

《计算机组成与设计》

实验报告

姓名: 刘韬

学院: 竹可桢学院

专业: 人工智能

邮箱: 3220103422@zju.edu.cn

报告日期: 2024 年 5 月 3 日

1 SCPU 实现

这是 4-3 的实验报告。在本节中实现了全部的 SCPU 指令集，包括：

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

1.1 模块实现

1.1.1 datapath

首先给出 SCPU 的数据通路图，如下所示：

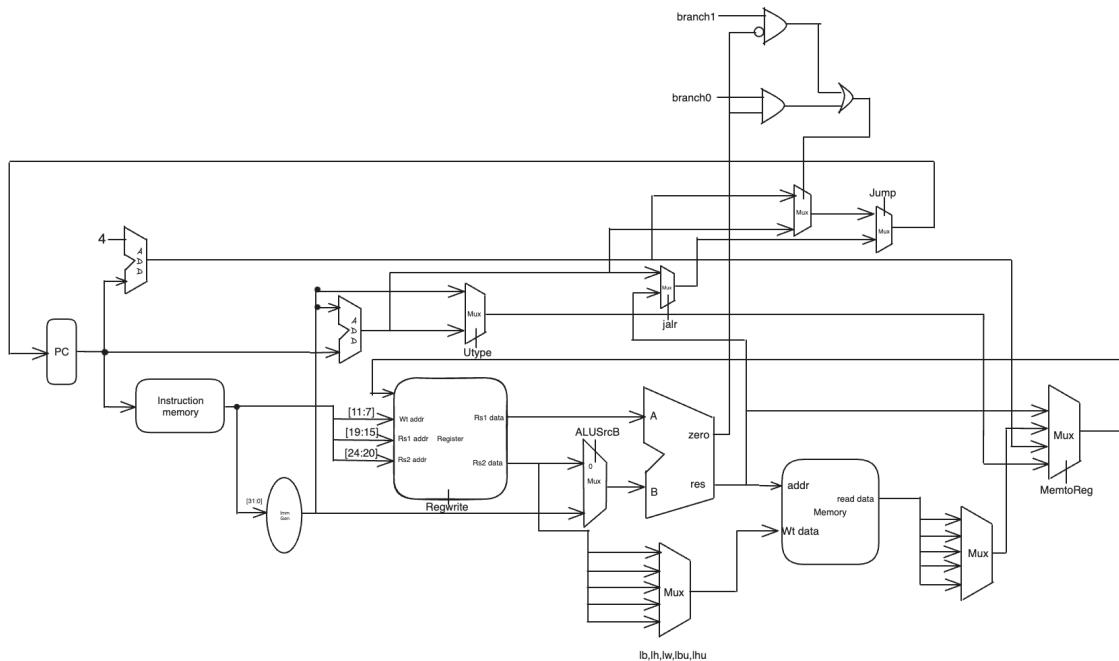


图 1: datapath (不含中断)

数据流主要由 PC 和寄存器文件。其中 PC 的值来源于 `PC+4, PC+imm, rs1 + imm`，根据一系列的判断信号进行选择，具体逻辑不在这里展开，下面结合具体的指令进行说明。另一部分是写到寄存器的值，来源是 `ALU_res, memory, PC+4, PC+imm`，也是根据一系列的判断信号进行选择。另外涉及位长的转换也是通过代码实现的。

接着根据 datapath 将 SCPU 的各个元件进行连接：使用了提供的头文件

`"Lab4_header.vh"`，其中使用到的元件包括 `ImmGen, MUX, ALU, REG32, Regs`，其中 `ImmGen` 用于生成立即数，`MUX` 用于选择输入，`ALU` 用于进行运算，`REG32` 用于存储 PC 的值，`Regs` 用于存储寄存器的值。`ImmGen` 是本次实验新实现的，其余都是之前实验完成的，就不再赘述。

以下是 ImmGen 的代码:

```
1 `include "Lab4_header.vh"
2
3 module ImmGen(
4     input [2:0] ImmSel,
5     input [31:0] inst_field,
6     output reg [31:0] Imm_out
7 );
8
9     always @(*) begin
10         case(ImmSel)
11             `IMM_SEL_I: Imm_out = {{20{inst_field[31]}},
12             inst_field[31:20]};
13             `IMM_SEL_S: Imm_out = {{20{inst_field[31]}}, inst_field[31:25],
14             inst_field[11:7]};
15             `IMM_SEL_B: Imm_out = {{19{inst_field[31]}}, inst_field[31],
16             inst_field[7], inst_field[30:25], inst_field[11:8], 1'b0};
17             `IMM_SEL_J: Imm_out = {{11{inst_field[31]}}, inst_field[31],
18             inst_field[19:12], inst_field[20], inst_field[30:21], 1'b0};
19             `IMM_SEL_U: Imm_out = {inst_field[31:12], 12'b0};
20             default: Imm_out = 32'b0;
21         endcase
22     end
23
24 endmodule
```

