实验 4-4：中断的相关设计

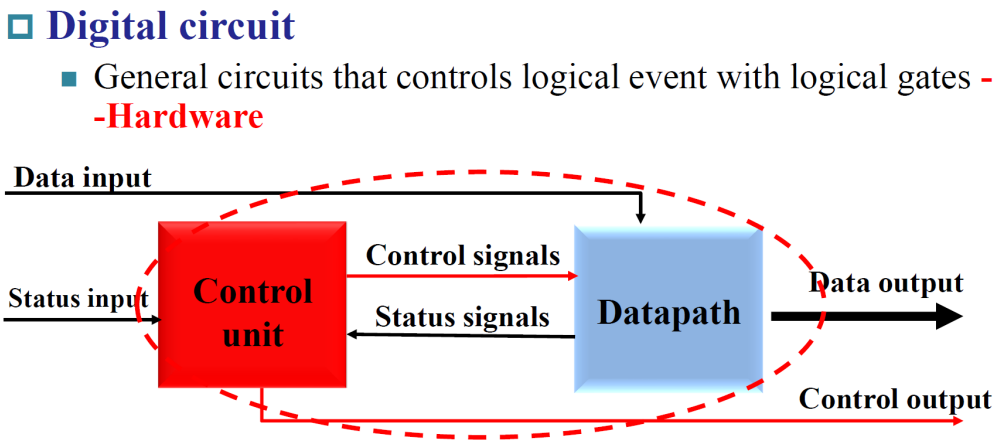
一、操作方法与实验步骤

目标：

熟悉RISC-V 中断的原理，了解引起CPU中断产生的原因及其处理方法，扩展包含中断的CPU

任务一：

扩展实验CPU中断功能;修改设计数据通路和控制器;修改或替换Exp04-3的数据通路及控制器;兼容Exp04-3数据通路增加中断通路;增加中断控制;扩展CPU中断功能;非法指令中断；外部中断；ecall。

