction[31]}};
                imm0[11:5]<=instruction[31:25];
                imm0[4:0]<=instruction[11:7];
            end
        7'b1100011://B type
            begin
                imm0[31:13]<={19{instruction[31]}};
                imm0[12]<=instruction[31];
                imm0[10:5]<=instruction[30:25];
                imm0[4:1]<=instruction[11:8];
                imm0[11]<=instruction[7];
                imm0[0]<=1'b0;
            end
        7'b1101111://J type
            begin
```

```
                imm0[20]<=instruction[31];
                imm0[10:1]<=instruction[30:21];
                imm0[11]<=instruction[20];
                imm0[19:12]<=instruction[19:12];
                imm0[31:21]<={11{instruction[31]}};
                imm0[0]<=1'b0;
            end
        7'b0110111,7'b0010111:
            begin
                imm0[31:12]<=instruction[31:12];
                imm0[11:0]<=12'b0000_0000_0000;
            end
    endcase
end
endmodule
```

Imm_Gen 模块实现的功能是：根据输入的不同指令，生成不同的立即数。

RegFiles / ALU 这两个模块几乎可以直接调用前几个实验的结果，这里不再作额外赘述。

1.2.5SCPU 顶层模块：

```
`timescale 1ns / 1ps

module SCPU(
    input MIO_ready,
    input clk,
    input rst,
    input [31:0] Data_in,
    input [31:0] inst_in,
    output CPU_MIO,
    output [3:0]  MemRW,
    output [31:0] Addr_out,
    output [31:0] Data_out,
    output [31:0] PC_out
);
reg [3:0] MemRW0;
assign MemRW=MemRW0;
//Control_Unit
wire [1:0] Jump;
wire [2:0] MemRead;
wire [1:0] MemWrite;
wire Branch;
wire [1:0] ALU_op;
wire [1:0]MemtoReg;
wire ALUSrc;
```

```
wire aui;
wire RegWrite;
//ALU_Ctrl
wire [3:0] ALU_operation;
//RegFiles
wire [31:0] write_data;//the line to write data into the regs
wire [31:0] Rs1_data , Rs2_data;// rd1/rd2
//ALU
wire zero;//if the result of alu is 0
wire [31:0] alu1 , alu2 ,ALU_result;// the 2 source of the alu and the result
assign Addr_out=ALU_result;
wire[1:0] w;
assign w=Addr_out%4;
//Imm_Gen
wire [31:0] imm;
//PC
wire [31:0] pc;
wire [31:0] pc1;
wire [31:0] pc2;
assign pc=PC_out;
assign pc1=pc+4;
assign pc2=pc+imm;
//MUX3
wire [31:0] branch_pc;
//MUX2
wire [31:0] next_pc;
wire [31:0] din;
reg  [31:0] din0,dout0;
always @(*) begin
    case(MemRead)
        3'b001: //lb
            begin
                case(w)
                    2'b00:begin
                        din0[7:0]<=Data_in[7:0];
                        din0[31:8]<={24{Data_in[7]}};
                    end
                    2'b01:begin
                        din0[7:0]<=Data_in[15:8];
                        din0[31:8]<={24{Data_in[15]}};
                    end
                    2'b10:begin
                        din0[7:0]<=Data_in[23:16];
                        din0[31:8]<={24{Data_in[23]}};
```

```
                end
                2'b11:begin
                    din0[7:0]<=Data_in[31:24];
                    din0[31:8]<={24{Data_in[31]}};
                end
            endcase
        end
    3'b010: //lh
        begin
            case(w)
                2'b00:begin
                    din0[15:0]<=Data_in[15:0];
                    din0[31:16]<={16{Data_in[15]}};
                end
                2'b01:begin
                    din0[15:0]<=Data_in[23:8];
                    din0[31:16]<={16{Data_in[23]}};
                end
                2'b10:begin
                    din0[15:0]<=Data_in[31:16];
                    din0[31:16]<={16{Data_in[31]}};
                end
            endcase
        end
    3'b011: //lw
        din0<=Data_in;
    3'b101: begin//lbu
        case(w)
            2'b00:begin
                din0[7:0]<=Data_in[7:0];
                din0[31:8]<=0;
            end
            2'b01:begin
                din0[7:0]<=Data_in[15:8];
                din0[31:8]<=0;
            end
            2'b10:begin
                din0[7:0]<=Data_in[23:16];
                din0[31:8]<=0;
            end
            2'b11:begin
                din0[7:0]<=Data_in[31:24];
                din0[31:8]<=0;
            end
```

```
            endcase
        end
        3'b110: begin//lhu
            case(w)
                2'b00:begin
                    din0[15:0]<=Data_in[15:0];
                    din0[31:16]<=0;
                end
                2'b01:begin
                    din0[15:0]<=Data_in[23:8];
                    din0[31:16]<=0;
                end
                2'b10:begin
                    din0[15:0]<=Data_in[31:24];
                    din0[31:16]<=0;
                end
            endcase
        end
        default: din0<=0;
    endcase
end
always @(*)begin
    case(MemWrite)
        2'b01: //sb
            begin
                case(w)
                    2'b00:begin
                        dout0[7:0]<=Rs2_data[7:0];
                        dout0[31:8]<=0;
                    end
                    2'b01:begin
                        dout0[15:8]<=Rs2_data[7:0];
                        dout0[31:16]<=0;
                        dout0[7:0]<=0;
                    end
                    2'b10:begin
                        dout0[23:16]<=Rs2_data[7:0];
                        dout0[31:24]<=0;
                        dout0[15:0]<=0;
                    end
                    2'b11:begin
                        dout0[31:24]<=Rs2_data[7:0];
                        dout0[23:0]<=0;
                    end
```