这是根据上面 DataPath 图连线的代码:

```
1 `include "Lab4_header.vh"
2
3 module DataPath(
4     input clk,
5     input rst,
6     input [31:0] inst_field,
7     input [31:0] Data_in,
8     input [3:0] ALU_Control,
9     input [2:0] ImmSel,
10    input [1:0] MemtoReg,
11    input ALUSrc_B,
12    input Jump,
13    input Branch0,
14    input Branch1,
15    input RegWrite,
16    input jalr,
17    input [2:0] wordtype, // 0:byte 1:half 2:word 最高位 0 signed 1
18    unsigned
19        input utype,
20        `RegFile_Regs_Outputs
21    output reg [3:0] RAM_wt, //RAM 写使能
22    output [31:0] PC_out,
23    output [31:0] Data_out,
24    output [31:0] ALU_out,
25    output [31:0] Reg_in
26 );
27
28    wire [31:0] Imm_out;
29    wire [31:0] Rs1_data, Rs2_data;
30    wire [31:0] memory_in;
31    reg [31:0] reg_in, data_temp;
```

```

31     wire zero, s4;
32     wire [31:0] o0 ,o1, o2, o3, o4, o5;
33     wire [31:0] c0, c1;
34
35     ImmGen ImmGen(
36         .ImmSel(ImmSel),
37         .inst_field(inst_field),
38         .Imm_out(Imm_out)
39     );
40
41     assign c0 = PC_out + 32'd4;
42     assign c1 = PC_out + Imm_out;
43     assign s4 = (Branch0 & zero) | (Branch1 & ~zero);
44     assign Data_out = data_temp;
45     assign Reg_in = reg_in;
46     assign memory_in = Data_in >> ({3'b000, ALU_out[1:0]} << 2'b11);
47
48
49 //input
50 always @(*) begin
51     case(wordtype)
52         3'b000 : reg_in <= {{24{memory_in[7]}}, memory_in[7:0]]; //
53         signed byte
54             3'b100 : reg_in <= {24'b0, memory_in[7:0]}; //unsigned byte
55             3'b001 : reg_in <= {{16{memory_in[15]}}, memory_in[15:0]}; //
56         signed half
57             3'b101 : reg_in <= {16'b0, memory_in[15:0]}; //unsigned half
58             3'b010 : reg_in <= memory_in; //signed word
59             default: reg_in <= 32'b0;
60     endcase
61 end
62 //store
63 always @(*) begin
64     case(wordtype[2:0])
65         3'b000 : begin
66             case(ALU_out[1:0])
67                 2'b00 : begin
68                     data_temp <= {24'b0, Rs2_data[7:0]};
69                     RAM_wt <= 4'b0001;
70                 end
71                 2'b01 : begin
72                     data_temp <= {16'b0, Rs2_data[7:0], 8'b0};
73                     RAM_wt <= 4'b0010;
74                 end
75                 2'b10 : begin
76                     data_temp <= {8'b0, Rs2_data[7:0], 16'b0};
77                     RAM_wt <= 4'b0100;
78                 end
79                 2'b11 : begin
80                     data_temp <= {Rs2_data[7:0], 24'b0};
81                     RAM_wt <= 4'b1000;
82                 end
83             endcase
84         end
85         3'b001 : begin
86             case(ALU_out[1:0])
87                 2'b00 : begin
88                     data_temp <= {16'b0, Rs2_data[15:0]};
89                     RAM_wt <= 4'b0011;
90                 end
91                 2'b01 : begin
92                     data_temp <= {8'b0, Rs2_data[15:0], 8'b0};
93                 end
94             endcase
95         end
96     end
97 
```

```

91           RAM_wt <= 4'b0110;
92       end
93   2'b10 : begin
94       data_temp <= {Rs2_data[15:0], 16'b0};
95       RAM_wt <= 4'b1100;
96   end
97   default : begin
98       data_temp <= 0;
99       RAM_wt <= 0;
100      end
101     endcase
102 end
103 3'b010 : begin
104     data_temp <= Rs2_data;
105     RAM_wt <= 4'b1111;
106 end
107 default : begin
108     data_temp <= 0;
109     RAM_wt <= 0;
110    end
111  endcase
112 end
113
114 MUX2T1_32 m0(
115     .I0(Rs2_data),
116     .I1(Imm_out),
117     .s(ALUSrc_B),
118     .o(o0)
119 );
120
121 MUX4T1_32 m1(
122     .I0(ALU_out),
123     .I1(reg_in),
124     .I2(c0),
125     .I3(o2),
126     .s(MemtoReg),
127     .o(o1)
128 );
129
130 MUX2T1_32 m2(
131     .I0(Imm_out),
132     .I1(c1),
133     .s(utype),
134     .o(o2)
135 );
136
137 MUX2T1_32 m3(
138     .I0(c1),
139     .I1(ALU_out),
140     .s(jalr),
141     .o(o3)
142 );
143
144 MUX2T1_32 m4(
145     .I0(c0),
146     .I1(c1),
147     .s(s4),
148     .o(o4)
149 );
150
151 MUX2T1_32 m5(
152     .I0(o4),
153     .I1(o3),

```

```

154     .s(Jump),
155     .o(o5)
156 );
157
158 Regs Regs(
159     .clk(clk),
160     .rst(rst),
161     .Rs1_addr(inst_field[19:15]),
162     .Rs2_addr(inst_field[24:20]),
163     .Wt_addr(inst_field[11:7]),
164     .Wt_data(o1),
165     `RegFile_Regs_Arguments
166     .RegWrite(RegWrite),
167     .Rs1_data(Rs1_data),
168     .Rs2_data(Rs2_data)
169 );
170
171 ALU ALU(
172     .A(Rs1_data),
173     .B(o0),
174     .ALU_operation(ALU_Control),
175     .res(ALU_out),
176     .zero(zero)
177 );
178
179 REG32 PC(
180     .clk(clk),
181     .rst(rst),
182     .CE(1'b1),
183     .D(o5),
184     .Q(PC_out)
185 );
186
187 endmodule

```

上面代码中的 `memory_in` 是根据读写信号的具体位置进行转换的，因为读入的指令都是一个字长的，需要移动到正确的位置。并且根据写的类型进行扩展。几个选择器是根据下面模块给的信号进行选择的。

1.1.2 CPU_ctrl

第二部分是 `CPU_ctrl` 模块，用于控制 `datapath` 中的各个元件。根据输入的指令，输出一系列的控制信号，这一部分是通过组合逻辑实现的，没有涉及时序问题：

```

1 `include "Lab4_header.vh"
2
3 module SCPU_ctrl(
4     input [4:0]OPcode,
5     input [2:0]Fun3,
6     input Fun7,
7     input MIO_ready, // CPU_wait
8     output reg [2:0]ImmSel,
9     output reg ALUSrc_B, // ALUB 的值, 0 为 Rs2, 1 为 Imm
10    output reg [1:0]MemtoReg, // 写回寄存器堆的值, 00 为 ALUout, 01 为 Memout, 10
        为 PC+4, 11 为 Utype 的值
11    output reg Jump, // 是否跳转
12    output reg Branch0, // beq,bge,bgeu
13    output reg Branch1, // bne,blt,bltu

```

```

14     output reg RegWrite, // 是否写回寄存器堆
15     output reg MemRW, // 是否读写内存
16     output reg [3:0]ALU_Control, // ALU 的操作
17     output reg jalr, // 是否是 jalr
18     output reg utype, // Uttype 的类型, 0 为 lui 即 imm<<12, 1 为 auipc 即
PC+imm<<12 我的实现是在生成立即数的时候就进行移位
19     output reg [2:0] wordtype, // 0:byte 1:half 2:word 最高位 0 signed 最高位
1 unsigned
20     output reg CPU_MIO
21 );
22     reg [1:0] ALU_op;
23     `define CPU_ctrl_signals {ALUSrc_B, MemtoReg, RegWrite, MemRW, Branch1,
Branch0, Jump, ALU_op}
24     always @(*) begin
25       case (OPcode)
26         /* R-type */
27         `OPCODE_ALU : begin
28           ImmSel <= 3'b000;
29           `CPU_ctrl_signals <= 10'b0_00_1_0_00_0_10;
30           jalr <= 1'b0;
31           utype <= 1'b0;
32           wordtype <= 3'b000;
33         end
34         /* I-type */
35         `OPCODE_ALU_IMM : begin
36           ImmSel <= `IMM_SEL_I;
37           `CPU_ctrl_signals <= 10'b1_00_1_0_00_0_11;
38           jalr <= 1'b0;
39           utype <= 1'b0;
40           wordtype <= 3'b000;
41         end
42         `OPCODE_LOAD : begin
43           ImmSel <= `IMM_SEL_I;
44           `CPU_ctrl_signals <= 10'b1_01_1_0_00_0_00;
45           jalr <= 1'b0;
46           utype <= 1'b0;
47           case (Fun3)
48             `FUNC_BYTE : wordtype <= 3'b000;
49             `FUNC_HALF : wordtype <= 3'b001;
50             `FUNC_WORD : wordtype <= 3'b010;
51             `FUNC_BYTE_UNSIGNED : wordtype <= 3'b100;
52             `FUNC_HALF_UNSIGNED : wordtype <= 3'b101;
53             default: wordtype <= 3'b000;
54           endcase
55         end
56         `OPCODE_JALR : begin
57           ImmSel <= `IMM_SEL_I;
58           `CPU_ctrl_signals <= 10'b1_10_1_0_00_1_00;
59           jalr <= 1'b1;
60           utype <= 1'b0;
61           wordtype <= 3'b000;
62         end
63         /* S-type */
64         `OPCODE_STORE : begin
65           ImmSel <= `IMM_SEL_S;
66           `CPU_ctrl_signals <= 10'b1_00_0_1_00_0_00;
67           jalr <= 1'b0;
68           utype <= 1'b0;
69           case(Fun3)
70             `FUNC_BYTE : wordtype <= 3'b000;
71             `FUNC_HALF : wordtype <= 3'b001;
72             `FUNC_WORD : wordtype <= 3'b010;

