

浙江大学

本科实验报告

课程名称: 计算机组成与设计

姓名: 张晋恺

学院: 竺可桢学院

系: 所在系

专业: 计算机科学与技术

学号: 3230102400

指导教师: 刘海风

2024年11月27日

浙江大学实验报告

课程名称: 计算机组成与设计 实验类型: 综合

实验项目名称: lab4-4

学生姓名: 张晋恺 专业: 计算机科学与技术 学号: 3230102400

同组学生姓名: 指导老师: 刘海风

实验地点: 东 4-512 实验日期: 2024 年 11 月 22 日

一、操作方法与实验步骤

本次实验主要要求我们在 lab4-3 的基础上, 为我们的 RISC-V 处理器添加中断处理功能。具体来说, 我们需要做到:

- 实现 Machine Mode 下的的常用寄存器的管理, 主要包括
 - mepc 寄存器: 存储异常返回地址
 - mcause 寄存器: 存储异常原因, 我设计的异常代码为
 - * 0: 无异常
 - * 1: 外部中断
 - * 2: ecall 指令
 - * 3: 非法指令
 - * 4: 数据访存不对齐
 - * 5: 跳转地址不对齐
 - mtvec 寄存器: 存储中断处理入口地址

- mstatus 寄存器：存储处理器状态
- mtval 寄存器：存储异常附加信息
- 实现常见的特权级指令，包括 csrrw, csrrs, csrrc, csrrwi, csrrsi, csrrci
- 实现常见的中断处理指令，包括 ecall, mret
- 实现中断处理程序，并支持在中断发生时，能够正确的跳转到中断处理程序，执行完后返回到原来的程序，需要支持的中断有
 - IO 设备的外部中断
 - 非法指令中断
 - ecall 指令

CSR-Register

首先，我们实现 CSR 寄存器堆，代码如下

```

1 module CSRRegs(
2     input clk, rst,
3     input[11:0] raddr, waddr,           // 读、写 CSR 寄存器的地址
4     input[31:0] wdata,                 // 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     output[31:0] rdata,                // 读出 CSR 寄存器的数据
8     output wire [31:0] mstatus,
9     output wire [31:0] mtvec,
10    output wire [31:0] mcause,
11    output wire [31:0] mtval,
12    output wire [31:0] mepc,
13
14    input expt_int,                      // 是否有异常中断
15    // 旁路输入
16    input [31:0] mepc_bypass_in,
17    input [31:0] mcause_bypass_in,
18    input [31:0] mtval_bypass_in,
19    input [31:0] mstatus_bypass_in,
20    input [31:0] mtvec_bypass_in
21
22 );
23

```

```

24 reg[31:0] csrs[1:5]; // CSR 寄存器
25
26 assign mstatus = csrs[1]; //h300
27 assign mtvec = csrs[2]; //h305
28 assign mcause = csrs[3]; //h342
29 assign mtval = csrs[4]; //h343
30 assign mepc = csrs[5]; //h341
31
32 reg[31:0] readcsr;
33
34 assign rdata = readcsr;
35
36 initial begin
37     csrs[1] <= 32'h0;
38     csrs[2] <= 32'h320; //trap 地址
39     csrs[3] <= 32'h0;
40     csrs[4] <= 32'h0;
41     csrs[5] <= 32'h0;
42 end
43
44 always @ (*) begin
45     case(raddr)
46         12'h300: readcsr = csrs[1];
47         12'h305: readcsr = csrs[2];
48         12'h342: readcsr = csrs[3];
49         12'h343: readcsr = csrs[4];
50         12'h341: readcsr = csrs[5];
51         default: readcsr = 32'h0;
52     endcase
53 end
54
55
56
57 always @(posedge clk or posedge rst) begin
58     if (rst) begin
59         csrs[1] <= 32'h0;
60         csrs[2] <= 32'h320; //trap 地址
61         csrs[3] <= 32'h0;
62         csrs[4] <= 32'h0;
63         csrs[5] <= 32'h0;
64     end
65     else begin
66         if (csr_w) begin
67             case(waddr)
68                 12'h300: csrs[1] <= wdata;

```

```

69         12'h305: csrs[2] <= wdata;
70         12'h342: csrs[3] <= wdata;
71         12'h343: csrs[4] <= wdata;
72         12'h341: csrs[5] <= wdata;
73         default: ;
74     endcase
75 end
76
77     if (expt_int) begin
78         csrs[5] <= mepc_bypass_in;
79         csrs[3] <= mcause_bypass_in;
80         csrs[4] <= mtval_bypass_in;
81         csrs[1] <= mstatus_bypass_in;
82         csrs[2] <= mtvec_bypass_in;
83     end
84 end
85
86 end

```

与实验文档的做法稍有不同，对于 CSR 指令等，我使用 `csr_w` 作为写使能信号，对于中断处理时，我们需要批量修改 CSR 寄存器，因此我使用 `expt_int` 作为中断写使能信号，并且增加五个寄存器的旁路输入，当时钟上升沿到来时，判断是哪一种写；在这个过程中，我始终保证了 `csr_w` 和 `expt_int` 不会同时为 1，因此不会出现冲突。为了方便机器码的转换，对于寄存器的编号，我与 RISC-V 的规范保持一致，即 `mepc=0x341`, `mcause=0x342`, `mtvec=0x305`, `mstatus=0x300`, `mtval=0x343`，根据输入的不同，选择不同的寄存器进行写操作。最后，在 `reset` 的时候，由于不需要完全按照 RISC-V 的规范，我只实现了最简单的中断；对于 `mstatus` 的 MIE 等位置，我并没有考虑，我只保证在中断处理时 `mstatus` 为 0 代表可以中断，为 1 代表中断进行中，不再接受新的中断。