```
                endcase
            end
        2'b10: //sh
            begin
              case(w)
                2'b00:begin
                    dout0[15:0]<=Rs2_data[15:0];
                    dout0[31:16]<=0;
                end
                2'b01:begin
                    dout0[23:8]<=Rs2_data[15:0];
                    dout0[31:24]<=0;
                    dout0[7:0]<=0;
                end
                2'b10:begin
                    dout0[31:16]<=Rs2_data[15:0];
                    dout0[15:0]<=0;
                end
                endcase
            end
        2'b11: //sw
            dout0<=Rs2_data;
        default: dout0<=0;
    endcase
end
always @(*)begin
    if(inst_in[6:0]==7'b0100011)begin
        case(inst_in[14:12])
            3'b000://sb
                begin
                    case(w)
                        2'b00:MemRW0<=4'b0001;
                        2'b01:MemRW0<=4'b0010;
                        2'b10:MemRW0<=4'b0100;
                        2'b11:MemRW0<=4'b1000;
                    endcase
                end
            3'b001://sh
                begin
                    case(w)
                        2'b00:MemRW0<=4'b0011;
                        2'b01:MemRW0<=4'b0110;
                        2'b10:MemRW0<=4'b1100;
                    endcase
```

```
                end
            3'b010://sw
                begin
                    MemRW0<=4'b1111;
                end
            default:MemRW0<=4'b0000;
        endcase
    end
    else MemRW0<=4'b0000;
end
assign din=din0;
assign Data_out=dout0;
Regs Regs(
    .clk(clk),
    .rst(rst),
    .Rs1_addr(inst_in[19:15]),
    .Rs2_addr(inst_in[24:20]),
    .Wt_addr(inst_in[11:7]),
    .Wt_data(write_data),
    .RegWrite(RegWrite),
    .Rs1_data(Rs1_data),
    .Rs2_data(Rs2_data)
);
ALU ALU(
    .A(alu1),
    .B(alu2),
    .ALU_operation(ALU_operation),
    .res(ALU_result),
    .zero(zero)
);
Control_Unit Control_Unit(
    .opcode(inst_in[6:0]),
    .funct3(inst_in[14:12]),
    .ALUSrc(ALUSrc),
    .Jump(Jump),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .RegWrite(RegWrite),
    .ALU_op(ALU_op),
    .aui(aui)
);
ALU_Ctrl ALU_Ctrl(
```

```
    .opcode(inst_in[6:0]),
    .ALU_op(ALU_op),
    .fun3(inst_in[14:12]),
    .fun7(inst_in[31:25]),
    .ALU_operation(ALU_operation)
);
Imm_Gen Imm_Gen(
    .instruction(inst_in),
    .imm(imm)
);
MUX1 MUX1(
    .MemtoReg(MemtoReg),
    .from_alu(ALU_result),
    .from_imm(imm),
    .from_memory(din),
    .from_pc(pc1),
    .reg_write_data(write_data)
);
MUX2 MUX2(
    .Jump(Jump),
    .from_pc_branch(branch_pc),
    .from_jal(pc2),
    .from_jalr(ALU_result),
    .next_pc(next_pc)
);
MUX3 MUX3(
    .Branch(Branch),
    .zero(zero),
    .fun3(inst_in[14:12]),
    .from_pc(pc1),
    .from_branch(pc2),
    .ALU_result(ALU_result),
    .branch_pc(branch_pc)
);
MUX4 MUX4(
    .aui(aui),
    .rd1(Rs1_data),
    .pc(pc),
    .alu1(alu1)
);
MUX5 MUX5(
    .ALUSrc(ALUSrc),
    .rd2(Rs2_data),
    .imm(imm),
```