```

```

73           default: wordtype <= 3'b000;
74       endcase
75   end
76 /* B-type */
77 `OPCODE_BRANCH : begin
78     ImmSel <= `IMM_SEL_B;
79     case (Fun3)
80         `FUNC_EQ : `CPU_ctrl_signals <= 10'b0_00_0_0_01_0_01;
81         `FUNC_NE : `CPU_ctrl_signals <= 10'b0_00_0_0_10_0_01;
82         `FUNC_LT : `CPU_ctrl_signals <= 10'b0_00_0_0_10_0_01;
83         `FUNC_GE : `CPU_ctrl_signals <= 10'b0_00_0_0_01_0_01;
84         `FUNC_LTU : `CPU_ctrl_signals <= 10'b0_00_0_0_10_0_01;
85         `FUNC_GEU : `CPU_ctrl_signals <= 10'b0_00_0_0_01_0_01;
86         default: `CPU_ctrl_signals <= 10'b0_00_0_0_00_0_00;
87     endcase
88     jalr <= 1'b0;
89     utype <= 1'b0;
90     wordtype <= 3'b000;
91 end
92 /* J-type */
93 `OPCODE_JAL : begin
94     ImmSel <= `IMM_SEL_J;
95     `CPU_ctrl_signals <= 10'b0_10_1_0_00_1_00;
96     jalr <= 1'b0;
97     utype <= 1'b0;
98     wordtype <= 3'b000;
99 end
100 /* U-type */
101 `OPCODE_LUI : begin
102     ImmSel <= `IMM_SEL_U;
103     `CPU_ctrl_signals <= 10'b0_11_1_0_00_0_00;
104     jalr <= 1'b0;
105     utype <= 1'b0;
106     wordtype <= 3'b000;
107 end
108 `OPCODE_AUIPC : begin
109     ImmSel <= `IMM_SEL_U;
110     `CPU_ctrl_signals <= 10'b0_11_1_0_0_00_00;
111     jalr <= 1'b0;
112     utype <= 1'b1;
113     wordtype <= 3'b000;
114 end
115     default: ImmSel <= 3'b0;
116 endcase
117
118 case (ALU_op)
119     2'b00 : ALU_Control <= `ALU_OP_ADD;
120     2'b01 : begin
121         case (Fun3)
122             `FUNC_EQ : ALU_Control <= `ALU_OP_SUB;
123             `FUNC_NE : ALU_Control <= `ALU_OP_SUB;
124             `FUNC_LT : ALU_Control <= `ALU_OP_SLT;
125             `FUNC_GE : ALU_Control <= `ALU_OP_SLT;
126             `FUNC_LTU : ALU_Control <= `ALU_OP_SLTU;
127             `FUNC_GEU : ALU_Control <= `ALU_OP_SLTU;
128             default: ALU_Control <= `ALU_OP_SUB;
129         endcase
130     end
131     /* R-type */
132     2'b10 : begin
133         case (Fun3)
134             `FUNC_ADD : ALU_Control <= (Fun7 == 1'b0) ?
`ALU_OP_ADD : `ALU_OP_SUB;

```

```

135      `FUNC_SL : ALU_Control <= `ALU_OP_SLL;
136      `FUNC_SLT : ALU_Control <= `ALU_OP_SLT;
137      `FUNC_SLTU : ALU_Control <= `ALU_OP_SLTU;
138      `FUNC_XOR : ALU_Control <= `ALU_OP_XOR;
139      `FUNC_SR : ALU_Control <= (Fun7 == 1'b0) ? `ALU_OP_SRL :
`ALU_OP_SRA;
140          `FUNC_OR : ALU_Control <= `ALU_OP_OR;
141          `FUNC_AND : ALU_Control <= `ALU_OP_AND;
142          default: ALU_Control <= `ALU_OP_ADD;
143      endcase
144  end
145  2'b11 : begin
146      case (Fun3)
147          `FUNC_ADD : ALU_Control <= `ALU_OP_ADD;
148          `FUNC_SL : ALU_Control <= `ALU_OP_SLL;
149          `FUNC_SLT : ALU_Control <= `ALU_OP_SLT;
150          `FUNC_SLTU : ALU_Control <= `ALU_OP_SLTU;
151          `FUNC_XOR : ALU_Control <= `ALU_OP_XOR;
152          `FUNC_SR : ALU_Control <= (Fun7 == 1'b0) ?
`ALU_OP_SRL : `ALU_OP_SRA;
153              `FUNC_OR : ALU_Control <= `ALU_OP_OR;
154              `FUNC_AND : ALU_Control <= `ALU_OP_AND;
155              default: ALU_Control <= `ALU_OP_ADD;
156          endcase
157      end
158      default: ALU_Control <= `ALU_OP_ADD;
159  endcase
160  CPU_MIO <= 1'b0;
161 end
162
163 endmodule

```

为了支持全部的指令，在所给的接口上加了 Branch0,Branch1,jalr,utype,wordtype 几个包括跳转信号、位宽信号。

1.2 仿真测试

1.2.1 IMM

1.2.1.1 测试代码

```

1 `timescale 1ns/1ps
2
3 `include "Lab4_header.vh"
4 `include "../source/ImmGen.v"
5
6 module ImmGen_tb();
7     reg [2:0] ImmSel;
8     reg [31:0] inst_field;
9     wire[31:0] Imm_out;
10
11     ImmGen m0 (.ImmSel(ImmSel), .inst_field(inst_field), .Imm_out(Imm_out));
12
13 `define LET_INST_BE(inst) \
14     inst_field = inst; \