所以 `reset` 的时候，我只将除 `mtvec` 外的寄存器清零，`mtvec` 为 `0x00000320`，即我的中断处理程序的入口地址。

实现了寄存器堆后，我们要为为了中断增添新的控制信号，具体的信号有

1. CSRTYPE: 用于选择 CSR 指令的类型，具体的选择如下:

- 000: don't care, 代表不是 CSR 指令，可能是 `ecall` 和 `mret`，也可能是其他指令
- 001: `csrrw`
- 010: `csrrs`

- 011: csrrc
- 101: csrrwi
- 110: csrrsi
- 111: csrrci

这一信号根据其指令的 Fun3 字段来选择

2. MRET: 用于选择是否执行 mret 指令
3. ECALL: 用于选择是否执行 ecall 指令
4. illegal: 用于判断是否执行非法指令，如果是非法指令则置 1，反之置 0

对 MainCtrl 更改的部分如下

```

1  ...
2  input [2:0]      Fun_ecall, //inst[22:20]
3  input [1:0]      Fun_mret, //inst[29:28]
4  ...
5  output reg [2:0] CSRTYPE, //000 don't care// 001 csrrw// 010 csrrs// 011
   ↪  csrrc// 101 csrrwi// 110 csrrsi// 111 csrrci
6  output reg      MRET, //1 enable, 0 disable
7  output reg      ECALL, //1 enable, 0 disable\
8  output reg      illegal//not supported instruction illegal=1
9  ...
10
11 //Case 语句增加分支
12 `OPCODE_ENV: begin // ecall, ebreak
13 ImmSel = `IMM_SEL_I;
14 ALUSrc_B = 0;
15 MemtoReg = (Fun3==3'b000)?3'b000:`MEM2REG_CSR;
16 Jump = 2'b00;
17 Branch = 4'b0000;
18 RegWrite = (Fun3[0]|Fun3[1]|Fun3[2]);
19 MemRW = 0;
20 ALU_op = 2'b00;
21 CPU_MIO = 0;
22 signal = 0;
23 width = 2'b00;
24 //
25 CSRTYPE = Fun3; //000 ecall or mret// 001 csrrw// 010 csrrs// 011 csrrc//
   ↪  101 csrrwi// 110 csrrsi// 111 csrrci
26 if(Fun3 = 3'b000) begin
27     MRET = ((Fun_mret = 2'b11)&&(Fun_ecall = 3'b010));

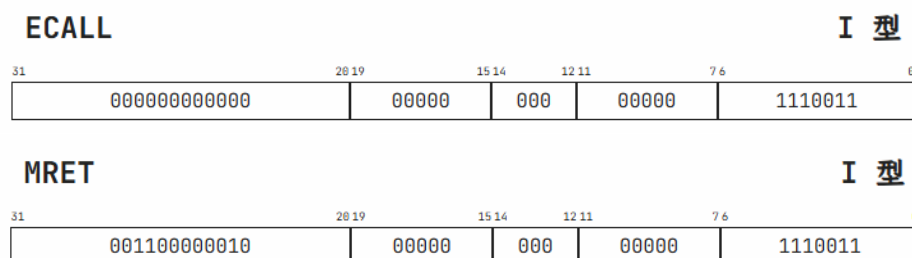
```

```

28     ECALL = (Fun_ecall == 3'b000);
29 end
30 else begin
31     MRET = 0;
32     ECALL = 0;
33 end
34 //
35 illegal = 0;
36
37 end
38
39 default: begin
40     ImmSel = 3'b000;
41     ALUSrc_B = 0;
42     MemtoReg = 3'b000;
43     Jump = 2'b00;
44     Branch = 4'b0000;
45     RegWrite = 0;
46     MemRW = 0;
47     ALU_op = 2'b00;
48     CPU_MIO = 0;
49     signal = 0;
50     width = 2'b00;
51     //
52     CSRTYPE = 3'b000;
53     MRET = 0;
54     ECALL = 0;
55     //
56     illegal = 1;
57 end
58 endcase

```

对于其它指令，新增的信号都为 0，对于 ecall 和 mret 指令，我们观察指令格式



我们可以看到，两种指令的 Fun3 字段都为 000，因此我们可以通过 Fun3 字段来判断是否是 ecall 和 mret 指令，如果是则看 Fun_ecall(inst[22:20]) 和 Fun_

mret(inst[29:28]) 字段来判断是哪一种指令。

如果输入的指令不能被正确译码，那么就抛出非法指令异常，即 illegal=1。

将 SCPUctrl 信号做相应的更改之后，我们的控制信号就准备好了，接下来我们在 Datapath 中做修改以便实现 CSR 指令

对 Datapath 的修改如下

```
1  ...
2  CSRRegs csrreg(
3  .clk(clk),
4  .rst(rst),
5  .raddr(Imm_out[11:0]),
6  .waddr(Imm_out[11:0]),
7  .wdata(CSR_wt_data),
8  .csr_w(CSRTYPE[2]|CSRTYPE[1]|CSRTYPE[0]),
9  .csr_wsc_mode(2'b00),
10 .rdata(CSR_rdata)
11 );
12 ...
13 case(MemtoReg)
14 `MEM2REG_CSR: Wt_data = CSR_rdata;
15 ...
16 //csr_wt_data
17 case(CSRTYPE)
18     3'b001:CSR_wt_data = ALUSrc_A_Reg; //rs1_data csrrw
19     3'b010:CSR_wt_data = ALUSrc_A_Reg | CSR_rdata; //rs1_data csrrs
20     3'b011:CSR_wt_data = (~ALUSrc_A_Reg) & CSR_rdata; //rs1_data csrrc
21     3'b101:CSR_wt_data = {27'b0,rs1}; //rs1 csrrwi
22     3'b110:CSR_wt_data = {27'b0,rs1} | CSR_rdata; //rs1 csrrsi
23     3'b111:CSR_wt_data = {27'b0,~rs1} & CSR_rdata; //rs1 csrrci
24     default:CSR_wt_data = 32'b0;
25 endcase
26 ...
```