```
    .alu2(alu2)
);
PC PC(
    .clk(clk),
    .rst(rst),
    .D(next_pc),
    .Q(PC_out)
);
endmodule
```

可以看到，在顶层模块中，我除了实现了各模块的调用连线，还有 3 段额外的代码，分别实现了：根据 Load 和 Store 指令具体读写长度的不同，对 Data_in、Data_out 分别进行了处理，然后也对 MemRW 实现了从一位到四位的扩展。

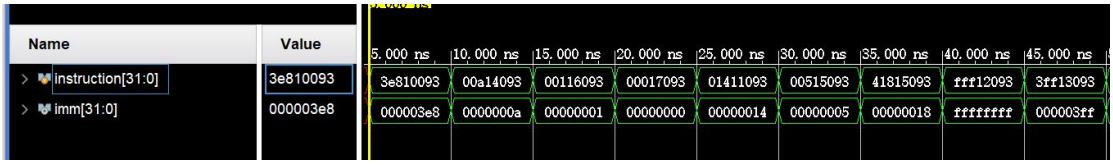
二、实验结果与分析

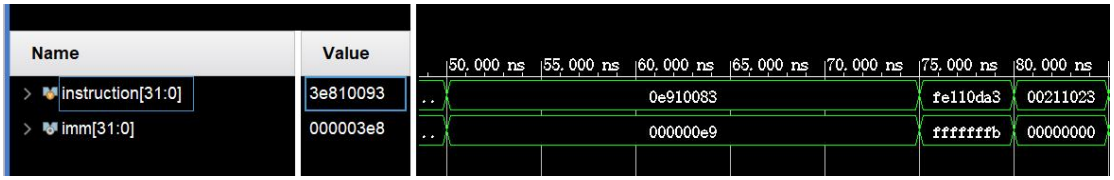
1.Imm_Gen 模块仿真
仿真代码如图所示：

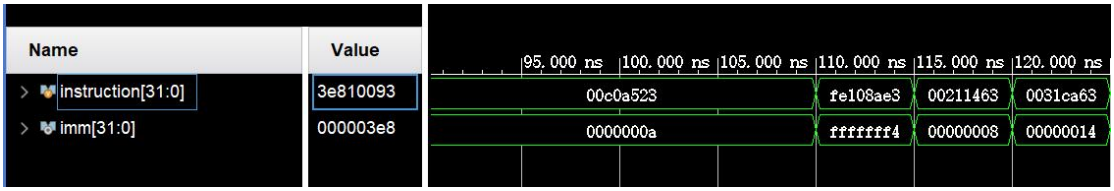
```
`timescale 1ns/1ps
`include "E:/CO_LABS/LAB4/Downloads/Lab4_header.vh"
module ImmGen_tb();
  reg [31:0]  instruction;
  wire[31:0]  imm;
  Imm_Gen m0 (.instruction(instruction), .imm(imm));
`define LET_INST_BE(inst) \
  instruction = inst; \
  #5;
  initial begin
    $dumpfile("E:/CO_LABS/LAB4/Downloads/ImmGen.vcd");
    $dumpvars(1, ImmGen_tb);
    #5;
    /* Test for 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)
    #20;
    /* Test for S-Type */
```

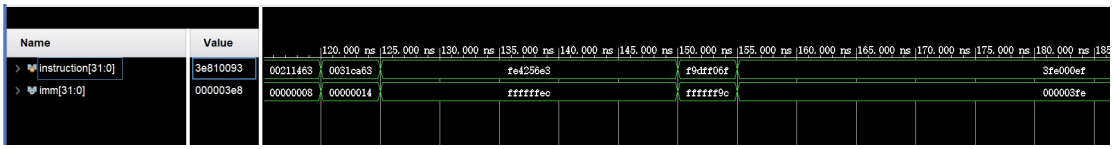
```
    `LET_INST_BE(32'hFE110DA3);   //sb x1, -5(x2)
    `LET_INST_BE(32'h00211023);   //sh x2, 0(x2)
    `LET_INST_BE(32'h00C0A523);   //sw x12, 10(x1)
    #20;
    /* Test for B-Type */
    `LET_INST_BE(32'hFE108AE3);   //beq x1, x1, -12
    `LET_INST_BE(32'h00211463);   //bne x2, x2, 8
    `LET_INST_BE(32'h0031CA63);   //blt x3, x3, 20
    `LET_INST_BE(32'hFE4256E3);   //bge x4, x4, -20
    #20;
    /* Test for J-Type */
    `LET_INST_BE(32'hF9DFF06F);   //jal x0, -100
    `LET_INST_BE(32'h3FE000EF);   //jal x1, 1023 NOTE: does ImmGen output 1023?
    #50; $finish();
  end
endmodule
```

我对 I\S\B\J 类型的测试分别做了立即数模块的仿真，仿真结果如下：









具体来说：比如对于 I 型操作，instruction=0x3E810093,语义是：addi x1, x2, 1000，产生的立即数是 0x3e8，符合预期；对于 B 型操作，instruction=0xFE108AE3，语义是：beq x1, x1, -12，产生的立即数是 0xFFFFFFF4，符合预期。其余的也可类似验证，不再赘述。

2.Control_Unit 模块仿真

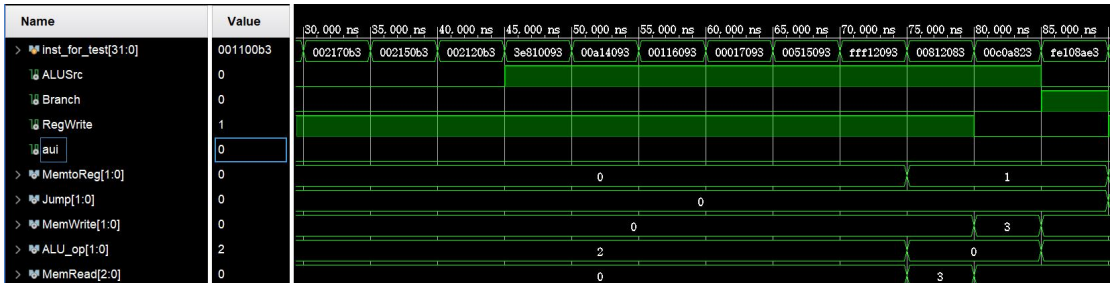
仿真代码如图所示：

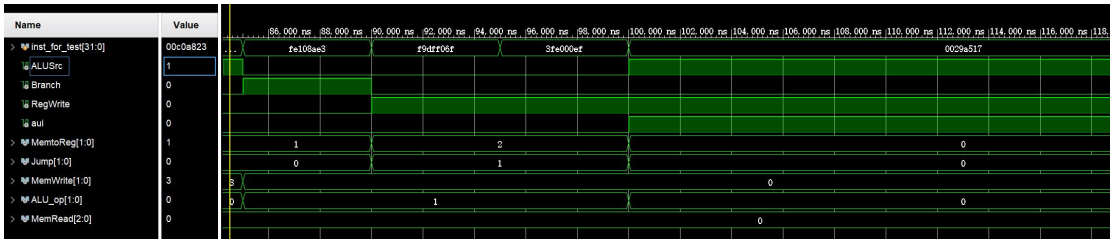
```
`timescale 1ns/1ps


`include "E:/CO_LABS/LAB4/Downloads/Lab4_header.vh"
```

```
module SCPU_ctrl_tb();
  reg [6:0]     opcode;
  reg [2:0]     funct3;
  wire          ALUSrc;
  wire [1: 0]   MemtoReg;
  wire [1:0]    Jump;
  wire          Branch;
  wire          RegWrite;
  wire [1:0]    MemWrite;
  wire [1:0]    ALU_op;
  wire          aui;
  wire [2:0]    MemRead;
  Control_Unit m0 (
    .opcode(opcode),
    .funct3(funct3),
    .ALUSrc(ALUSrc),
    .Jump(Jump),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .MemWrite(MemWrite),
    .RegWrite(RegWrite),
    .ALU_op(ALU_op),
    .aui(aui)
  );
  reg [31:0] inst_for_test;
`define LET_INST_BE(inst) \
  inst_for_test = inst; \
  opcode = inst_for_test[6:0]; \
  funct3 = inst_for_test[14:12]; \
  #5;
  initial begin
    $dumpfile("E:/CO_LABS/LAB4/Downloads/SCPU_ctrl.vcd");
    $dumpvars(1, SCPU_ctrl_tb);
    #5;

    #5;
    `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'h002150B3);   //srl x1, x2, x2
    `LET_INST_BE(32'h002120B3);   //slt x1, x2, x2
```

```
    `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'h00515093);   //srli x1, x2, 5
    `LET_INST_BE(32'hFFF12093);   //slti x1, x2, -1
    `LET_INST_BE(32'h00812083);   //lw x1, 8(x2)
    `LET_INST_BE(32'h00C0A823);   //sw x12, 16(x1)
    `LET_INST_BE(32'hFE108AE3);   //beq x1, x1, -12
    `LET_INST_BE(32'hF9DFF06F);   //jal x0, -100
    `LET_INST_BE(32'h3FE000EF);   //jal x1, 1023
    `LET_INST_BE(32'h0029a517);   //auipc x10,666