```

```

15  #5;
16
17 initial begin
18   $dumpfile("ImmGen.vcd");
19   $dumpvars(1, ImmGen_tb);
20
21   #5;
22   /* Test for I-Type */
23   ImmSel = `IMM_SEL_I;
24   `LET_INST_BE(32'h3E810093); //addi x1, x2, 1000
25   `LET_INST_BE(32'h00A14093); //xori x1, x2, 10
26   `LET_INST_BE(32'h00116093); //ori x1, x2, 1
27   `LET_INST_BE(32'h00017093); //andi x1, x2, 0
28   `LET_INST_BE(32'h01411093); //slli x1, x2, 20
29   `LET_INST_BE(32'h00515093); //srli x1, x2, 5
30   `LET_INST_BE(32'h41815093); //srai x1, x2, 24
31   `LET_INST_BE(32'hFFF12093); //slti x1, x2, -1
32   `LET_INST_BE(32'h3FF13093); //sltiu x1, x2, 1023
33   `LET_INST_BE(32'h0E910083); //lb x1, 233(x2)
34
35   #20;
36   /* Test for S-Type */
37   ImmSel = `IMM_SEL_S;
38   `LET_INST_BE(32'hFE110DA3); //sb x1, -5(x2)
39   `LET_INST_BE(32'h00211023); //sh x2, 0(x2)
40   `LET_INST_BE(32'h00C0A523); //sw x12, 10(x1)
41
42   #20;
43   /* Test for B-Type */
44   ImmSel = `IMM_SEL_B;
45   `LET_INST_BE(32'hFE108AE3); //beq x1, x1, -12
46   `LET_INST_BE(32'h00211463); //bne x2, x2, 8
47   `LET_INST_BE(32'h0031CA63); //blt x3, x3, 20
48   `LET_INST_BE(32'hFE4256E3); //bge x4, x4, -20
49
50   #20;
51   /* Test for J-Type */
52   ImmSel = `IMM_SEL_J;
53   `LET_INST_BE(32'hF9DFF06F); //jal x0, -100
54   `LET_INST_BE(32'h3FE000EF); //jal x1, 1023 NOTE: does ImmGen output
55   1023?
56   ImmSel = `IMM_SEL_U;
57   `LET_INST_BE(32'h003e8537); //lui x10, 1000
58   `LET_INST_BE(32'h003e8517); //auipc x10, 1000
59
60   #20;
61   #50; $finish();
62 end
63 endmodule

```

1.2.1.2 仿真波形

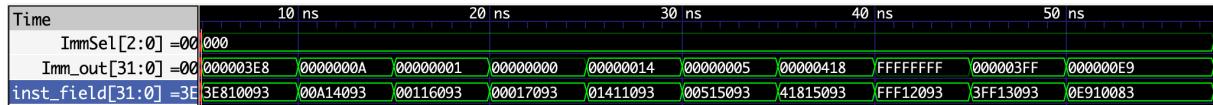


图 2: Imm 仿真 1

- I-type 立即数，有符号拓展到 32 位

Time	80 ns	90 ns	100 ns	110 ns	120 ns
ImmSel[2:0] =00001				010	
Imm_out[31:0] =00FFFFFFFB	00000000	0000000A		FFFFFFFFFF4	00000008 00000014
inst_field[31:0] =3EFE110DA3	00211023	00C0A523		FE108AE3	00211463 0031CA63

图 3: Imm 仿真 2

- S-type,B-type

Time	130 ns	140 ns	150 ns	160 ns	170 ns
ImmSel[2:0] =0010			011		
Imm_out[31:0] =00FFFFFFEC			FFFFFFFFFF9C	000003FE	
inst_field[31:0] =3EF44256E3			F9DFF06F	3FE000EF	

图 4: Imm 仿真 3

- B-type,J-type

与所给的标准仿真波形进行对比，发现仿真结果正确

1.2.2 SCPU 仿真

1.2.2.1 测试代码

仿真指令与所给的测试代码一致，最后结果是 `reg_31` 的值为 666。为了简化，建立了一个由 SCPU, RAM, ROM 组成的测试平台 testbench，如下：

```

1 module testbench(
2     input clk,
3     input rst
4 );
5
6     /* SCPU ???? */
7     wire [31:0] Addr_out;
8     wire [31:0] Data_out;
9     wire CPU_MIO;
10    wire MemRW;
11    wire [31:0] PC_out;
12    wire [3:0] RAM_wt;
13    /* RAM ?? */
14    wire [31:0] douta;
15    /* ROM ?? */
16    wire [31:0] spo;
17
18    SCPU u0(
19        .clk(clk), .rst(rst),
20        .Data_in(douta),
21        .MIO_ready(CPU_MIO),
22        .inst_in(spo),
23        .Addr_out(Addr_out),
24        .Data_out(Data_out),
25        .CPU_MIO(CPU_MIO),
26        .MemRW(MemRW),
27        .RAM_wt(RAM_wt),
28        .PC_out(PC_out)
29    );
30
31    myRAM2 u1(
32        .clka(~clk),
33        .wea({4{MemRW}} & RAM_wt),
34        .addra(Addr_out[11:2]),
35        .dina(Data_out),
36        .douta(douta)

```

```

37 );
38
39 myROM3 u2(
40     .a(PC_out[11:2]),
41     .spo(spo)
42 );
43
44 endmodule

```

使用的测试代码如下：

```

1 module testbench_tb();
2
3     reg clk;
4     reg rst;
5
6     testbench m0(.clk(clk), .rst(rst));
7
8     initial begin
9         clk = 1'b0;
10        rst = 1'b1;
11        #5;
12        rst = 1'b0;
13    end
14
15    always #50 clk = ~clk;
16 endmodule