至此，我们的 CSR 指令已经实现，接下来我们要实现中断处理程序

RV-int

新建文件 RV_int.v，用于处理中断

```
1
2
```



```

3  module RV_INT(
4      input      clk,
5      input      rst,
6      input      INT,          // 外部中断信号
7      input      ecall,       // ECALL 指令
8      input      mret,        // MRET 指令
9      input      illegal_inst, // 非法指令信号
10     input      l_access_fault, // 数据访存不对齐
11     input      j_access_fault, // 跳转地址不对齐
12     input [31:0] pc_current,   // 当前指令 PC 值
13     input [31:0] PC_next,     // 当前正常指令流下一个 PC 值
14     output [31:0] pc,         // 将执行的指令 PC 值
15     output      CSRregs_wen,  // 是否写入 CSR 寄存器
16
17
18     input [31:0] inst_in,      //
19     input [31:0] mstatus,
20     input [31:0] mtvec,
21     input [31:0] mcause,
22     input [31:0] mtval,
23     input [31:0] mepc,
24
25     output reg [31:0] mepc_bypass_in,
26     output reg [31:0] mcause_bypass_in,
27     output reg [31:0] mtval_bypass_in,
28     output reg [31:0] mstatus_bypass_in,
29     output reg [31:0] mtvec_bypass_in
30 );
31
32 wire wen;
33
34 assign wen = (mstatus==0)?(INT|ecall|illegal_inst|l_access_fault|j_access_
    ↪ s_fault):0; //发生异常时需要写入CSR寄存器
35
36 assign CSRregs_wen=wen|mret; //mret 时需要写入 CSR 寄存器
37
38 always @(*) begin
39     if (rst) begin
40         mepc_bypass_in <= 0;
41         mcause_bypass_in <= 0;
42         mtval_bypass_in <= 0;
43         mstatus_bypass_in <= 0;
44         mtvec_bypass_in <= 32'h320;
45     end
46     else begin

```

```

47     if (mret) begin
48         mstatus_bypass_in <= 0; //可以接受新的异常
49         mcause_bypass_in <= 0;
50         mepc_bypass_in <= 0;
51         mtval_bypass_in <= 0;
52     end
53
54     //mepc and mstatus update
55     else begin
56         if(mstatus=0) begin
57             mepc_bypass_in <= pc_current;
58             mstatus_bypass_in <=
59                 (INT|ecall|illegal_inst|l_access_fault|j_access_fault)
60                 ?32'h1:32'h0; //有异常设置为1不再接受新的异常,
61                 ↪ 无异常设置为0
62                 //mcause update
63             if(INT) begin
64                 mcause_bypass_in <= 32'h1;
65                 mtval_bypass_in <= mtval;
66             end
67             else if(ecall) begin
68                 mcause_bypass_in <= 32'h2;
69                 mtval_bypass_in <= mtval;
70             end
71             else if(illegal_inst) begin
72                 mcause_bypass_in <= 32'h3;
73                 mtval_bypass_in <= inst_in;
74             end
75             else if(l_access_fault) begin
76                 mcause_bypass_in <= 32'h4;
77                 mtval_bypass_in <= mtval;
78             end
79             else if(j_access_fault) begin
80                 mcause_bypass_in <= 32'h5;
81                 mtval_bypass_in <= mtval;
82             end
83             else begin
84                 mcause_bypass_in <= mcause;
85                 mtval_bypass_in <= mtval;
86             end
87         end
88         else begin
89             mepc_bypass_in <= mepc;
90             mstatus_bypass_in <= mstatus;
91             mcause_bypass_in <= mcause;

```

```

89         mtval_bypass_in <= mtval;
90     end
91 end
92 //
93 end
94 end
95
96 assign pc=(rst=1'b1)?32'h0:
97         (mret=1'b1)?mepc:
98         (mstatus=1'b1)?PC_next://正常指令流,继续执行
99         (INT=1'b1)?32'h320:
100        (ecall=1'b1)?32'h320:
101        (illegal_inst=1'b1)?32'h320:
102        (l_access_fault=1'b1)?32'h320:
103        (j_access_fault=1'b1)?32'h320:
104        PC_next;
105 endmodule

```

在这个文件中，我们实现了中断处理程序，接下来我们具体解释各个部分的作用

模块输入端口

- **clk**: 时钟信号，实际上没有用到，因为我没有在这里实例化寄存器
- **rst**: 重置信号。
- **INT**: 外部中断信号，我设置为板子上的 SW[13]，当 SW[13] 为 1 时，表示发生了外部中断。
- **ecall**: ECALL（环境调用）指令触发的信号。
- **mret**: MRET（返回中断）指令信号。
- **illegal_inst**: 非法指令信号。
- **l_access_fault**: 数据访问故障信号（如内存对齐错误）。
- **j_access_fault**: 跳转地址访问故障信号。
- **pc_current**: 当前指令的程序计数器（PC）值。
- **PC_next**: 正常情况下程序应该执行的下一条指令的 PC。
- **inst_in**: 当前执行的指令。

- **mstatus**: 控制寄存器 **MSTATUS** 的值（RISC-V 中的控制状态寄存器）。
- **mtvec**: 异常向量基地址寄存器，定义了中断/异常处理的入口地址。
- **mcause**: 异常原因寄存器。
- **mtval**: 异常值寄存器，用于保存与异常相关的地址或数据。
- **mepc**: 异常发生时的程序计数器值（通常保存导致异常的指令的 PC）。

模块输出端口

- **pc**: 当前执行的指令的 PC 地址。
- **CSRregs_wen**: 控制是否写入 CSR 寄存器，通常在发生异常时为 1。
- **mepc_bypass_in, mcause_bypass_in, mtval_bypass_in, mstatus_bypass_in, mtvec_bypass_in**: 这些寄存器用于控制批量修改寄存器时候的旁路输入