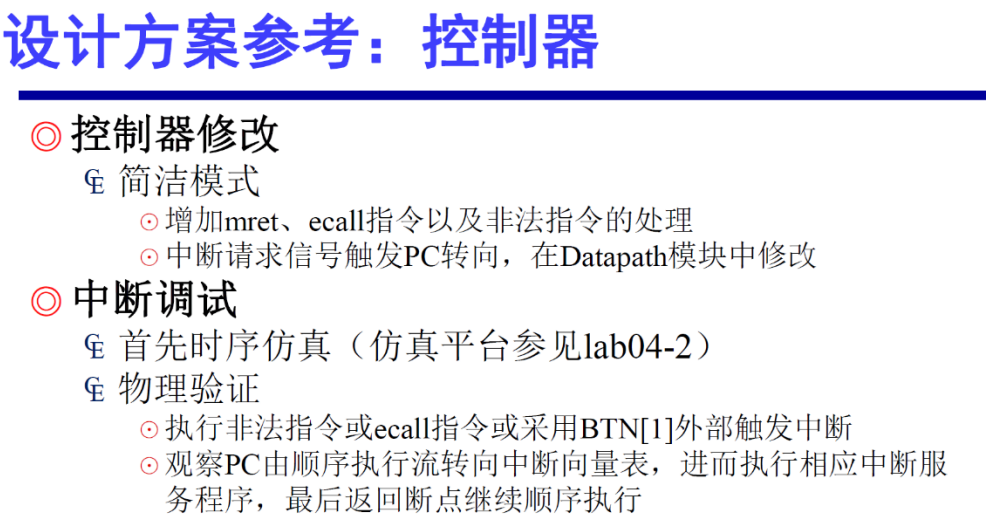


任务二：

设计CPU中断测试方案并完成测试

具体模块方案：

1. 控制器模块：




```
module SCPU_ctrl(
input[31:0]inst_field,
input[6:0]OPcode, //Opcode------inst[6:0]
input[2:0]Fun3, //Function-----inst[14:12]
input Fun7, //Function-----inst[30]
```

```
input MIO_ready, //CPU Wait
output reg [2:0]ImmSel, //立即数选择控制
output reg ALUSrc_B, //源操作数 2 选择
output reg [2:0] MemtoReg, //写回数据选择控制
output reg [1:0]Jump, //jal
output reg Branch, //beq
output reg RegWrite, //寄存器写使能
output reg MemRW, //存储器读写使能
output reg [31:0]ALU_Control, //alu 控制
output reg CPU_MIO, //not use
output reg mret,
output reg ill_instr,
output reg ecall,
output reg[1:0] choose,//用于 RV 写入数据的选择
output reg csr_wen
   );
   reg [1:0] ALUop;
always @* begin
case(OPcode)
7'b1110011:begin
case(Fun3)
3'b000:begin//ecall 或 mret
case(inst_field[31:20])
12'h000:
begin
RegWrite=0;
ImmSel=3'b100;//无关项
ALUSrc_B=1;//无关项
Branch=0;
Jump=0;
MemtoReg=0;//无关项
MemRW=0;//应该不写也不读此处写 0 也没啥
ALUop=2'b00;//无关项
mret=0;
ill_instr=0;
ecall=1;
csr_wen=0;
end
12'h302:
begin
RegWrite=0;//不要写
ImmSel=3'b100;//无关项
ALUSrc_B=1;//无关项
Branch=0;
Jump=0;
MemtoReg=0;//无关项
MemRW=0;//应该不写也不读此处写 0 也没啥
ALUop=2'b00;//无关项
mret=1;
ill_instr=0;
```

```
ecall=0;
csr_wen=0;
end
endcase
end
3'b001:begin//csrrw 把 csr 的值写入 x[rd]
//把寄存器 x[rs1]的值写入 csr
//MemToReg 等于 5 使得寄存器中的值可以写入 rd
//同时寄存器还需要一个输入值,rs1_data
RegWrite=1;
ImmSel=3'b100;//无关项
ALUSrc_B=0;//等于 0 使得 RV_input 的 I1 为 rs1_data
Branch=0;
Jump=0;
MemtoReg=3'b101;//MemToReg 扩展到 5，来自 csr 值写入 reg
MemRW=0;//应该不写也不读此处写 0 即可
ALUop=2'b00;//无关项
mret=0;
ill_instr=0;
ecall=0;
choose=2'b00;
csr_wen=1;
end

3'b101:begin//csrrwi 把 csr 的值写入 x[rd]
//把 imm 的值写入 csr
//MemToReg 等于 5 使得寄存器中的值可以写入 rd
//同时寄存器还需要一个输入值,imm_out,ImmSel=3'b101
RegWrite=1;
ImmSel=3'b101;
ALUSrc_B=1;
choose=2'b00;//等于 0 使得 RV 的 input 为 imm_out
Branch=0;
Jump=0;
MemtoReg=3'b101;//MemToReg 扩展到 5，来自 csr 值写入 reg
MemRW=0;//应该不写也不读此处写 0 即可
ALUop=2'b00;//无关项
mret=0;
ill_instr=0;
ecall=0;
csr_wen=1;
end

3'b011:begin//csrrc
RegWrite=1;
ImmSel=3'b101;//生成 csr 格式的立即数
ALUSrc_B=0;//等于 0 使得 RV_input 的 I1 为 rs1_data
Branch=0;
Jump=0;
MemtoReg=3'b101;//MemToReg 扩展到 5，来自 csr 值写入 reg
```

```
MemRW=0;//应该不写也不读此处写 0 即可
ALUop=2'b00;//无关项
choose=2'b01;//做与操作
mret=0;
ill_instr=0;
ecall=0;
csr_wen=1;
end

3'b111:begin//csrrci
RegWrite=1;
ImmSel=3'b101;//生成 csr 格式的立即数
ALUSrc_B=1;//等于 0 使得 RV_input 的 I1 为 imm_out
Branch=0;
Jump=0;
MemtoReg=3'b101;//MemToReg 扩展到 5，来自 csr 值写入 reg
MemRW=0;//应该不写也不读此处写 0 即可
ALUop=2'b00;//无关项
choose=2'b01;//做与操作
mret=0;
ill_instr=0;
ecall=0;
csr_wen=1;
end

3'b010:begin//csrrs
RegWrite=1;
ImmSel=3'b101;//生成 csr 格式的立即数
ALUSrc_B=0;//等于 0 使得 RV_input 的 I1 为 rs1_data
Branch=0;
Jump=0;
MemtoReg=3'b101;//MemToReg 扩展到 5，来自 csr 值写入 reg
MemRW=0;//应该不写也不读此处写 0 即可
ALUop=2'b00;//无关项
choose=2'b10;//做或操作
mret=0;
ill_instr=0;
ecall=0;
csr_wen=1;
end

3'b110:begin//csrrsi
RegWrite=1;
ImmSel=3'b101;//生成 csr 格式的立即数
ALUSrc_B=1;//等于 0 使得 RV_input 的 I1 为 imm_out
Branch=0;
Jump=0;
MemtoReg=3'b101;//MemToReg 扩展到 5，来自 csr 值写入 reg
MemRW=0;//应该不写也不读此处写 0 即可
ALUop=2'b00;//无关项
```