    #50; $finish();
  end
endmodule
```

我对 I\S\B\J\U 类型的测试分别做了立即数模块的仿真，仿真结果如下：





具体举例来说，比如对于 R 类操作，仅有 RegWrite 为 1，ALU_op 为 2，其余控制信号全部为 0；对于 Jal 操作（instruction=0x3FE000EF：jal x1, 1023）可以看到 RegWrite 和 Jump 为 1，MemtoReg 为 2，其余控制信号均为 0，符合预期；对于 auipc 操作：ALUSrc\RegWrite\aui 均为 1，其余信号全部为 0，符合预期。其余类型的指令控制信号产生也都符合预期，此处不再赘述。

3.SCPU 仿真

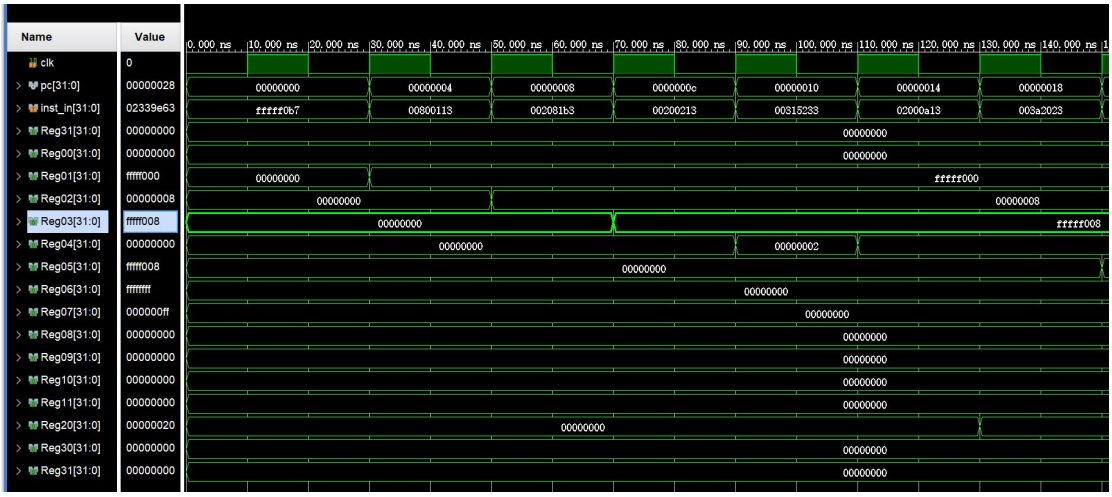
本次 SCPU 我进行了两次仿真。

第一次：我写了一段简单的汇编代码，初步测试了 SCPU 模块的正确性。具体汇编代码如下：

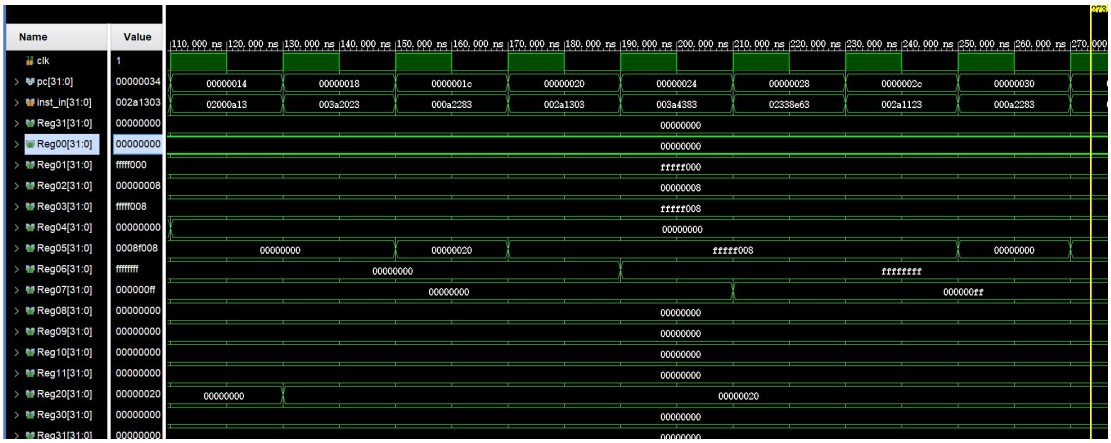
```
    #Write an assembly code to test all instructions in RISCV Integer
        lui x1,0xfffff #x1=fffff000

        # I R type
        addi x2,x0,8 #x2=00000008
```

```
        add x3,x1,x2  #x3=fffff008
        addi x4,x0,2 #x4=00000002
        srl x4,x2,x3 #x4=3ffffc02

        #Load Store
        addi x20,x0,0x20 #x20=00000020
        sw x3,0(x20) #mem[0x20]=fffff008
        lw x5,0(x20) #x5=fffff008
        lh x6,2(x20) #x6=
        lbu x7,3(x20) #x7=
        bne x7,x3,fail

        sh x2,2(x20)
        lw x5,0(x20) #x5=
        lh x6,2(x20) #x6=
        lbu x7,3(x20) #x7=

        #B type
        addi x8,x0,4 #x8=4
        addi x9,x0,6 #x9=6
        addi x10,x0,6 #x10=6
        beq x8,x9,fail #if failed
        beq x9,x10,next #branch success
        jal x0,fail
    next:
        bge x9,x8,next1
        jal x0,fail
    next1:
        #U type
        auipc x11,1#x11=
        jal x0,success