代码逻辑

- **CSRregs_wen**: 此信号表示是否需要写入 CSR 寄存器。当发生外部中断、ECALL、非法指令、数据访问故障、跳转地址故障或 MRET 指令时，信号会被设置为 1，表示需要更新 CSR 寄存器。
- **异常处理过程**:
 - 当复位信号 (**rst**) 为 1 时，所有的异常状态寄存器（如 **mepc_bypass_in, mcause_bypass_in** 等）都会被清零，并将 **mtvec_bypass_in** 设置为 0x320，这是我的 trap 处理程序的异常向量地址。
 - 如果是 **mret**（返回中断）指令，则清空异常相关寄存器，表示可以接受新的异常。
 - 如果 **mstatus** 为 0，表示处理器没有正在处理中断或异常。此时，程序计数器 **pc_current** 会被保存到 **mepc_bypass_in**，并且根据不同的异常信号（**INT, ecall, illegal_inst**, 等），更新 **mcause_bypass_in** 和 **mtval_bypass_in** 的值。
 - 如果 **mstatus** 不为 0，表示当前存在处理中断或异常，此时直接使用从异常发生时的寄存器值（如 **mepc, mcause, mtval** 等）进行处理。

- PC 更新:

- 如果复位信号为 1，PC 会被设置为 0。
- 如果是 MRET 指令，则 PC 会设置为异常发生前的 mepc。
- 如果没有异常发生，则 PC 会设置为 PC_next，继续执行下一条指令。
- 如果发生了某种异常（如 INT, ecall, illegal_inst, 等），PC 会被设置为 0x320，跳转到异常向量地址。

最后 DataPath 中的修改如下

```
1  ...
2  //
3  reg [31:0] CSR_wt_data;
4  wire [31:0] CSR_rdata;
5  wire rv_wen_csr;
6  wire [31:0] mstatus_din;
7  wire [31:0] mstatus_dout;
8  wire [31:0] mtvec_din;
9  wire [31:0] mtvec_dout;
10 wire [31:0] mcause_din;
11 wire [31:0] mcause_dout;
12 wire [31:0] mtval_din;
13 wire [31:0] mtval_dout;
14 wire [31:0] mepc_din;
15 wire [31:0] mepc_dout;
16 //
17
18 ...
19 //
20 CSRRegs csrreg(
21     .clk(clk),
22     .rst(rst),
23     .raddr(Imm_out[11:0]),
24     .waddr(Imm_out[11:0]),
25     .wdata(CSR_wt_data),
26     .csr_w(CSRTYPE[2]|CSRTYPE[1]|CSRTYPE[0]),
27     .csr_wsc_mode(2'b00),
28     .rdata(CSR_rdata),
29     .expt_int(rv_wen_csr),
30     //input fot rv_int
31     .mstatus(mstatus_din),
32     .mtvec(mtvec_din),
33     .mcause(mcause_din),
```

```

34     .mtval(mtval_din),
35     .mepc(mepc_din),
36     //input from rv_int
37     .mepc_bypass_in(mepc_dout),
38     .mcause_bypass_in(mcause_dout),
39     .mtval_bypass_in(mtval_dout),
40     .mstatus_bypass_in(mstatus_dout),
41     .mtvec_bypass_in(mtvec_dout)
42     //
43 );
44 //
45 RV_INT trap(
46     .clk(clk),
47     .rst(rst),
48     .INT(INT),
49     .ecall(ECALL),
50     .mret(MRET),
51     .inst_in(inst_field),
52     .illegal_inst(illegal),
53     .l_access_fault(1'b0),
54     .j_access_fault(1'b0),
55     .pc_current(PC_res), // 当前指令 PC 值
56     .PC_next(PC_in), // 正常指令流下一个 PC 值
57     .pc(PC_trap), // 下一条指令地址
58     .CSRregs_wen(rv_wen_csr),
59     //CSR input
60     .mstatus(mstatus_din),
61     .mtvec(mtvec_din),
62     .mcause(mcause_din),
63     .mtval(mtval_din),
64     .mepc(mepc_din),
65     // 旁路输入
66     .mepc_bypass_in(mepc_dout),
67     .mcause_bypass_in(mcause_dout),
68     .mtval_bypass_in(mtval_dout),
69     .mstatus_bypass_in(mstatus_dout),
70     .mtvec_bypass_in(mtvec_dout)
71     //
72 );
73 ...
74 wire [31:0] PC_trap;
75 ...
76 PC U4(.clk(clk), .rst(rst), .PC_in(PC_trap), .ctrl(1'b1),
77     ↪ .PC_out(PC_res));
78 ...

```

即我们修改指令流，让其经过 RV_int 模块，根据是否发生中断，来判断下一条指令是正常的，还是需要跳转到中断处理程序。

Trap 处理程序

我的 Trap 处理程序如下

```
1 trap:
2   csrrc x22 0x300 x0 #mstatus
3   csrrc x23 0x305 x0 #mtvec
4   csrrc x24 0x342 x0 #mcause
5   csrrc x25 0x343 x0 #mtval
6   csrrc x26 0x341 x0 #mepc
7   addi x14 x0 1
8   beq x14 x24 int_trap
9   addi x26 x26 4
10  csrrw x0 0x341 x26
11  int_trap:
12  mret
```

如果使用 mstatus 将各种不同的处理分开，代码如下

```
1 trap:
2   csrrc x22 0x300 x0 #mstatus
3   csrrc x23 0x305 x0 #mtvec
4   csrrc x24 0x342 x0 #mcause
5   csrrc x25 0x343 x0 #mtval
6   csrrc x26 0x341 x0 #mepc
7   addi x14 x0 1 #INT
8   addi x15 x0 2 # ecall
9   addi x16 x0 3 # illegal
10
11  beq x14 x24 INT
12  beq x15 x24 ex_ecall
13  beq x16 x24 ex_illegal
14
15
16 INT:
17  mret
```

```

18
19 ex_ecall:
20     addi x26, x26, 4
21     csrrw x0, 0x341, x26
22     mret
23
24 ex_illegal:
25     addi x26, x26, 4
26     csrrw x0, 0x341, x26
27     mret