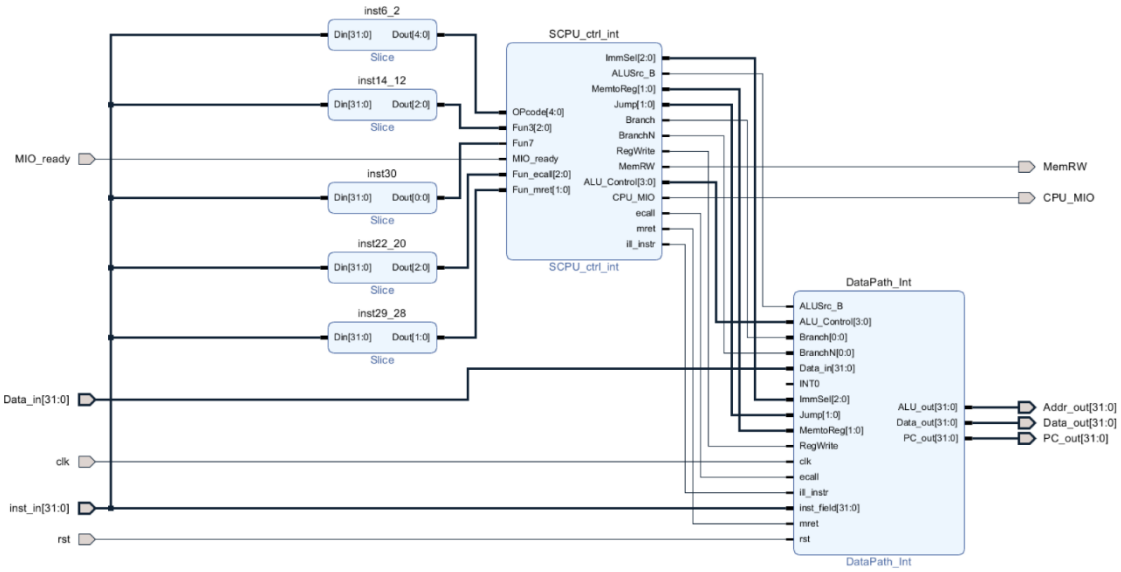
```
choose=2'b10;//做或操作
mret=0;
ill_instr=0;
csr_wen=1;
ecall=0;
end
endcase
end
7'b0110111: begin //lui
RegWrite=1;
ImmSel=3'b100;
ALUSrc_B=1;//无关项
Branch=0;
Jump=0;
MemtoReg=3'b011;
MemRW=0;//应该不写也不读此处写 0 也没啥
ALUop=2'b00;//无关项
mret=0;
ill_instr=0;
ecall=0;
csr_wen=0;
end
7'b0010111: begin //auipc
RegWrite=1;
ImmSel=3'b100;
ALUSrc_B=1;
Branch=0;
Jump=0;//pc+4
csr_wen=0;
MemtoReg=3'b100;
MemRW=0;
ALUop=2'b00;
mret=0;
ill_instr=0;
ecall=0;
end
7'b1100111: begin //jalr
csr_wen=0;
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=1;
Branch=0;
Jump=2'b10;
MemtoReg=3'b010;
MemRW=0;
ALUop=2'b00;
mret=0;
ill_instr=0;
ecall=0;
end
```

```
7'b0110011: begin //ALU R-type ok
csr_wen=0;
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=0;
Branch=0;
Jump=0;
MemtoReg=0;
MemRW=0;
ALUop=2'b10;
mret=0;
ill_instr=0;
ecall=0;
end
7'b0000011: begin//load I-type ok
csr_wen=0;
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=1;
Branch=0;
Jump=0;
MemtoReg=3'b001;
MemRW=0;
ALUop=2'b00;
mret=0;
ill_instr=0;
ecall=0;
//ALU_Control=add
end
7'b0100011: begin//store S-type ok
csr_wen=0;
RegWrite=0;
ImmSel=3'b001;
ALUSrc_B=1;
Branch=0;
Jump=0;
MemtoReg=0;
MemRW=1;
ALUop=2'b00;
mret=0;
ill_instr=0;
ecall=0;
end
7'b1100011: begin //beq B-type ok
csr_wen=0;
RegWrite=0;
ImmSel=3'b010;
ALUSrc_B=0;
Branch=1;
Jump=0;
```

```
MemtoReg=0;
MemRW=0;
ALUop=2'b01;
mret=0;
ill_instr=0;
ecall=0;
end
7'b1101111: begin //jump  J-type
csr_wen=0;
RegWrite=1;
ImmSel=3'b011;
ALUSrc_B=1;
Branch=0;
Jump=1;
MemtoReg=3'b010;
MemRW=0;
ALUop=2'b00;
mret=0;
ill_instr=0;
ecall=0;
end
7'b0010011:begin//ALU(addi;;;;) I-type
csr_wen=0;
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=1;
Branch=0;
Jump=0;
MemtoReg=3'b000;
MemRW=0;
ALUop=2'b11;
mret=0;
ill_instr=0;
ecall=0;
//ALU_Control=add
end
default:begin
csr_wen=0;
RegWrite=0;
ImmSel=3'b000;
ALUSrc_B=1;
Branch=0;
Jump=0;
MemtoReg=3'b000;
MemRW=0;
ALUop=2'b00;
mret=0;
ill_instr=1;
ecall=0;
end
```

```
endcase
end
assign Fun = {Fun3,Fun7};
always @* begin
case(ALUop)
2'b00:begin ALU_Control =  32'd0; end//add计算地址   lw,sw
2'b01:begin
case(Fun3)
3'b000:begin ALU_Control=32'd10;  end//beq
3'b001:begin ALU_Control=32'd11;  end//bne
3'b100:begin ALU_Control=32'd12;  end//blt
3'b101:begin ALU_Control=32'd13;  end//bge
3'b110:begin ALU_Control=32'd14;  end//bltu
3'b111:begin ALU_Control=32'd15;  end//bgeu
endcase//sub比较条件   beq
end
2'b10:begin  //实现了第四行的全部指令
case({Fun3,Fun7}) //R-formats
4'b0000:begin ALU_Control = 32'd0 ; end//add
4'b0001:begin ALU_Control = 32'd1 ; end//sub
4'b0010:begin ALU_Control = 32'd2 ; end//sll
4'b0100:begin ALU_Control = 32'd3 ; end//slt
4'b0110:begin ALU_Control = 32'd4 ; end//sltu
4'b1000:begin ALU_Control = 32'd5 ; end//xor
4'b1010:begin ALU_Control = 32'd6 ; end//srl
4'b1011:begin ALU_Control = 32'd7 ; end//sra
4'b1100:begin ALU_Control = 32'd8 ; end//or
4'b1110:begin ALU_Control = 32'd9 ; end//and
default:begin ALU_Control=32'bx ; end
endcase
end
2'b11:begin
case(Fun3) //I-format 实现了第三行的全部指令
3'b000:begin ALU_Control = 32'd0;end//addi
3'b010:begin ALU_Control = 32'd3;end//slti
3'b011:begin ALU_Control = 32'd4;end//sltiu
3'b100:begin ALU_Control = 32'd5;end//xori
3'b110:begin ALU_Control = 32'd8;end//ori
3'b111:begin ALU_Control = 32'd9;end//andi
3'b101:begin ALU_Control = Fun7==0?32'd6:32'd7;end //srli srai
//case(Fun7)
//1'b0:begin ALU_Control = 32'd6;end//srli
//1'b1:begin ALU_Control = 32'd7;end//srai
//endcase
3'b001:begin ALU_Control = 32'd2;end//slli
endcase
end
```