    fail:
        addi x31,x31,1
        jal fail

    success:
        addi x20,x30,1
        jal x0,success
```

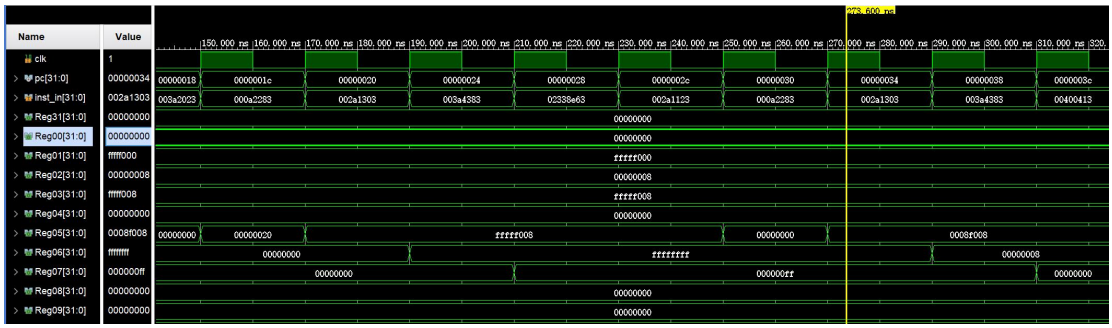
得到如下仿真结果：



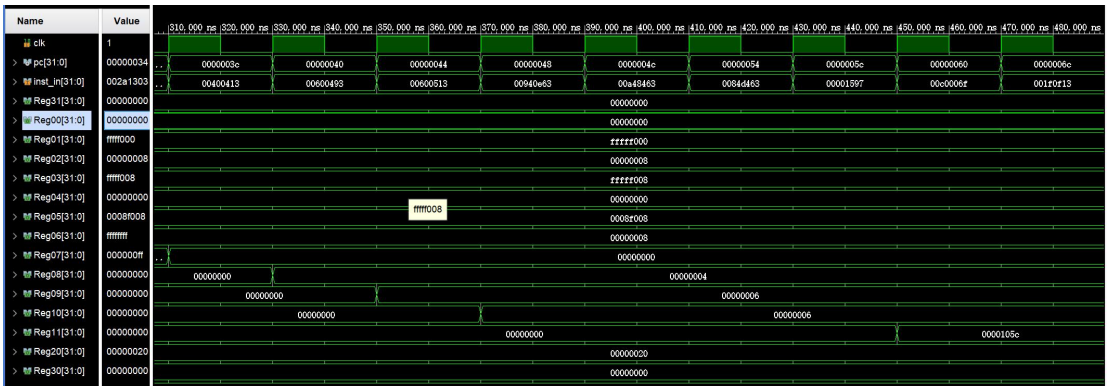
上面一张图展现了对 U、R、I(addi)类指令的仿真。可以看到，在前五条指令中，每次读入一条指令，目标寄存器的值就发生相应变化。比如第一条指令 fffff0b7，实际上是 lui x1,0xFFFFF,可以看 Reg01 的波形变化从 0 变成了 FFFFF000，符合预期；第二条指令是 addi x2,x0,8 可以看到 Reg02 成功从 0 变成了 00000008，符合预期。



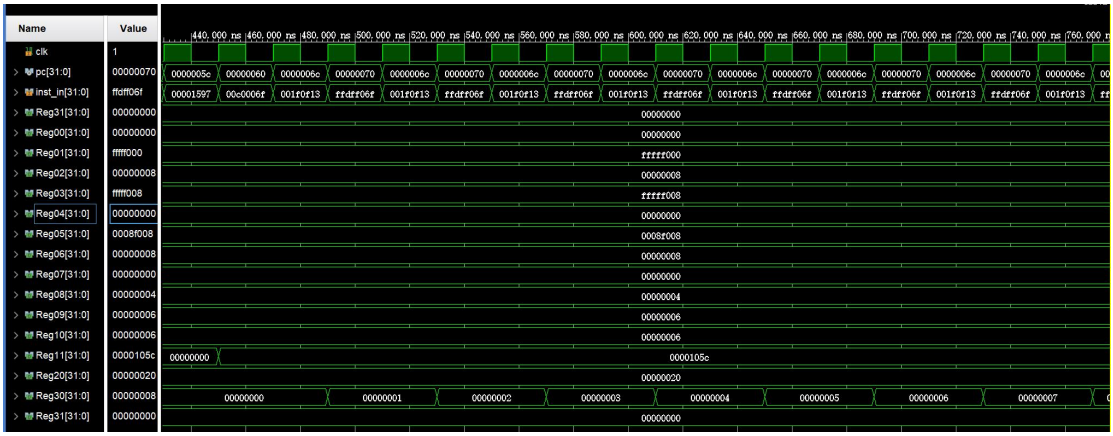
上图中验证了 S 类指令和 L 类指令。003a2023 的指令是 sw x3,0(x20)，意在将 fffff008 存入 RAM 中。接下来三条指令分别是 lw x5,0(x20) 、lh x6,2(x20) 、lbu x7,3(x20)，我们可以看到，三条指令后 Reg05、Reg06、Reg07 分别变成了 fffff008、ffffffff、000000ff，非常符合我们的预期，也就是说以上的 sw、lw、lh、lbu 操作都没有出问题。同理，我验证了 sh 等操作，波形如下所示，一样没有出问题。



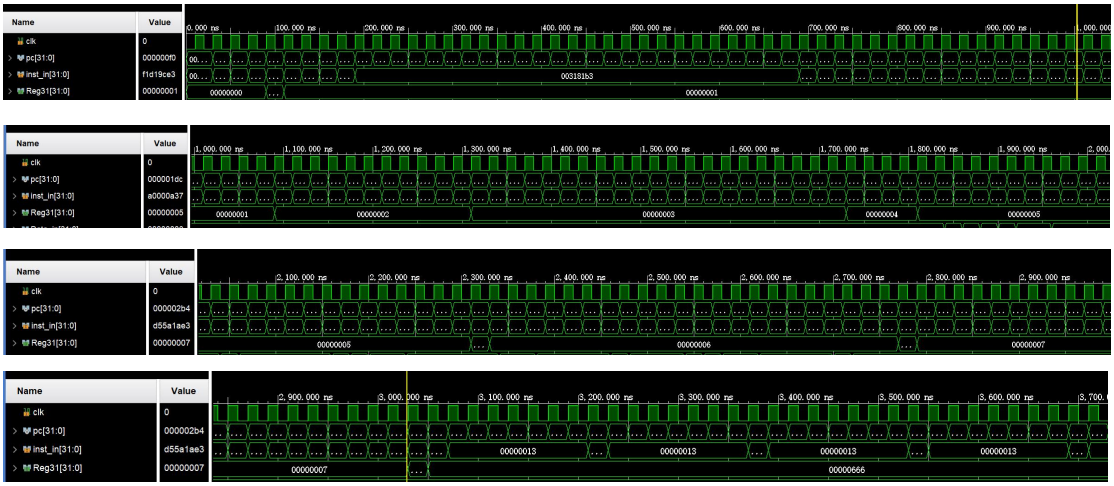
最后我在下图中验证了 Branch 类操作和 U 类操作。先令 x8\x9\x10 分别等于 4\6\6,可以看到 beq x8,x9 的指令并没有跳转（因为没有进入 fail 死循环，fail 死循环里 reg31 的值不停增加），beq x9，x10 的指令则发生了跳转。最后指令 00001597，实际上是 auipc x11,1 此处 pc 是 000005c，可以看到 Reg11 变成了 pc+imm<<12 的数值，也就是 0000105c，符合预期。



此外，J 类操作，在我实现代码跳转进入死循环的过程中已经得到验证。根据代码可以看到，如果符合预期，则进入 Reg30 不断自增的死循环；如果失败，则进入 Reg31 不断自增的死循环，根据以下波形（最后两行），仿真符合预期。



然后我进行了第二次仿真，仿真的 Imem 是来源于下板验收的代码，这里就不给出了，得到仿真结果如图所示：



可以看到，最终 Reg31 从 0 逐渐增加到 7，然后跳入死循环变成 666，完全符合预期。

三、讨论、心得

本次实验花了我很长的时间，主要原因在于对时序的理解不够透彻，导致了仿真没问题，但是一上板就漏洞百出的尴尬场面。历时将近一个月，终于彻底完工。以后再写 verilog 代码的时候一定会更加注意时序逻辑，不然像这次一样不停返工实在是太痛苦了 orz。

4-4 在下一页

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

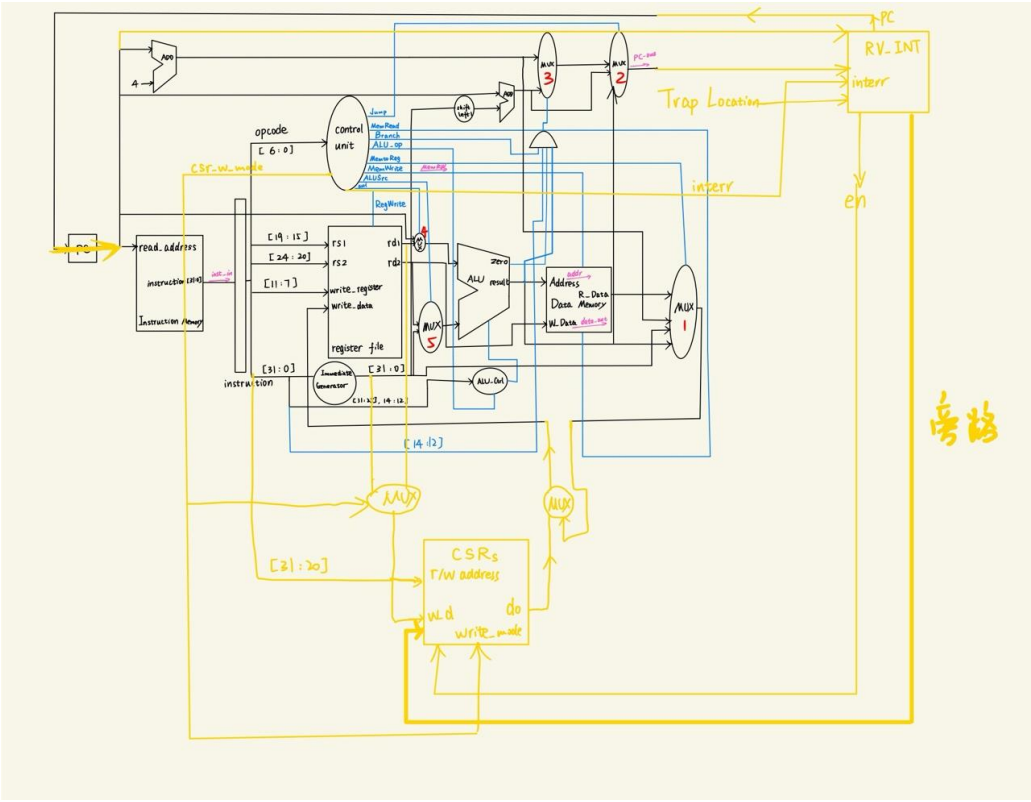
实验项目名称：SCPU 的异常与中断

学生姓名：汪珉凯　　学号：3220100975　　同组学生姓名：赵桉、马扬松

实验地点：紫金港东四 509 室　　实验日期：2024 年 5 月 8 日

目前我的 4-4 进行到了已经把 **RV_INT / CSRRegs** 两个重要模块写完，并把 **Control_Unit** 的调整做完了。但由于时间原因，我还没来得及做 Datapath 的连接与修改，因此在这里只能先给出 **RV_INT / CSRRegs / Control_Unit** 这 3 个模块的代码，以及我再原 datapath 通路图上做出的修改示意图。仿真结果和上板结果留待以后用自由时间补交 Orz。

新的 Datapath 通路图：



其中黄色部分表示修改的通路。可以看到，我新增了 RV_INT/CSRRegs/MUX 等模块，并从 Control_Unit 中多产生了两个控制信号 interr/csr_w_mode,分别用于记录中断类型以及具体属于 csrrw、csrrs、csrrc 中的哪一类指令。

CSRRegs 模块代码如下：