```

测试代码

测试代码，我分为两部分，第一部分为专门测试 CSR 指令，第二部分为在 lab4-3 的基础上，测试修改后的 SCPU 支持原本的指令，并且可以进行中断处理。

```

1     j start
2 dummy:
3     nop
4     nop
5     nop
6     nop
7     nop
8     j dummy
9 start:                                #testbench on csrrw and csrrs
10    li x31, 1
11    li x1, 0xBEEF                       #x1 = 0xBEEF
12    auipc x30, 0
13    csrrw x2, 0x341, x1                 #x2 = mepc, mepc = x1
14    csrrw x2, 0x341, x0                 #x2 = mepc, mepc = x0
15    auipc x30, 0
16    bne x2, x1, dummy                   #x2 == x1?
17    li x3, 5                             #x3 = 5
18    csrrw x0, 0x305, x3                 #mvec = x3
19    li x4, 10                            #x4 = 10
20    csrrs x5, 0x305, x4                 #x5 = mvec, mvec |= x4
21    csrrw x6, 0x305, x0                 #x6 = mvec, mvec = 0
22    or x7, x4, x3                       #x7 = x4|x3

```



```

23     auipc x30, 0
24     bne x7,x6,dummy
25     j    pass_1
26 pass_1:                                #test on csrrc
27     li x31, 2
28     li x2, 15
29     csrwr x0, 0x343, x2    #mtval = 15
30     li x3, 6              #x3 = 6
31     li x4, 9
32     csrrc x0, 0x343, x3    #mtval &= ~x3 (mtval = 9)
33     csrwr x5, 0x343, x0    #x5 = mtval, mtval = 0
34     auipc x30, 0
35     bne x5, x4, dummy
36     j    pass_2
37 pass_2:
38     li x31, 3
39     li x3, 32
40     li x4, 20
41     csrwi x0, 0x342, x20    #mcause = 20 (10100)
42     csrwr x2, 0x342, x3    #x2 = mcause, mcause = 32 (100000)
43     auipc x30, 0
44     bne x2, x4, dummy
45     csrssi x5, 0x342, x0    #x5 = mcause
46     auipc x30, 0
47     bne x3, x5, dummy      #
48     csrssi x3, 0x342, x10   #x3 = mcause, mcause = 0xBEEF
49     csrwr x4, 0x342, x0    #x4 = mcause, mcause = 0
50     li x7, 42
51     auipc x30, 0
52     bne x7, x4, dummy
53     j    pass_3
54 pass_3:
55     li x31, 4
56     li x2, 13              #x2 = 45(01101)
57     li x4, 63
58     li x9, 31
59     csrwi x0, 0x300, x31    #mstatus = 63(11111)
60     csrcci x3, 0x300, x18   # 10100

```

```

61     auipc x30, 0
62     bne x3, x9, dummy
63     csrrsi x5, 0x300, x0      #x5 = mstatus
64     auipc x30, 0
65     bne x2, x5, dummy
66     j    pass_4
67
68 pass_4:
69     li x31, 666
70     j    dummy

```

Listing 1: CSR 指令的测试代码

```

1 ...
2 pass_7:
3 # jalr ->
4     addi x20, x20, 8
5     auipc x30, 0
6     bne x20, x21, dummy
7     li x31, 0x666
8 # exception and interrupt
9     add x0 x0 x0 #nop
10    add x0 x0 x0
11    ecall #ecall
12    addi x1 x1 0x66
13    abc x1 x1 x0 #illigel
14    li x31 0x660
15    j dummy
16 ...
17 # -> address 0x320
18 trap:
19 ...

```

Listing 2: 中断处理的测试代码

即我在 lab4-3 的验收代码后面做了修改，将 x31 设置为 0x666 之后，进入中断测试，结束后返回，再将 x31 设置为 0x660，进入 dummy 循环。由于生成 ROM 核时需要保证处理程序的地址为 0x320，因此在 trap 程序与主程序最后 jump dummy 之间，我加了若干的 nop 指令，使得 ROM 核生成正确。

二、实验结果与分析

CSR 指令

我们需要测试所有的 CSR 指令，以及其对各个 CSR 寄存器的影响。
仿真结构与 lab4-3 的类似

```
1 MyScpu inst0(  
2     .clk(clk),  
3     .rst(rst),  
4     .INT(INT),  
5     .Data_in(douta),  
6     .MIO_ready(CPU_MIO),  
7     .inst_in(spo),  
8     .RAM_wt_bits(wt_bit),  
9     .Addr_out(Addr_out),  
10    .Data_out(Data_out),  
11    .CPU_MIO(CPU_MIO),  
12    .MemRW(MemRW),  
13    .PC_out(PC_out)  
14 );  
15 RAM_B u1(  
16     .clka(~clk),  
17     .wea({4{MemRW}} & wt_bit),  
18     .addra(Addr_out[11:2]),  
19     .dina(Data_out),  
20     .douta(douta)  
21 );  
22  
23 ROMForCSR u2(  
24     .a(PC_out[11:2]),  
25     .spo(spo)  
26 );
```

其中 ROMForCSR 是根据 CSR 测试代码生成的 ROM 核，用于测试 CSR 指令。
其仿真波形分析如下：

我们细节关注对于 mtvec 的修改：

```
1 li x3, 5           #x3 = 5  
2 csrrw x0, 0x305, x3 #mvec = x3
```

```

3 li x4, 10           #x4 = 10
4 csrrs x5, 0x305, x4 #x5 = mvec, mvec |= x4
5 csrrw x6, 0x305, x0 #x6 = mvec, mvec = 0