增加中断后的CPU模块



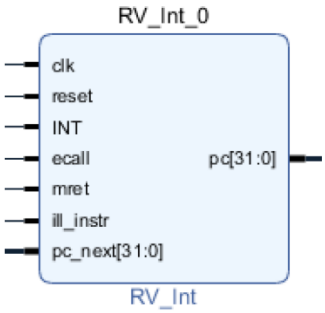
2．数据通路模块

设计方案参考：DataPath

◎ **DataPath修改**

- CPU复位时，MEPC=PC=0x00000000
- 修改PC模块增加
  - ⊙mtvec寄存器型变量，中断和异常触发PC转向中断地址
    - ◆ 相当于硬件触发Jal，用mret返回
  - ⊙mepc寄存器，中断和异常返回PC的地址
  - ⊙增加控制信号INT、mret、ecall、ill_instr
    - ◆ INT宽度根据扩展的外中断数量设定

注意：INT是电平信号，不要重复响应



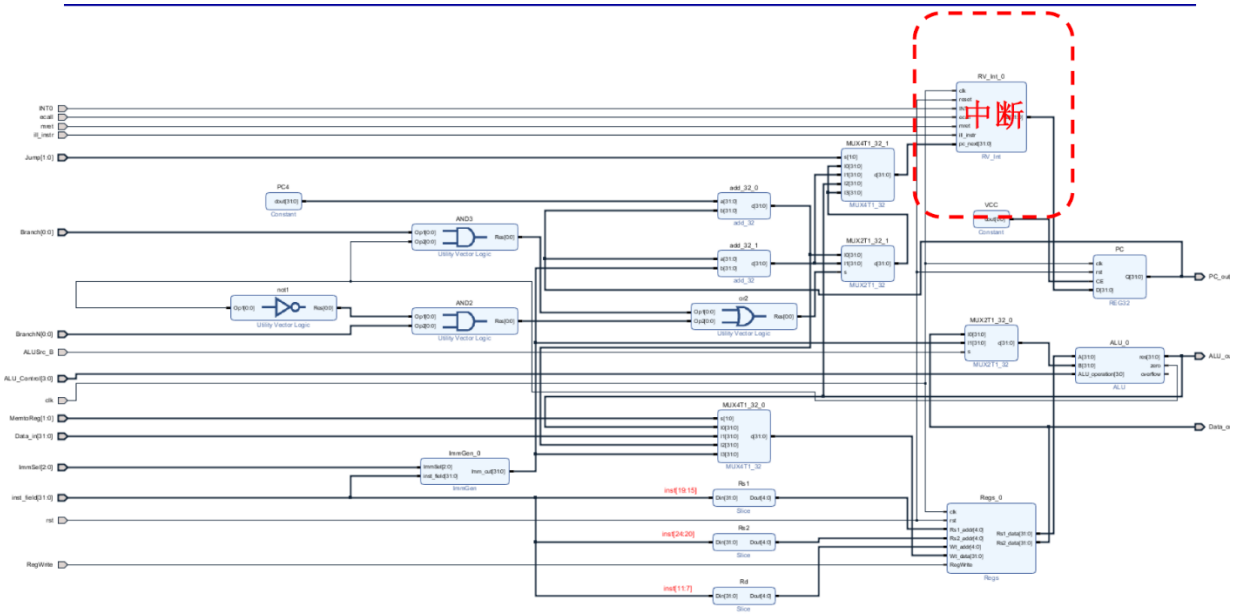

```
`include "Defines.vh"
module DataPath(
`VGA_DBG_Core_Outputs//加入 vga 中的 32 个寄存器和其他的指令数据的 vga 输出显示
input wire clk,
input wire rst,
input wire tim_int,
input wire ext_int,
input wire mret,
```

```
input wire ill_instr,
input wire ecall,
input wire csr_wen,
input wire [31:0] inst_field,
input wire [31:0]Data_in,
input wire [31:0] ALU_Control,
input wire [2:0] ImmSel,
input wire [2:0]MemtoReg,
input wire ALUSrc_B,
input wire [1:0] csr_ctrl,
input wire [1:0]Jump,
input wire Branch,
input wire RegWrite,
output wire [31:0]PC_out,
output wire [31:0] Data_out,
output wire [31:0] ALU_out
    );
    wire [31:0] Regs_0Rs1_data;
    wire [31:0] MUX2T1_32_0_o;
    wire ALU_0_zero;
    wire and2_Res;
    wire [31:0] PC_4;
    wire [31:0] PC_ano;
    wire [31:0]Imm_out;
    wire [31:0]PC2Choose;//第一次选出来的 pc，用来进入第二次以 Jump 为依据的选择中
    wire [31:0]newPC;//送到 RV_Int 中
    wire [31:0]lastPC;//送到 PC 中
    wire [31:0]MUX6T1_output;
    wire [31:0]CSR_out;
    wire [11:0]csr_ind;
    wire [1:0]whichEx;
    ImmGen IMM(
    .ImmSel(ImmSel),
    .inst_field(inst_field),
    .Imm_out(Imm_out));

    ALU ALU_U(//补上
    .a_val(Regs_0Rs1_data),
    .b_val(MUX2T1_32_0_o),
    .ctrl(ALU_Control),
    .result(ALU_out),
    .zero(ALU_0_zero));

    and_2 SB2(
    .Op1(Branch),
    .Op2(ALU_0_zero),
    .Res(and2_Res));//补线

    add32 add_32_0(
    .a(PC_out),
```

```
.b(4),
.c(PC_4));

 add32 add_32_1(
.a(PC_out),
.b(Imm_out),
.c(PC_ano));


MUX2T1 MUX2T1_32_1(
.s(and2_Res),
.I0(PC_4),
.I1(PC_ano),
.o(PC2Choose));


MUX4T1 MUX4T1_32_1(
.s(Jump),
.I0(PC2Choose),
.I1(PC_ano),
.I2(ALU_out),
.I3(PC2Choose),
.o(newPC));


MUX6T1 MUX6T1_32_0(
.s(MemtoReg),
.I0(ALU_out),
.I1(Data_in),
.I2(PC_4),
.I3(Imm_out),
.I4(PC_ano),
.I5(CSR_out),
.o(MUX6T1_output));

RegFile Regs(
`VGA_DBG_RegFile_Arguments//寄存器赋初值
.clk(clk),
.rst(rst),
.wen(RegWrite),
.i_data(MUX6T1_output),//补上
.rs1(inst_field[19:15]),
.rs2(inst_field[24:20]),
.rd(inst_field[11:7]),
.rs1_val(Regs_0Rs1_data),
.rs2_val(Data_out)
);

MUX2T1 MUX2T1_32_0(
.s(ALUSrc_B),
```

```
.I0(Data_out),
.I1(Imm_out),
.o(MUX2T1_32_0_o));

wire [31:0]MUX2T1_32_2_o;
MUX2T1 RV_data(
.s(ALUSrc_B),
.I0(Regs_0Rs1_data),
.I1(Imm_out),
.o(MUX2T1_32_2_o));


wire [31:0] csr_input;

 Exception EX(
.inst_field(inst_field),
.lastPC(lastPC),
.addr(ALU_out),
.whichEx(whichEx)
);

RV_input RV_input_0(
.s(csr_ctrl),
.I0(CSR_out),
.I1(MUX2T1_32_2_o),
.o(csr_input));

RV_Int  RV(
    `VGA_DBG_Csr_Arguments
    .clk(clk),
    .whichEx(whichEx),
    .inst_field(inst_field),
    .data_in(csr_input),
    .reset(rst),
    .mem_add(ALU_out),
    .tim_int(tim_int),
    .ext_int(ext_int),
    .ecall(ecall),
    .ill_instr(ill_instr),
    .mret(mret),
    .pc_next(newPC),
    .pc(lastPC),
    .CSR_out(CSR_out)
);

REG32 PC(
.clk(clk),
.rst(rst),
.CE(1),
.D(lastPC),
```

```
//    .D(newPC_after),
    .Q(PC_out));

   assign dbg_pc=PC_out;
   assign dbg_inst=inst_field;
   assign dbg_rs1=inst_field[19:15];
   assign dbg_rs2=inst_field[24:20];
   assign dbg_rs1_val=Regs_0Rs1_data;
   assign dbg_rs2_val=Data_out;
   assign dbg_rd=inst_field[11:7];
   assign dbg_reg_i_data=MUX6T1_output;
   assign dbg_reg_wen=RegWrite;
   assign dbg_is_imm=(inst_field[6:0]!=0110011)?1:0;//非R型指令
   assign dbg_is_auipc=(inst_field[6:0]==7'b0010111)?1:0;
   assign dbg_is_lui=(inst_field[6:0]==7'b0110111)?1:0;
   assign dbg_imm=Imm_out;
   assign dbg_a_val=Regs_0Rs1_data;
   assign dbg_b_val=MUX2T1_32_0_o;
   assign dbg_alu_ctrl=ALU_Control;
   assign dbg_alu_res=ALU_out;
   assign dbg_cmp_ctrl=0;
   assign dbg_cmp_res=ALU_0_zero;
   assign dbg_is_branch=Branch;
   assign dbg_is_jal=(Jump==1)?1:0;
   assign dbg_is_jalr=(Jump==2'b10)?1:0;
   assign dbg_do_branch=and2_Res;
   assign dbg_pc_branch=lastPC;
   assign dbg_mem_wen=(inst_field[6:0]==7'b0100011)?1:0;//sw
   assign dbg_mem_ren=(inst_field[6:0]==7'b0000011)?1:0;
   assign dbg_dmem_o_data=Data_in;
   assign dbg_dmem_i_data=Data_out;
   assign
dbg_dmem_addr=(inst_field[6:0]==7'b0100011)|(7'b0000011)?ALU_out:0;

    assign dbg_csr_wen=csr_wen;//csr_wen;
    assign dbg_csr_ctrl=csr_ctrl;
    assign dbg_csr_ind=lastPC;
    assign dbg_csr_r_data=CSR_out;
endmodule
```