```
`timescale 1ns / 1ps
module CSRRegs(
    input clk, rst,
    input [11:0] raddr, waddr,
    input [31:0] wdata,
    input [1:0] csr_wsc_mode,

    input expt_int,//旁路输出信号
    input [31:0] mepc_bypass_in,
    input [31:0] mcause_bypass_in,
    input [31:0] mtval_bypass_in,
    input [31:0] mstatus_bypass_in,  // 为 mstatus 添加旁路输入
    input [31:0] mtvec_bypass_in,    // 为 mtvec 添加旁路输入
    output reg [31:0] rdata
);
    reg [31:0] mstatus, mtvec, mcause, mtval, mepc;
    // CSR 操作类型
    localparam CSR_W = 2'b01,  // csrrw
               CSR_S = 2'b10,  // csrrs
               CSR_C = 2'b11;  // csrrc
    // CSR 读逻辑
    always @(*) begin
        case (raddr)
            12'h300: rdata = mstatus;
            12'h305: rdata = mtvec;
            12'h341: rdata = mepc;
            12'h342: rdata = mcause;
            12'h343: rdata = mtval;
            default: rdata = 32'b0;
        endcase
    end
    // CSR 写逻辑
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            mstatus <= 32'b0;
            mtvec <= 32'b0;
            mcause <= 32'b0;
            mtval <= 32'b0;
            mepc <= 32'b0;
```