```

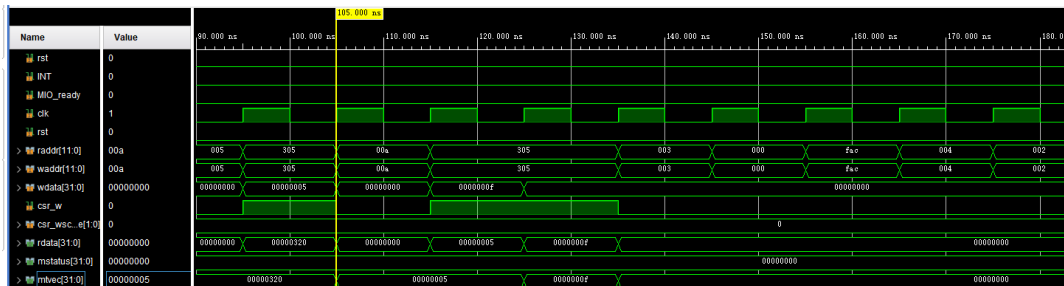


图 1: 对 mtvec 的修改

可以看到 rst 之后，在 mtvec 修改之前，其值为 0x320(trap 程序的入口地址)，之后通过 csrrw 指令，将其修改为 5(0101)，然后通过 csrrs 指令，将其与 10(1010) 或运算，得到 15(1111)，最后通过 csrrw 指令，将其修改为 0(0000)。这与我们的预期相符。

如果我们的 CSR 指令实现正确，那么最后我们的 x31 为十进制的 666；事实也是如此。

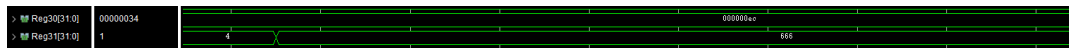


图 2: 最终结果

为了更加直观地验证各种指令执行时非 CSR 寄存器的值也是正确的，我将其写到一起，只对 mstatus 进行修改 (对于其它也是一样的)

```

1 addi x1 x0 0xFE #X1=0xFE
2 csrrw x3 0x300 x1 #X3=0, mstatus=0xFE
3 addi x2 x0 1 #x2=1
4 csrrs x3 0x300 x2 # x3=0xFE mstatus=0xFF
5 addi x2 x2 1 # x2=2
6 csrrc x3 0x300 x2 #x3=0xFF, mstatus=0xFD
7 csrrwi x3 0x300 x2 #x3=0xFD, mstatus=0x02
8 csrrsi x3 0x300 x1 #x3=2, mstatus=3
9 csrrci x3 0x300 x1 #x3=3, mstatus=2

```

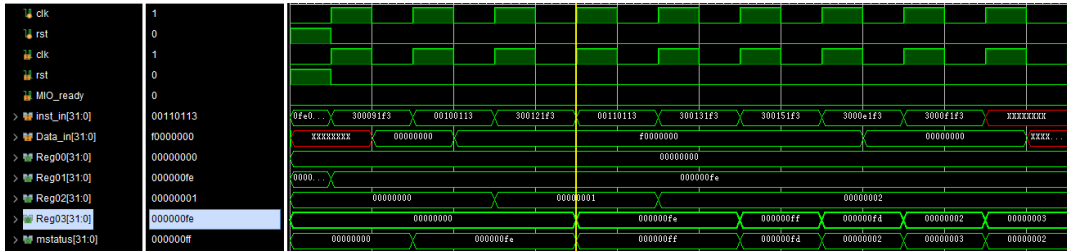


图 3: 对 mstatus 的修改

可以看到首先将 x1 设置为 0xFE，CSR RW 后 mstatus 为 0xFE，CSR RS 后 mstatus 为 0xFF，CSR RS 后 mstatus 为 0xFD，CSR RWI 后 mstatus 为 0x02，CSR RSI 后 mstatus 为 0x03，CSR RCI 后 mstatus 为 0x02，与我们的预期相符。

最后，我们对这一部分进行上板验证，结果也是正确的。

```

RV32I Single Cycle CPU
pc: 00000014   inst: 00000013

x0: 00000000   ra: 0000BEEF   sp: 0000000D   gp: 0000001F   tp: 0000003F
t0: 0000000D   t1: 0000000F   t2: 0000002A   s0: 00000000   s1: 0000001F
a0: 00000000   a1: 00000000   a2: 00000000   a3: 00000000   a4: 00000000
a5: 00000000   a6: 00000000   a7: 00000000   s2: 00000000   s3: 00000000
s4: 00000000   s5: 00000000   s6: 00000000   s7: 00000000   s8: 00000000
s9: 00000000   s10:00000000   s11:00000000   t3: 00000000   t4: 00000000
t5: 000000EC   t6: 0000029A

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000   reg_wen: 1

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

is_branch: 0   is_jal: 0   is_jalr: 0
do_branch: 0   pc_branch: 00000018

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: 0000000D   mcause: 00000000   mepc: 00000000   mtval: 00000000
mtvec: 00000000   mie: 00000000   mip: 00000000

```

图 4: 上板验证

可以看到最后 x31(t6) 为 0x29A，恰好是十进制的 666，与我们的预期相符。

异常处理

将修改后的验收代码写入 ROM 中，然后进行仿真，接下来我们根据代码分析 lab4-3 内容结束后中断处理的仿真波形

```

1   addi x20, x20, 8
2   auipc x30, 0
3   bne x20, x21, dummy
4   li x31, 0x666
5   # exception and interrupt
6   add x0 x0 x0 #nop
7   add x0 x0 x0
8   ecall #ecall
9   addi x1 x1 0x66
10  abc x1 x1 x0 #illigel
11  li x31 0x660
12  j dummy
13
14  #->0x320
15  trap:
16  csrrc x22 0x300 x0 #mstatus
17  csrrc x23 0x305 x0 #mtvec
18  csrrc x24 0x342 x0 #mcause
19  csrrc x25 0x343 x0 #mtval
20  csrrc x26 0x341 x0 #mepc
21  addi x14 x0 1
22  beq x14 x24 int_trap
23  addi x26 x26 4
24  csrrw x0 0x341 x26
25  int_trap:
26  mret