3. top 模块

```
`include "Defines.vh"
module Top(
    input clk_100mhz,
    input rstn,
    input [15:0] sw_in,
    input [4:0] key_col,
    output [4:0] key_row,
    output hs,
    output vs,
    output [3:0] vga_r,
    output [3:0] vga_g,
    output [3:0] vga_b,
    output [7:0] LED_o
);

    wire rst;
    wire [15:0] sw;
    wire [31:0] clk_div;
    wire [4:0] key_x;
    wire [4:0] key_y;
    wire dr1_wen;
    wire [31:0] dr1_o_data;
    wire [31:0] dr1_i_data;
    wire [31:0] imem_addr;
    wire [31:0] imem_o_data;
    wire [31:0] dmem_addr;
    wire [31:0] dmem_o_data;
    wire [31:0] dmem_i_data;
    wire dmem_wen;

    `VGA_DBG_Core_Declaration
    `VGA_DBG_RegFile_Declaration
    `VGA_DBG_Csr_Declaration
```

```
wire [31:0] imem_addr_new;
wire [31:0] imem_o_data_new;
wire dmem_wen_new;
wire [31:0]dmem_addr_new;
wire [31:0]dmem_i_data_new;
wire [31:0]dmem_o_data_new;
ClockDividor clock_dividor(
    .clk(clk_100mhz),
    .rst(rst),
    .step_en(sw[0]),
    .clk_step(key_x[0]),
    .clk_div(clk_div),
    .clk_cpu(clk_cpu)
);

InputAntiJitter inputter(
    .clk(clk_100mhz),
    .rstn(rstn),
    .key_col(key_col),
    .sw_in(sw_in),
    .rst(rst),//output
    .key_row(key_row),
    .key_x(key_x),
    .key_y(key_y),
    .sw(sw)
);


SCPU scpu(
    `VGA_DBG_Core_Arguments
    .clk(clk_cpu),
    .rst(rst),
    .tim_int(clk_div[31]),
    .ext_int(key_x[1]),
    .MIO_ready(),
    .PC_out(imem_addr),
    .inst_in(imem_o_data_new),////input
    .Addr_out(dmem_addr),
    .Data_in(dmem_o_data_new),
    .Data_out(dmem_i_data),
    .MemRW(dmem_wen),
    .CPU_MIO()
);
MACCtrl memacc(
    //facing core
    .i_iaddr(imem_addr),/////
    .o_idata(imem_o_data_new),//////
    .i_dwen(dmem_wen),//////
    .i_daddr(dmem_addr),//////
    .i_d_idata(dmem_i_data),//////
```