```

指令都已经写入 ROM 中了，所以只需要给时钟信号就可以让 CPU 开始工作了。

1.2.2.2 仿真波形

根据验收代码的要求，我们只需要看最后 Reg31 的结果即可，如下：

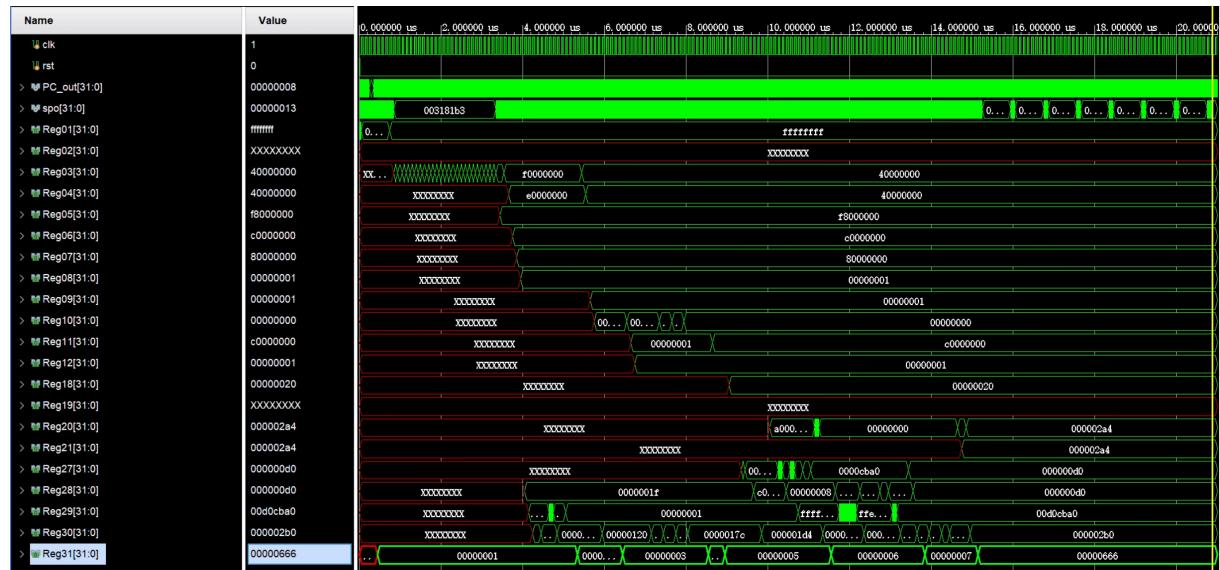


图 5：SCPU 仿真

可以看到仿真结果正确，最后 Reg31 的值为 0x666。并且从 PC 的地址也可以看出，顺利进入了 dummy 死循环。

1.3 下板处理

重新切换回 Lab2 完成的实验平台，将实现好的 SCPU 替换掉原来的 CPU，然后生成 bit 文件，进行下板验证，下板过程中我使用串口来显示寄存器的值和一些中间信号。

以下是在最后进入 dummy 死循环后的寄存器的值：

```
RV32I Single Cycle CPU

pc: 00000014    inst: 00000013

x0: 00000000    ra: FFFFFFFF    sp: 00000000    gp: 40000000    tp: 40000000
t0: F8000000    t1: C0000000    t2: 80000000    s0: 00000001    s1: 00000001
a0: 00000000    a1: C0000000    a2: 00000001    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000    s11: 000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000666

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 01 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0      is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1      cmp_ctrl: 0
alu_res: 00000000      cmp_res: 0

is_branch: 0    is_jal: 0      is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: F0000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000000      mcause: 00000000      mepc: 00000000    mtval: 00000000
mtvec: 00000000      mie: 00000000      mip: 00000000
```

图 6: SCPU 下板结果

可以看到最后 Reg31 的值为 0x666，而且也进入了 dummy 死循环。说明下板验证也同样正确。

2 中断处理

中断处理在 SCPU 的基础上增加了特权级寄存器，并通过特权集寄存器的读写实现了 `csrrw, csrrs, csrrc, csrrwi, csrrsi, csrrci, ecall, mret` 等指令，并且编写了 trap 程序用于处理中断异常。我们实现的是整个流程，具体的异常处理略过。

2.1 模块实现

2.1.1 CSR 寄存器

实现逻辑分为两部分，一部分用于接受 CSR 指令，另一部分用于处理中断。首先是 CSR 指令的实现，主要是根据 CSR 指令的不同模式实现读和写的操作，具体的实现如下：

```
1 module CSRRegs(
2     input clk, rst,
3     input[11:0] read_addr, write_addr,           // 读写 CSR 寄存器的地址
4     input[31:0] write_data,                      // 写入 CSR 寄存器的数据
5     input csr_w,                                // 写使能
6     input[1:0] csr_wsc_mode,                     // 写入 CSR 寄存器的模式
7     input by_en,                                // 旁路使能
8     input [31:0] by_mstatus, by_mtvec, by_mcause, by_mtval, by_mepc, // 旁路
9     // 输入，用于中断修改寄存器的值
10    output [31:0] read_data,                   // 读出 CSR 寄存器的数据
11    output [31:0] mstatus_o,
12    output [31:0] mtvec_o,
13    output [31:0] mcause_o,
14    output [31:0] mtval_o,
15    output [31:0] mepc_o
16 );
17     reg [31:0] mstatus;      // MSTATUS
18     reg [31:0] mtvec;       // MTVEC
19     reg [31:0] mcause;      // MCAUSE
20     reg [31:0] mtval;       // MTVAL
21     reg [31:0] mepc;        // MEPC
22
23     assign mstatus_o = mstatus;
24     assign mtvec_o = mtvec;
25     assign mcause_o = mcause;
26     assign mtval_o = mtval;
27     assign mepc_o = mepc;
28
29     // reg [31:0] mstatus, mtvec, mcause, mtval, mepc;
30     // read_data
31
32     assign read_data = (read_addr == 12'h300) ? mstatus :
33             (read_addr == 12'h305) ? mtvec :
34             (read_addr == 12'h342) ? mcause :
35             (read_addr == 12'h343) ? mtval :
36             (read_addr == 12'h341) ? mepc : 0;
37
38     // write_data
39     always @(posedge clk or posedge rst) begin
40         if(rst == 1) begin
41             mstatus <= 0;
42             mtvec <= 32'h50; // to modify
43             mcause <= 0;
44             mtval <= 0;
```

```

44         mepc <= 0;
45     end
46     else begin
47         if (csr_w == 1 && write_data != 0) begin
48             case(read_addr)
49                 12'h300: mstatus <= (csr_wsc_mode == 2'b01) ?
50                     write_data :
51                         (write_data | mstatus) :
52                             (~write_data & mstatus) : mstatus;
53                             12'h305: mtvec <= (csr_wsc_mode == 2'b01) ?
54                                 write_data :
55                                     (write_data | mtvec) :
56                                         (~write_data & mtvec) : mtvec;
57                                         12'h342: mcause <= (csr_wsc_mode == 2'b01) ?
58                                             write_data :
59                                                 (write_data | mcause) :
60                                                     (~write_data & mcause) : mcause;
61                                                     12'h343: mtval <= (csr_wsc_mode == 2'b01) ?
62                                         write_data :
63                                             (write_data | mtval) :
64                                                 (~write_data & mtval) : mtval;
65                                                 12'h341: mepc <= (csr_wsc_mode == 2'b01) ? write_data :
66                                                     (csr_wsc_mode == 2'b10) ?
67                                         (write_data | mepc) :
68                                             (~write_data & mepc) : mepc;
69                                             endcase
70                                         end
71                                         else if(by_en == 1)begin
72                                             mstatus <= by_mstatus;
73                                             //mtvec <= by_mtvec;
74                                             mcause <= by_mcause;
75                                             mtval <= by_mtval;
76                                             mepc <= by_mepc;
77                                         end
78                                         end
79     end
80 endmodule

```

和之前 Regs 的实现一样，写部分使用时序完成，读部分使用组合逻辑，外部输入的写数据通过 `csr_wsc_mode` 来选择不同的模式与原值进行运算最后写回。除此以外还有一个 `by_en` 信号，用于中断时的寄存器修改。这样可以方便批量修改寄存器的值。