```
    end else begin
        if (expt_int) begin
            mepc <= mepc_bypass_in;
            mcause <= mcause_bypass_in;
            mtval <= mtval_bypass_in;
            mstatus <= mstatus_bypass_in; // 使用旁路数据更新 mstatus
            mtvec <= mtvec_bypass_in;    // 使用旁路数据更新 mtvec
        end else if (csr_wsc_mode != 2'b00) begin
            case (waddr)
                12'h300: begin
                    case (csr_wsc_mode)
                        CSR_W: mstatus <= wdata;
                        CSR_S: mstatus <= mstatus | wdata;
                        CSR_C: mstatus <= mstatus & ~wdata;
                    endcase
                end
                12'h305: begin
                    case (csr_wsc_mode)
                        CSR_W: mtvec <= wdata;
                        CSR_S: mtvec <= mtvec | wdata;
                        CSR_C: mtvec <= mtvec & ~wdata;
                    endcase
                end
                12'h341: begin
                    case (csr_wsc_mode)
                        CSR_W: mepc <= wdata;
                        CSR_S: mepc <= mepc | wdata;
                        CSR_C: mepc <= mepc & ~wdata;
                    endcase
                end
                12'h342: begin
                    case (csr_wsc_mode)
                        CSR_W: mcause <= wdata;
                        CSR_S: mcause <= mcause | wdata;
                        CSR_C: mcause <= mcause & ~wdata;
                    endcase
                end
                12'h343: begin
                    case (csr_wsc_mode)
                        CSR_W: mtval <= wdata;
                        CSR_S: mtval <= mtval | wdata;
                        CSR_C: mtval <= mtval & ~wdata;
                    endcase
                end
```

```
            endcase
        end
    end
end
endmodule
```

RV_INT 模块代码如下所示：

```
`timescale 1ns / 1ps

module RV_INT(
    input       clk,
    input       rst,
    input       INT0,            // 外部中断信号
    input [1:0] interr,          // 中断类型
    input [31:0] current_pc,
    input       l_access_fault,   // 数据访存不对齐
    input       j_access_fault,   // 跳转地址不对齐
    input [31:0] pc_next,         // 不发生意外的情况下 下一条 PC 值
    output       en,              // 用于控制寄存器堆、内存等器件的写使能
    output[31:0] pc,              // 将执行的指令 PC 值
    output [31:0] mepc_bypass_in,
    output [31:0] mcause_bypass_in,
    output [31:0] mtval_bypass_in,
    output [31:0] mstatus_bypass_in,  // 为 mstatus 添加旁路输入
    output [31:0] mtvec_bypass_in    // 为 mtvec 添加旁路输入
);
parameter PC_illegal = 32'h 00000004;
parameter PC_ecall = 32'h 00000008;
parameter PC_hardware = 32'h0000000C;
reg en0;
assign en = en0;
reg [31:0] pc0;
assign pc = pc0;
reg [31:0] mepc,mcause,mtval,mstatus,mtvec;
assign mepc_bypass_in = mepc;
assign mcause_bypass_in = mcause;
assign mtval_bypass_in = mtval;
assign mstatus_bypass_in = mstatus;
assign mtvec_bypass_in = mtvec
always @ (posedge clk or posedge rst or posedge INT0) begin
    // Reset
    if (rst) begin
        en0 <= 1;
```

```
        pc0 <= 0;
        mepc <= 0;
        mcause <= 0;
        mtval <= 0;
        mstatus <= 0;
        mtvec <= 0;
    end
    // Hardware triggered interrupt
    else if (INT0) begin
        if (en0) begin // Just entered interrupt triggered by hardware
            pc0 <= PC_hardware;
            en0 <= 0;
            mepc <= current_pc;
            mcause <= 32'h1000000b;
            mtval <= current_pc;
            mstatus <= 0;
            mtvec <= 32'h00000000;
        end
    end
    // Interruption
    else begin
        if (interr == 2'b01 && en0) begin // Just entered illegal instruction triggered interrupt
            pc0 <= PC_illegal;
            en0 <= 0;
            mepc <= pc_next;
            mcause <= 32'h00000002;
            mtval <= current_pc;
            mstatus <= 0;
            mtvec <= 32'h000000000;
        end
        else if (interr == 2'b10 && en0) begin // Just entered ecall triggered interrupt
            pc0 <= PC_ecall;
            en0 <= 0;
            mepc <= pc_next;
            mcause <= 32'h10000003;
            mtval <= current_pc;
            mstatus <= 0;
            mtvec <= 32'h00000000;
        end
        else if (interr == 2'b11 && (!en0)) begin // Just entered mret triggered interrupt
            pc0 <= mepc;
            en0 <= 1;
        end
```

```
        else begin
            pc0 <= pc_next;
        end
    end
end
endmodule
```

其中 mstatus 还没有完成正确赋值。

修改后的 Control_Unit 如下所示：

```
`timescale 1ns / 1ps


module Control_Unit(
    input [6:0] opcode,
    input [2:0] funct3,
    input [11:0] funct12,
    input [3:0] csr_kind,
    output ALUSrc,//0->use rs2 ; 1->use imm
    output [1:0] Jump,//00->PC+4/Branch ; 01->jal ; 10->jalr //this is the last mux before PC
    output Branch,
    output [2:0] MemRead, //001->lb ; 010->lh ; 011->lw ; 101->lbu ; 110->lhu
    output [1:0] MemtoReg,//00->ALU_result ; 01->Memory ; 10->PC+4 ; 11->imm
    output [1:0] MemWrite,//01->sb ; 10->sh ; 11->sw
    output RegWrite,
    output [1:0] ALU_op,//00->ld/sd/auipc ; 01->beq ; 10->R_type
    output aui,//1->auipc ; 0->others
    output [1:0] interr,//00:normal ;01: illegal ; 10: ecall ; 11:mret
    output [1:0] csr_wsc_mode//00-> not 3 kinds ; 01->cssrw ; 10->csrrs ; 11->csrrc
    );



reg ALUSrc0,RegWrite0,Branch0,aui0;
reg[1:0]  Jump0,ALU_op0,MemtoReg0,MemWrite0;
reg[2:0]  MemRead0;
reg [1:0] interr0;
reg [1:0] csr_wsc_mode0;
assign RegWrite=RegWrite0;
assign MemRead=MemRead0;
assign MemWrite=MemWrite0;
assign Branch=Branch0;
assign aui=aui0;
assign Jump=Jump0;
assign ALU_op=ALU_op0;
assign ALUSrc=ALUSrc0;
assign MemtoReg=MemtoReg0;
```