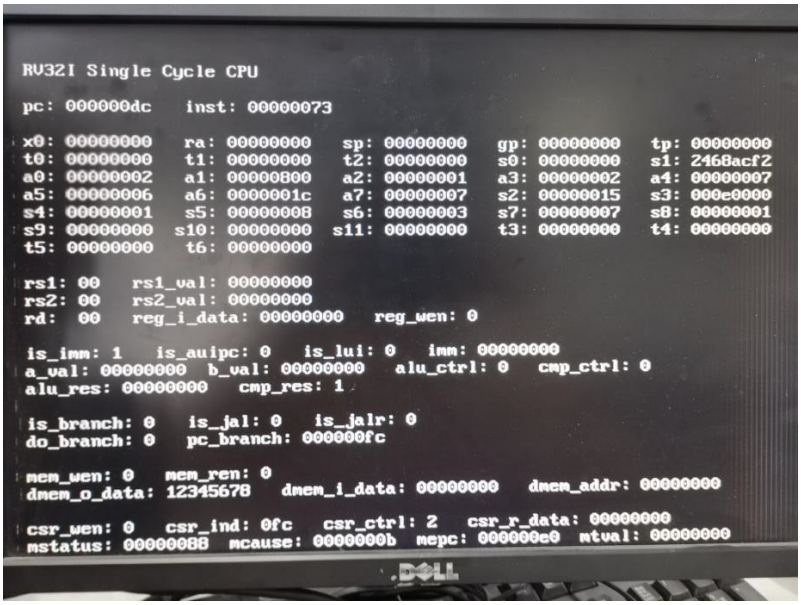
```
        .o_d_odata(dmem_o_data_new),//////
        //facing IMem
        .o_iaddr(imem_addr_new),/////
        .i_idata(imem_o_data),//////
        .o_dwen(dmem_wen_new),//////
        .o_daddr(dmem_addr_new),//////
        .o_d_idata(dmem_i_data_new),//////
        .i_d_odata(dmem_o_data),//////
        .o_dr1wen(dr1_wen),//////
        .o_dr1_idata(dr1_i_data),
        .i_dr1_odata(dr1_o_data)
    );

        DMem d_mem(
        .clk(~clk_cpu),
        .wen(dmem_wen_new),
        .addr(dmem_addr_new),
        .i_data(dmem_i_data_new),
        .o_data(dmem_o_data));


    dist_mem_gen_0 Imm(
    .a(imem_addr[11:2]),
    .spo(imem_o_data));

    VGA vga(
        `VGA_DBG_VgaDebugger_Arguments
        .rst(rst),
        .clk_div(clk_div),
        .hs(hs),
        .vs(vs),
        .vga_r(vga_r),
        .vga_g(vga_g),
        .vga_b(vga_b)
    );


    LEDCtrl dr1(
    .clk(~clk_cpu),
    .wen(dr1_wen),
    .i_data(dr1_i_data),
    .o_data(dr1_o_data),
    .o_led_ctrl(LED_o)
    );
Endmodule
```

二、实验结果验证

1. ecall 指令测试

```
test_ecall: # PC = 0xd0
    addi    a0, zero, 0       d0
    addi    a0, a0, 1  d4
    addi    a0, a0, 1  d8
    ecall              dc
    addi    a0, zero, 0       e0
```