```

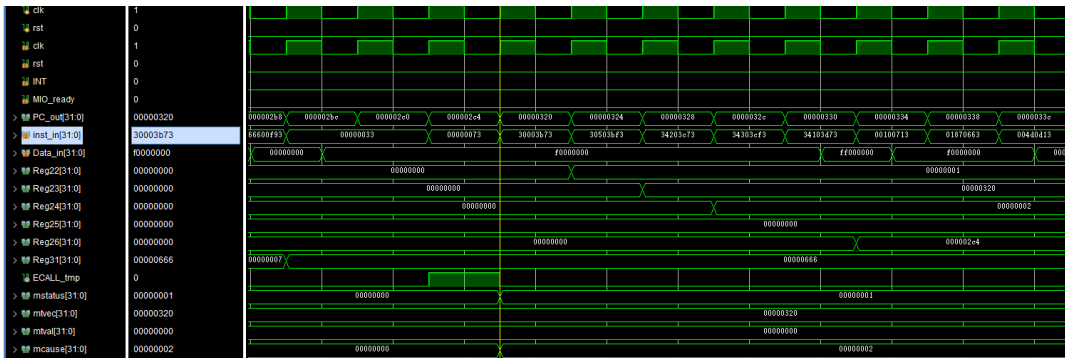


图 5: ecall

我们从 x31 变为 0x666 开始观察；再经过两个 nop 指令 (0x00000033) 之后，我们遇到了 ecall 指令 (0x00000073)，这个时候 ECALL_tmp 信号拉高，说明此时为 ecall 指令；下一个时钟上升沿到来时

- mcause 为 2，代表是一个 ecall 指令
- mepc 为 0x2c4，存储了发生异常指令的 PC 地址
- mtval 为 0，代表没有附加信息
- mtvec 为 0x320，代表异常处理程序的入口地址
- mstatus 为 0x00000001，代表此时正在处理异常，不再接受新的异常程序

接下来依次将 5 个寄存器的值写到 x22-x26 中，此时并没有改变原 CSR 寄存器的值；

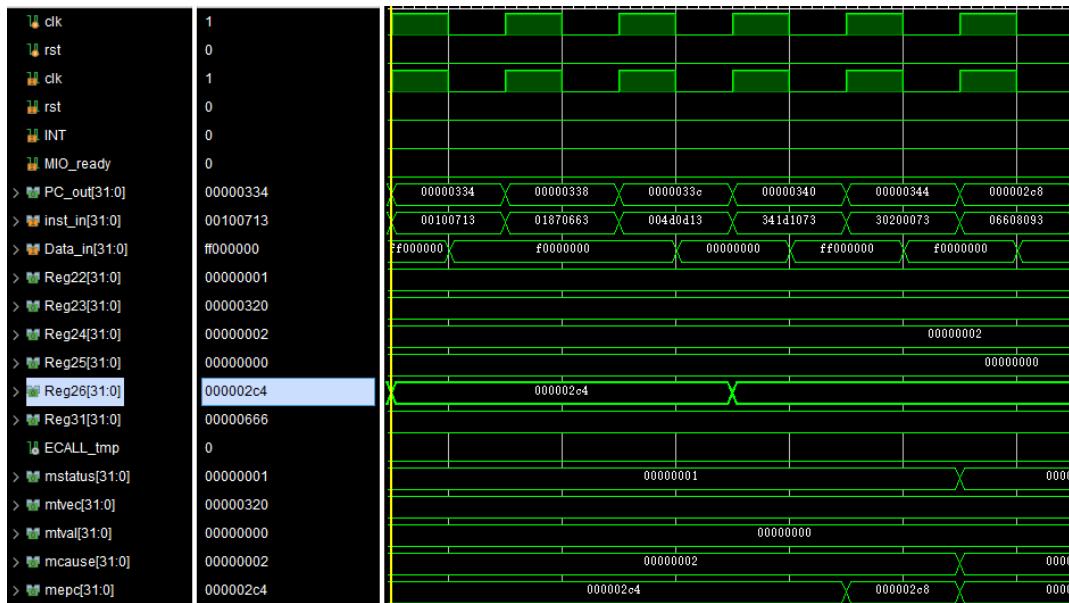


图 6: ecall

接下来进入判断，如果我们的 x24(cause) 为 1，那么我们进入 int_trap，如果不是，说明我们并不是外部中断，此时我们的 x26 会加 4，然后将 x26 的值写回到 mepc 中，然后 mret，这个时候我们的 PC 会跳转到 0x2c8，即我们的异常指令的下一条指令。同时相应的 CSR 寄存器的值会恢复到未异常时的值。