2.1.2 RV_INT

这个模块接受 SCPU_ctrl 产生的异常信号与硬件输入的 INT 中断信号，然后给出外界寄存器使能和要改变的 PC 值来改变 PC 流进入中断异常处理程序，我的实现是在这个模块的内部

实例化 CSRRegs 模块，用于修改特权级寄存器的值。实际上这一部分也可以接到整个 datapath 上。

```
1 module RV_INT(
2     input      clk,
3     input      rst,
4     input      INT,           // 外部中断信号
5     input      ecall,         // ECALL 指令
6     input      mret,          // MRET 指令
7     input      illegal_inst, // 非法指令信号
8     input      l_access_fault, // 数据访存不对
9     input      j_access_fault, // 跳转地址不对
10    input [31:0] inst_field, // 指令
11    input [31:0] pc_current, // 当前指令 PC
12    // CSRRegs 操作-----
13    input [11:0] read_addr, write_addr,      // 读写 CSR 寄存器的地址
14    input [31:0] write_data,                  // 写入 CSR 寄存器的数据
15    input csr_w,                          // 写使能
16    input [1:0] csr_wsc_mode,                // 写入 CSR 寄存器的模式
17    output[31:0] read_data,                 // 读出 CSR 寄存器的数据
18    // -----
19    output reg en,                      // 用于控制寄存器堆、内存等器件的写使能，同时改变
PC
20    output reg [31:0] pc,             // 将执行的指令 PC
21    output [31:0] mstatus,          // MSTATUS 寄存
22    output [31:0] mtvec,            // MTVEC 寄存
23    output [31:0] mcause,           // MCAUSE 寄存
24    output [31:0] mtval,            // MTVAL 寄存
25    output [31:0] mepc,             // MEPC 寄存
26    // output [31:0] csrout        // 读出 CSR 寄存器的
27 );
28 // 用于处理中断，中断的 pc?
29 // mret 要做的事情是 mstatus = 0, mcause =0, 输出 PC= mepc
30 // ecall : mepc = ecall 的 PC, mcause = 2'b11
31 // illegal_inst : mcause = 2'b10
32
33
34    reg [31:0] by_mstatus, by_mtvec, by_mcause, by_mtval, by_mepc;
35    // 关闭写使
36    // assign csrout = read_data;
37
38    CSRRegs CSRRegs(
39        .clk(clk),
40        .rst(rst),
41        .read_addr(read_addr),
42        .write_addr(write_addr),
43        .write_data(write_data),
44        .csr_w(csr_w),
45        .csr_wsc_mode(csr_wsc_mode),
46        .by_en(~en & ~mret),
47        .by_mstatus(by_mstatus),
48        .by_mtvec(by_mtvec),
49        .by_mcause(by_mcause),
50        .by_mtval(by_mtval),
51        .by_mepc(by_mepc),
52        .read_data(read_data),
53        .mstatus_o(mstatus),
54        .mtvec_o(mtvec),
55        .mcause_o(mcause),
56        .mtval_o(mtval),
57        .mepc_o(mepc)
```

```

58 );
59
60
61 // 保存中断异常的相关信号
62 always @ (*) begin
63     en <= ~(mret | INT | ecall | illegal_inst | l_access_fault |
64         j_access_fault);
65     if (mstatus == 1)begin // 如果正在中断, 那就不接受新的中断, 此时是
66         trap 处理
67             // 中断处理 等待 mret 信号 具体处理由软件实现即 trap 程序
68             if (mret == 1) begin
69                 by_mstatus <= 0;
70                 by_mcause <= 0;
71                 pc <= mepc;
72             end
73             // else pc <= pc_current;
74             else begin
75                 if (~en && mret == 0) begin // 这表示从正常发生中,然后将 pc 转到
76                     trap 处理程序
77                     by_mstatus <= 1;
78                     by_mcause <= (ecall == 1) ? 2'b11 : (illegal_inst ==
79                         1) ? 2'b10 : (INT == 1) ? 2'b01 : 2'b00;
80                     by_mtval <= inst_field;
81                     by_mepc <= pc_current;
82                     pc <= mtvec;
83             end
84         end
85     end
86 endmodule

```

以下是几个特权寄存器的作用:

- mstatus 存储当前控制状态, 这里简化为 1 是中断状态, 0 是正常状态
- mtvec 存储中断向量表基地址, 本次采用 direct 模式, 存储中断处理程序的基地址
- mcause 存储中断原因, 最高位 1 代表异常, 若为中断则为 0。Exception code 记录异常类型
简化为 01 代表硬件中断, 10 代表非法指令, 11 代表 ecall 指令 (异常) 00 代表正常执行
- mtval 存储非法指令内容
- mepc trap 触发时将要执行的指令地址, 在 mret 时作为返回地址。

2.1.3 中断处理程序 trap

这一部分由软件实现, 以下是我编写的 trap 汇编代码:

```

1 addi x19, x0, 40
2 addi x18, x0, trap
3 csrrw x0, 0x305, x18 # mtvec = trap
4
5 csr_test:
6     lui x28, 0x0EDCB
7     addi x28, x28, 0x666
8     csrrw x29, 0x300, x28 # m tstatus = 0x0EDCB666
9     lui x18, 0x1235
10    addi x18, x18, -1093 # x18 = 0x01234BBB
11    csrrs x30, 0x300, x18 # mtstatus = 0x0FFFFFFF

```

```

12    csrrwi x27, 0x300, x31 # mtstatus = 31, x27 = 0xFFFFFFFF
13    addi x27, x0, 0b10101
14    csrrc x29, 0x300, x27 # mtstatus = 0x0000000A, x29 = 0x31
15    csrrsi x30, 0x300, x16 # mtstatus = 0x0000001A, x30 = 0xC
16    csrrci x27, 0x300, x15 # mtstatus = 0x00000010, x27 = 0x1C
17    csrrwi x29, 0x300, x0 # mtstatus = 0x00000010, x29 = 0x10
18    csrrsi x28, 0x300, x0 # mtstatus = 0x00000010, x28 = 0x10
19    csrrci x27, 0x300, x0 # mtstatus = 0x00000010, x27 = 0x10
20
21 Exception_test:
22    nop # INT test #
23    ecall #
24    nop # illegal instruction
25    lui x31, 0x666
26
27 dummy:
28    nop
29    nop
30    nop
31    nop
32    nop
33    jal x0, dummy
34
35 trap:
36    addi x19, x19, -20
37    sw x20, 0(x19)
38    sw x21, 4(x19)
39    sw x22, 8(x19)
40    sw x23, 12(x19)
41    sw x24, 16(x19)
42
43    csrrwi x20, 0x300, x0 # mstatus
44    csrrwi x21, 0x305, x0 # mtvec
45    csrrwi x22, 0x341, x0 # mepc
46    csrrwi x23, 0x342, x0 # mcause
47    csrrwi x24, 0x343, x0 # mtval
48
49    srai x28, x23, 1
50    andi x28, x28, 1 # x28 = 1 异常, x28 = 0 中断
51    beq x28, x0, Interruption
52    jal x0, Exception
53    nop
54    nop
55    nop
56
57 Exception:
58    addi x29, x22, 4
59    csrrw x0, 0x341, x29 # mepc = mepc + 4
60    beq x0, x0, ret
61
62 Interruption:
63    nop
64    nop
65    beq x0, x0, ret
66
67 ret:
68    lw x24, 16(x19)
69    lw x23, 12(x19)
70    lw x22, 8(x19)
71    lw x21, 4(x19)
72    lw x20, 0(x19)
73    addi x19, x19, 20
74    nop #mret