当 pc=dc 时，ecall 进入异常。

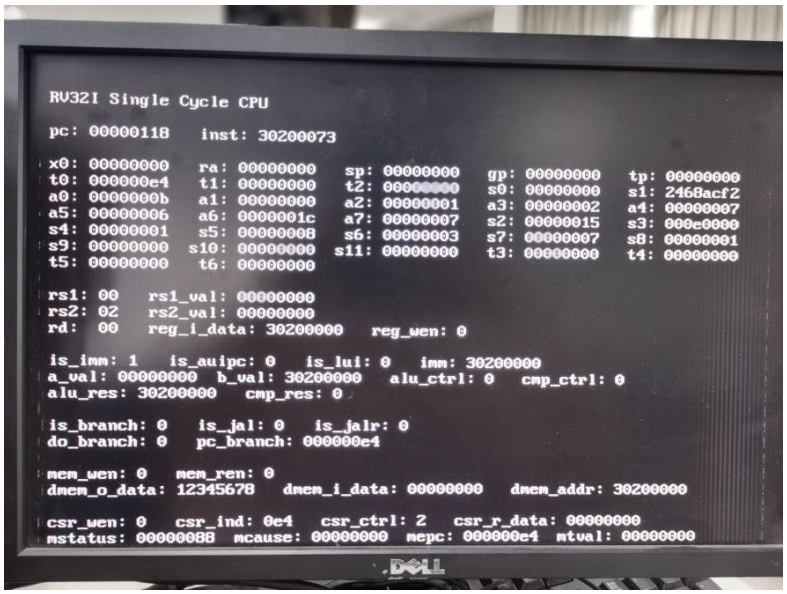


mcause=32’h0000000b,表示异常原因为 ecall,pc_branch 为寄存器 mtvec 中的值,, ecall 功能正常。

2. mret 指令测试

```
no_add_4:
  mret                118
```

当 pc=118 时，mret 跳出异常。

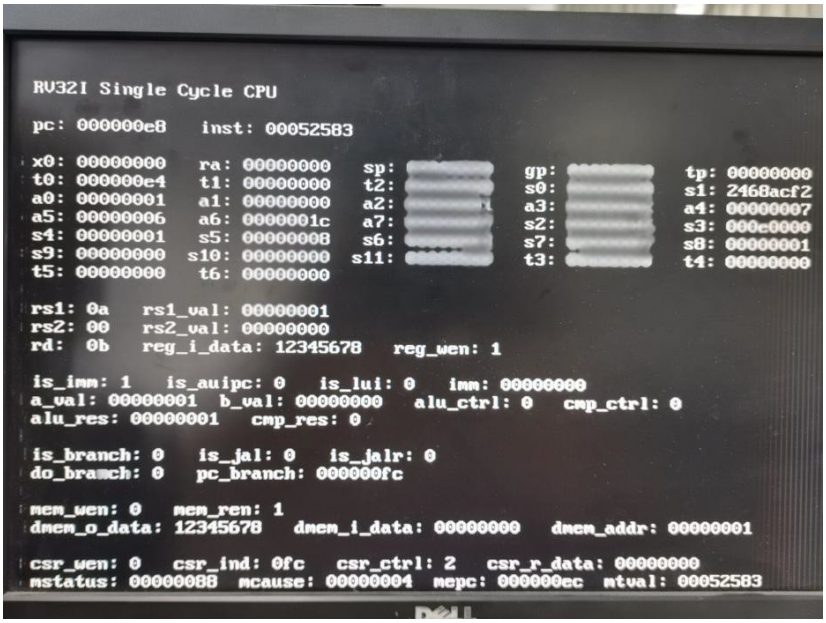


mcause 变为 0，pc_branch 为寄存器 mepc 中的值,mret 功能正常。

3. dMem 地址不以 4 对齐异常测试

```
test_load_misalign: # PC = 0xe4
  addi   a0, zero, 1        e4
  lw     a1, 0(a0)   e8                  //此处load地址有误，进入异常
  addi   a0, zero, 0 # PC = ec
```