```
assign interr=interr0;
assign csr_wsc_mode=csr_wsc_mode0;
always @ (*) begin
    case(opcode)
        7'b0110011://R add
            begin
                ALUSrc0=0;
                MemtoReg0=2'b00;
                RegWrite0=1;
                MemRead0=3'b0;
                MemWrite0=2'b0;
                Branch0=0;
                Jump0=2'b00;
                ALU_op0=2'b10;
                aui0=0;
                interr0=2'b00;
                csr_wsc_mode0=2'b00;
            end
        7'b0010011://I  addi
            begin
                ALUSrc0=1;
                MemtoReg0=2'b00;
                RegWrite0=1;
                MemRead0=3'b0;
                MemWrite0=2'b0;
                Branch0=0;
                Jump0=2'b00;
                ALU_op0=2'b10;
                aui0=0;
                interr0=2'b00;
                csr_wsc_mode0=2'b00;
            end
        7'b0000011://I  ld
            begin
                ALUSrc0=1;
                MemtoReg0=2'b01;
                RegWrite0=1;
                // MemRead0=1;
                MemWrite0=2'b0;
                Branch0=0;
                Jump0=2'b00;
                ALU_op0=2'b00;
                aui0=0;
                interr0=2'b00;
```

```
            csr_wsc_mode0=2'b00;
            case(funct3)
                3'b000:MemRead0=3'b001;
                3'b001:MemRead0=3'b010;
                3'b010:MemRead0=3'b011;
                3'b100:MemRead0=3'b101;
                3'b101:MemRead0=3'b110;
            endcase
        end
    7'b0100011://S sd
        begin
            ALUSrc0=1;
            RegWrite0=0;
            MemRead0=3'b0;
            Branch0=0;
            Jump0=2'b00;
            ALU_op0=2'b00;
            aui0=0;
            interr0=2'b00;
            csr_wsc_mode0=2'b00;
            case(funct3)
                3'b000:MemWrite0=2'b01;
                3'b001:MemWrite0=2'b10;
                3'b010:MemWrite0=2'b11;
            endcase
        end
    7'b1100011://B beq
        begin
            ALUSrc0=0;
            RegWrite0=0;
            MemRead0=3'b0;
            MemWrite0=2'b0;
            Branch0=1;
            Jump0=2'b00;
            ALU_op0=2'b01;
            aui0=0;
            interr0=2'b00;
            csr_wsc_mode0=2'b00;
        end
    7'b1101111://J jal
        begin
            MemtoReg0=2'b10;
            RegWrite0=1;
            MemRead0=3'b0;
```

```
            MemWrite0=2'b0;
            Branch0=0;
            Jump0=2'b01;
            aui0=0;
            interr0=2'b00;
            csr_wsc_mode0=2'b00;
        end
    7'b1100111://J jalr
        begin
            ALUSrc0=1;
            MemtoReg0=2'b10;
            RegWrite0=1;
            MemRead0=3'b0;
            MemWrite0=2'b0;
            Branch0=0;
            Jump0=2'b10;
            ALU_op0=2'b00;
            aui0=0;
            interr0=2'b00;
            csr_wsc_mode0=2'b00;
        end
    7'b0110111://U lui
        begin
            MemtoReg0=2'b11;
            RegWrite0=1;
            MemRead0=3'b0;
            MemWrite0=2'b0;
            Branch0=0;
            Jump0=2'b00;
            aui0=0;
            interr0=2'b00;
            csr_wsc_mode0=2'b00;
        end
    7'b0010111://auipc
        begin
            aui0=1;
            ALUSrc0=1;
            ALU_op0=2'b00;
            MemWrite0=2'b0;
            MemRead0=3'b0;
            Branch0=0;
            Jump0=2'b00;
            MemtoReg0=2'b00;
            RegWrite0=1;
```

```
            interr0=2'b00;
            csr_wsc_mode0=2'b00;
        end

    7'b1110011:// system call
        begin
            ALUSrc0=0;
            MemtoReg0=2'b00;
            RegWrite0=0;
            MemRead0=3'b0;
            MemWrite0=2'b0;
            Branch0=0;
            Jump0=2'b00;
            ALU_op0=2'b00;
            aui0=0;
            case(funct12)
                12'h000: begin
                    interr0=2'b10; // ecall
                    csr_wsc_mode0=2'b00;
                end
                12'h302: begin
                    interr0=2'b11; // mret
                    csr_wsc_mode0=2'b00;
                end
                12'hc00: begin
                    interr0=2'b00; // csrr
                    case(csr_kind)
                        4'd1, 4'd5: csr_wsc_mode0=2'b01;
                        4'd2, 4'd6: csr_wsc_mode0=2'b10;
                        4'd3, 4'd7: csr_wsc_mode0=2'b11;
                        default: csr_wsc_mode0=2'b00;
                    endcase
                end
                default: begin
                    interr0=2'b01; // illegal
                    csr_wsc_mode0=2'b00;
                end
            endcase
        end

    default:
        begin
            ALUSrc0=0;
            MemtoReg0=2'b00;
```

```
                RegWrite0=0;
                MemRead0=3'b0;
                MemWrite0=2'b0;
                Branch0=0;
                Jump0=2'b00;
                ALU_op0=2'b00;
                aui0=0;
                interr0=2'b01;//illeagal
                csr_wsc_mode0=2'b00;
            end
    endcase
end
endmodule
```

思考题

思考体中给出的代码无法正确赋值，因为 ADDI 一个负数，实际上不是 0x00000EFF,而是 0xFFFFFEFF,因此直接用 addi 无法得到预期结果。按照原来的代码，仅仅修改一个字符，可以这样改：

lui t1, 0xDEADB

addi t1, t1, 0x0EEF