```

前一部分是对特权寄存器的测试，后一部分异常处理程序。测试时使用的也是上面代码生成的指令。

2.2 仿真测试

2.2.1 测试代码

在 testbench 中添加了一个 INT 输入，用于表示外部中断信号。仿真代码与之前相同。

2.2.2 仿真波形

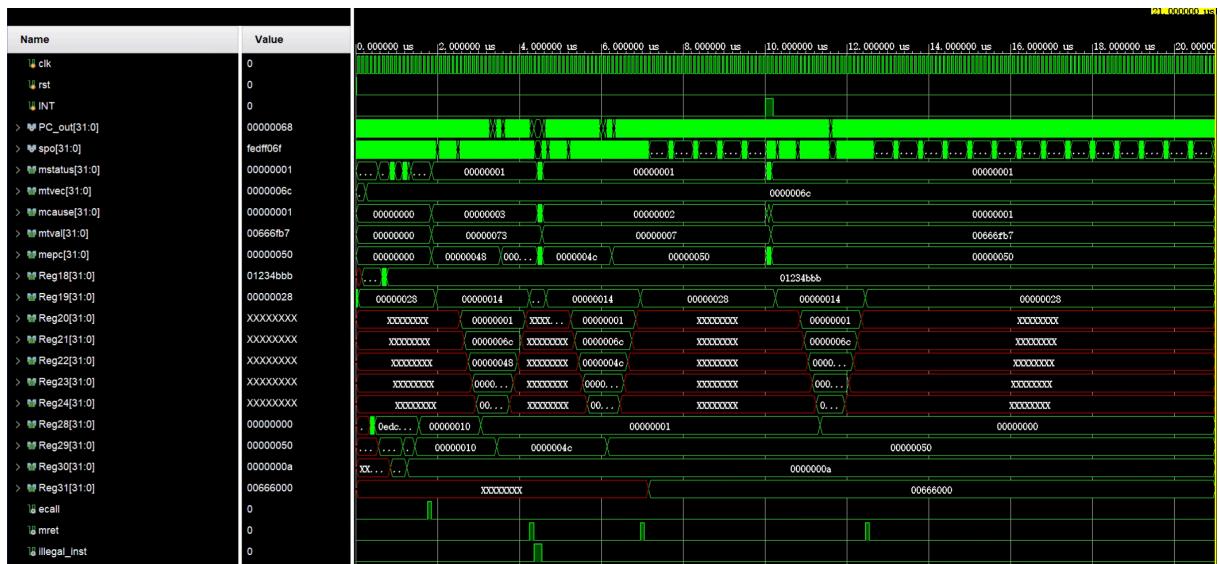


图 7：中断仿真

首先是全局的仿真波形，寄存器堆中的三个凹代表了三次中断异常的过程。第一次是 ecall 指令，第二次是非法指令，第三次是 INT 中断。最后进入了 dummy 死循环。

接下来分析前面的测试程序；

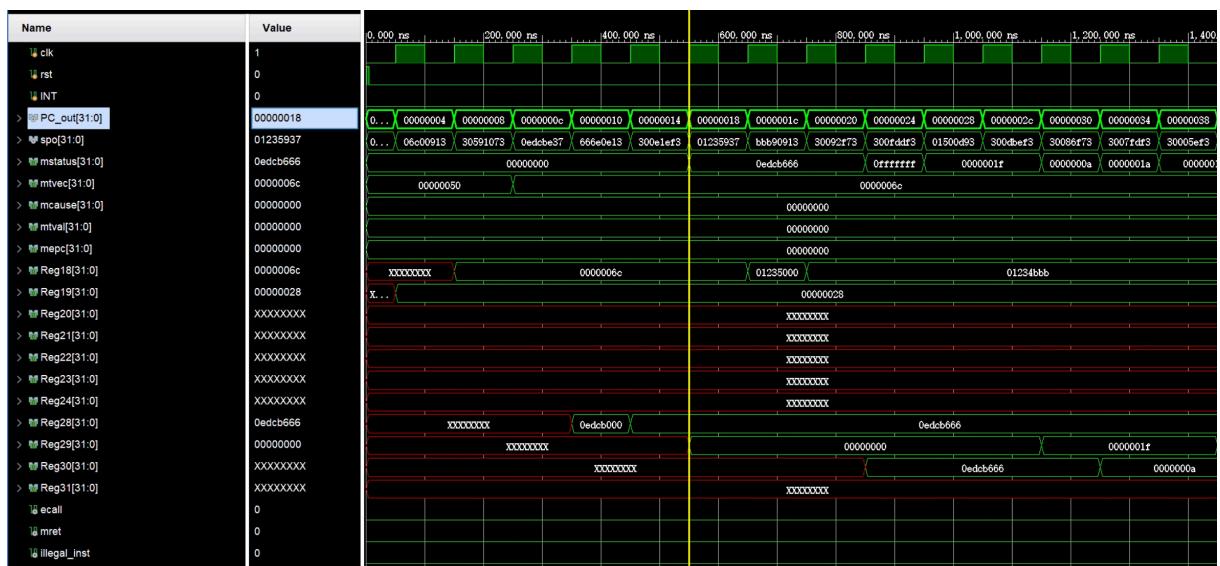


图 8：中断仿真

- 前两条指令是先写入内存指针，用于开辟栈空间，第二个是将 trap 处理程序的位置写入 mtvec 寄存器
- 接下来的几条指令都是对特权寄存器的测试，通过 csrrw,csrrs,csrrc,csrrwi,csrrsi,csrrci 指令对特权寄存器进行读写操作可以看到与预期一致（在汇编代码中写出预期结果）

下面分析异常处理程序：

如果接收到异常中断信号，就跳转到 mtvec 位置，然后开辟栈指针，将要使用的寄存器的值存入，然后将 csr 寄存器保存下来，然后判断异常类型，如果是异常就跳转到异常处理程序，如果是中断就直接返回。中断与异常处理的区别在于中断 mepc 不变，异常 mepc+4。即中断返回到中断发生的地方，异常返回到异常发生的下一条指令，跳过异常指令。