当 pc=e8 时，lw a1, 0(a0)，再次进入异常。



mcause 变为 32’h00000004，pc_branch 为寄存器 mtvec 中的值，正常。

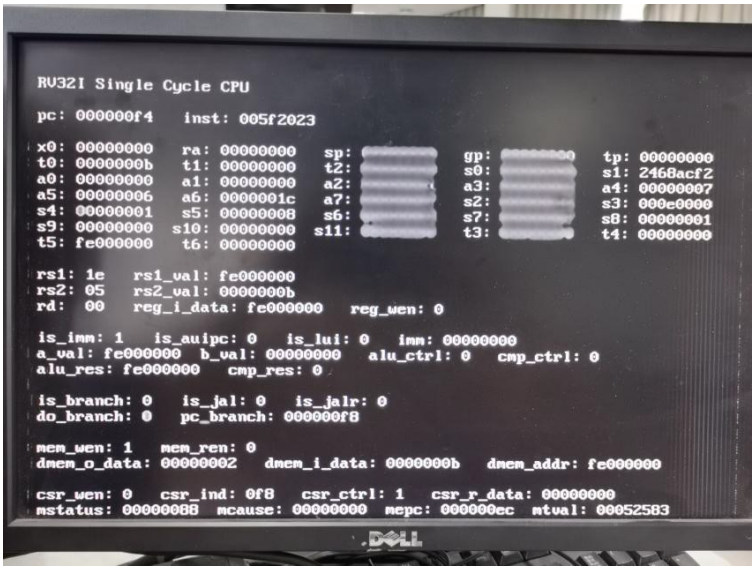
其他的异常、中断与这两种殊途同归，都是做的差不多的事。

4. J 型指令、U 型指令及 LED 灯测试

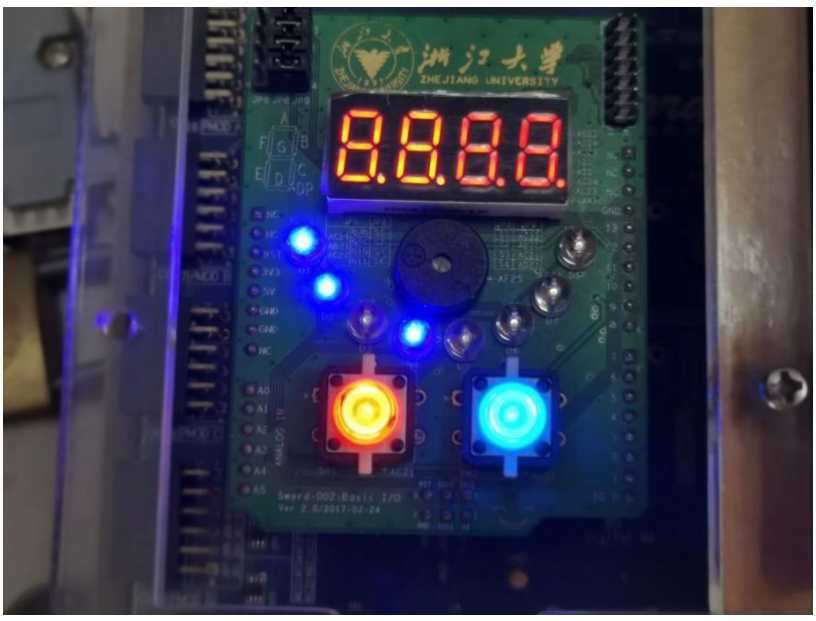
```
loop:   # PC = 0xf0
  lui x30 FE000          f0
  lw x5 0(x30)           f4
  jal    zero, loop      f8
```

通过f0 的lui 指令向 x30 中赋值 LED 的写入地址，然后在 f4 向对应地址存入x5 的值，以此获得 LED 灯的显示，再由 f8 的jal 指令跳转回来不断循环。

f4：

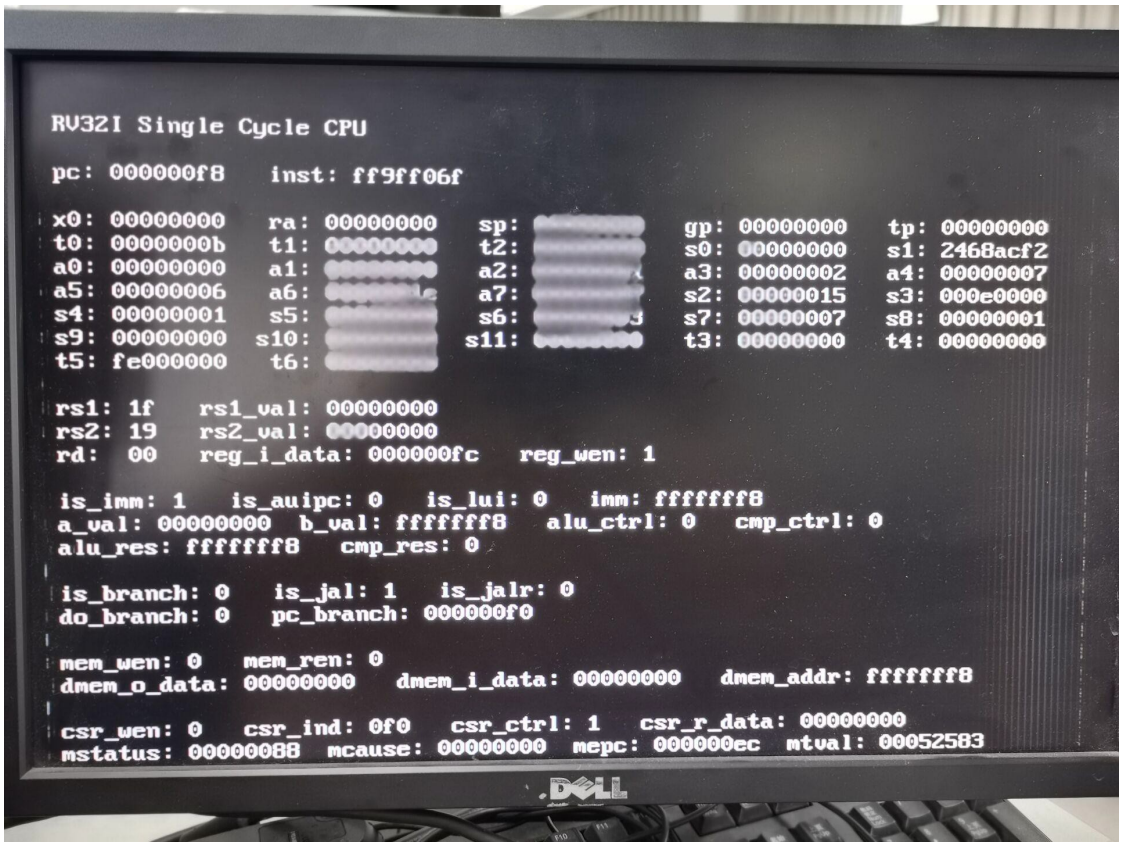


可以看到经过f0 的lui 指令之后，寄存器t5 的值变为了16’hfe000000f4 的dmem_addr 为 fe000000,写入数据为 b。



可以看到此时的 led 灯从右向左看为 2’b00001011，也即 8’h0b，显示正确。

f8：



is_jal=1,pc_branch=00000fc，正确。

三、实验心得

在这次实验中，我感觉最重要的是理解中断的概念，另外的就是利用网站，设计一个可以验证正确性的汇编代码，在这次的过程中，我曾经不理解 INTA,INT,INTR,mret 等信号的关系，主要是在计组的理论课堂上，这一块内容其实没有设计 CPU 的数据通路控制器这些内容讲的多，所以我在网上自学了很久。最后设计指令的时候，在和同学交流之后，对另一位同学的指令进行了修改，最终得到了一个自己设计的验证中断跳转的指令。可以说，这次实验对我的理论知识和理解硬件结构都是有很大帮助的。