2.3 下板验证

```
RV32I Single Cycle CPU

pc: 00000048    inst: 00000073

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 01234BBB    s3: 00000028
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000010    t3: 00000010    t4: 00000010
t5: 0000000A    t6: 00666000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0      is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000000      cmp_res: 0

is_branch: 0    is_jal: 0      is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: F0000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000010    mcause: 00000000    mepc: 00000000    mtval: 00000000
mtvec: 0000006C    mie: 00000000    mip: 00000000
```

图 9：中断验证 1

此时是 ecall 指令，接下来要跳转到 trap 处理程序。其 pc 为 6C

```

RV32I Single Cycle CPU

pc: 00000006C    inst: FEC98993

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 01234BBB    s3: 00000028
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000010    t3: 00000010    t4: 00000010
t5: 0000000A    t6: 00666000

rs1: 13 rs1_val: 00000028
rs2: 0C rs2_val: 00000000
rd: 13 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0      is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000014    cmp_res: 0

is_branch: 0    is_jal: 0      is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 00000005

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000001    mcause: 00000003    mepc: 00000048    mtval: 00000073
mtvec: 0000006C    mie: 00000000    mip: 00000000

```

图 10: 中断验证 2

观察此时 CSR 寄存器的值，可以看到 mstatus = 1, mtvec = 6C, mcause = 3, mtval = 0, mepc = 48，随后继续程序，直到 mret 指令。

```

RV32I Single Cycle CPU

pc: 000000E4    inst: 30200073

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 01234BBB    s3: 00000028
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000010    t3: 00000001    t4: 0000004C
t5: 0000000A    t6: 00666000

rs1: 00 rs1_val: 00000000
rs2: 02 rs2_val: 00000000
rd: 00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0      is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0      is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: F0000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000001    mcause: 00000003    mepc: 0000004C    mtval: 00000073
mtvec: 0000006C    mie: 00000000    mip: 00000000

```

图 11: 中断验证 3

mret 指令，此时的 mepc 变为 4C，与预期相符合，到了原来异常指令的下一条。

```

RV32I Single Cycle CPU

pc: 00000004C    inst: 00000007

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 01234BBB    s3: 00000028
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000010    t3: 00000001    t4: 0000004C
t5: 0000000A    t6: 00666000

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0      is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0      is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: F0000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000001    mcause: 00000003    mepc: 0000004C    mtval: 00000073
mtvec: 0000006C    mie: 00000000    mip: 00000000

```

图 12: 中断验证 4

可以看到，返回后的 $pc = 4C$ ，说明程序正确，同时这也是一条非法指令，接下来再次进入 trap 处理程序。

```

RV32I Single Cycle CPU

pc: 00000006C    inst: FEC98993

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 01234BBB    s3: 00000028
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000010    t3: 00000001    t4: 0000004C
t5: 0000000A    t6: 00666000

rs1: 13 rs1_val: 00000028
rs2: 0C rs2_val: 00000000
rd: 13 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0      is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000014    cmp_res: 0

is_branch: 0    is_jal: 0      is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 00000005

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000001    mcause: 00000002    mepc: 0000004C    mtval: 00000007
mtvec: 0000006C    mie: 00000000    mip: 00000000

```

图 13: 中断验证 5

这次是非法指令， $mcause = 2$, $mtval = 00000007$, $mepc = 4C$, 接下来继续程序，直到 $mret$ 指令。

```

RV32I Single Cycle CPU

pc: 000000E4    inst: 30200073

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 01234BBB    s3: 00000028
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000   s11:00000010    t3: 00000001    t4: 00000050
t5: 0000000A    t6: 00666000

rs1: 00 rs1_val: 00000000
rs2: 02 rs2_val: 00000000
rd: 00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0     is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0     is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: F0000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000001    mcause: 00000002    mepc: 00000050    mtval: 00000007
mtvec: 0000006C    mie: 00000000    mip: 00000000

```

图 14: 中断验证 6

可以看到，此时的 mepc 变为 50，与预期相符合，到了原来异常指令的下一条。

接下来再进行一次硬件中断。输入前的 pc 为 50，最后 mret 时 pc 也为 50，说明中断处理程序正确。

```

RV32I Single Cycle CPU

pc: 000000E4    inst: 30200073

x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 01234BBB    s3: 00000028
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000   s11:00000010    t3: 00000000    t4: 00000050
t5: 0000000A    t6: 00666000

rs1: 00 rs1_val: 00000000
rs2: 02 rs2_val: 00000000
rd: 00 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0     is_lui: 0      imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0     is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: F0000000    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000001    mcause: 00000001    mepc: 00000050    mtval: 00666FB7
mtvec: 0000006C    mie: 00000000    mip: 00000000

```

图 15: 中断验证 7

综合以上，我们的 trap 程序是正确的。

3 思考题

不能得到，第一条指令执行后， $x6 = 0xDEADB000$ ，但是在 I-type 的立即数生成中使用的是有符号扩展，得到的立即数是 0xFFFFFEEF，所以最后的结果应该是， $x6 = 0xDEADAEEF$ 。

解决方式可以是给前面的立即数加 1 即第一条指令改为 `lui t0, 0xDEADC` 这样就抵消了后面位扩展带来的影响。

4 实验总结

在报告截止的前一天晚上调通了所有的程序，然后拖延症导致报告在截止前还没写完（悲）。但其实报告的内容也并没有很详尽，如果对每一条指令都进行详尽的分析会使得报告的篇幅巨长，所以只对关键的部分进行了分析，比如 4-3 的最后 Reg31 值应该是 666，4-4 的中断处理关键点的特权寄存器的值，这些我认为是最重要的部分。其实开始的时候是想过分析每一条指令的，但是写不动了（摊），写了一半又都注释掉了。一开始其实是按照 4-1, 4-2 的顺序完成的，但是后来在修改数据通路的时候发现给的数据通路并不能满足要求，而且要改的地方很多，所以就推翻重做了，直接画了个新的数据通路，然后增加了拓展指令的控制信号。

在做的过程中有很多波折，这次实验前前后后应该做了半个月，在五一之前花了两天把 4-3 完成，然后在五一结束以后两天把中断写完。两部分都遇到了仿真顺利但是下板爆炸的情况。其中第一次的问题是 RAM 不知道出了什么问题，重新生成一下下板就通过了（白白查了 3 个小时的信号）。第二次的问题是 CPU_ctrl 没有完善，在新增了中断的指令和 CSR 指令后，没有把全部的信号都约束上，这样就导致一些信号还是上一条指令的值或者是一些未知的值，这就导致了上板后的结果不对。最后这些问题也都得到了解决。另外在修改代码的过程中也思考了关于时序和组合的问题，很多时候也许是时序的问题导致了下板时候与仿真不同，因此修改后的时序与组合的逻辑相对而言更加清晰。

这次对硬件的 debug 真的是相当痛苦，真的很需要耐心，好在最后都顺利完成了。这次要使用掉一天的自由时间，希望下次流水线能够在 ddl 之前完成，不能再拖了！