综上，我们的 ecall 指令的异常处理是正确的。

接下来，再将 x1 的加上 0x66，之后，我们遇到了一条异常指令

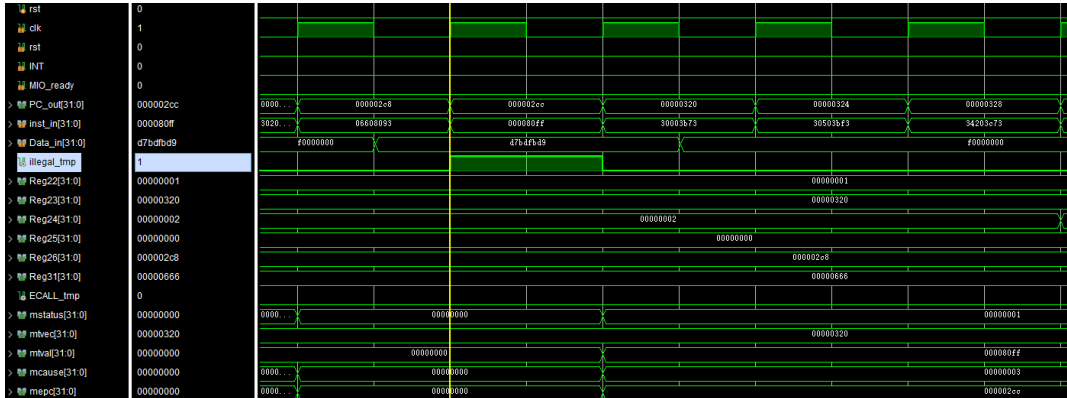


图 7: illegal 指令

其机器码为 (0x11111111), 这是不能被正确译码的, 所以此时抛出 illegal 信号, 下一条指令也是进入到了 trap 处理程序中, 在时钟上升沿到来后

- mcause 为 3, 代表是一个非法指令
- mepc 为 0x2cc, 存储了发生异常指令的 PC 地址
- mtval 为 0x00080ff, 储存了发生异常指令的机器码
- mtvec 为 0x320, 代表异常处理程序的入口地址
- mstatus 为 0x00000001, 代表此时正在处理异常, 不再接受新的异常程序

同样的经过判断后发现不是外部中断, mret 时会回到下一条指令 0x2d0, 同时 CSR 寄存器的值也会恢复到未异常时的值。最后, 将 x31 的值设置为 0x660, 然后跳转到 dummy 处, 结束。

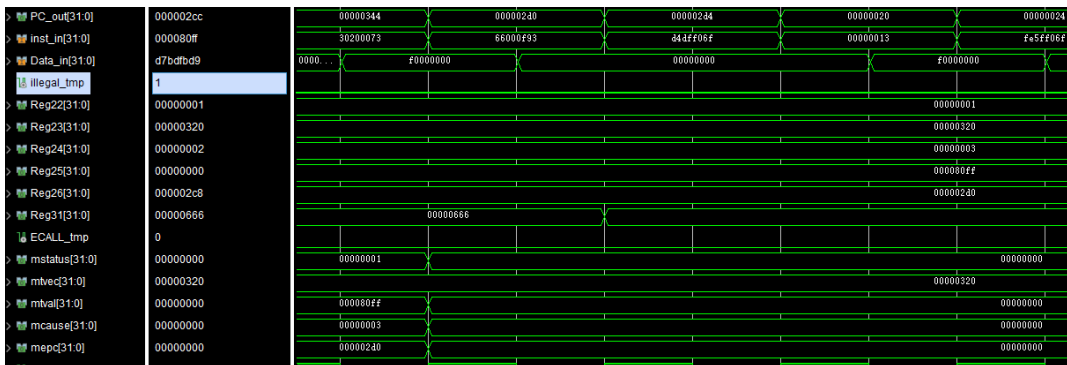


图 8: 结束

下板验证

最后，我们将这一部分写入 ROM 中，将相关 Debug 信号连接出串口，上板验证结果。

验证 ecall 指令

```
RV32I Single Cycle CPU
pc: 00002C4   inst: 0000073

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: 00002A4    s5: 00002A4    s6: 00000000   s7: 00000000   s8: 00000000
s9: 00000000   s10:00000000  s11:00000000  t3: 000000D0  t4: 00D0CBA0
t5: 00002B0    t6: 0000666

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

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

is_branch: 0   is_jal: 0      is_jalr: 0
do_branch: 0   pc_branch: 00002C8

mem_wen: 1     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: 00000320 mie: 00000000 mip: 00000000
```

(a) a

```
pc: 0000320   inst: 3003B73

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: 00002A4    s5: 00002A4    s6: 00000000   s7: 00000000   s8: 00000000
s9: 00000000   s10:00000000  s11:00000000  t3: 000000D0  t4: 00D0CBA0
t5: 00002B0    t6: 0000666

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 16 reg_i_data: 00000001   reg_wen: 1

is_imm: 0      is_auiopc: 0    is_lui: 0      imm: 00000300
a_val: 00000000 b_val: 00000000 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000   cmp_res: 0

is_branch: 0   is_jal: 0      is_jalr: 0
do_branch: 0   pc_branch: 0000324

mem_wen: 1     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: 00002C4 mtval: 00000000
mtvec: 00000320 mie: 00000000 mip: 00000000
```

(b) b

图 9: 发生 ecall

如 a 图，在 x31 得到 0x666 后，经过两个 nop 指令，我们遇到了 ecall 指令，在时钟上升沿到来之后，进入 trap 处理程序，如图 b 所示：此时 mstatus 被设置为 1，代表中断处理中，mcause 为 2，代表处理 ecall；mepc 为 0x2c4，代表 a 中发生 ecall 的地址；

```

RV32I Single Cycle CPU
pc: 00000340    inst: 341D1073

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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000002
s9: 00000000    s10:000002C8    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000666

rs1: 1A rs1_val: 000002C8
rs2: 01 rs2_val: FFFFFFFF
rd: 00 reg_i_data: 000002C4    reg_wen: 1

is_imm: 0    is_auipc: 0    is_lui: 0    imm: 00000341
a_val: 000002C8 b_val: FFFFFFFF alu_ctrl: 0    cmp_ctrl: 0
alu_res: 000002C7    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000344

mem_wen: 1    mem_ren: 0
dmem_o_data: FF000000    dmem_i_data: FF000000    dmem_addr: 000002C7

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000001    mcause: 00000002    mepc: 000002C4    mtval: 00000000
mtvec: 00000320    mie: 00000000    mip: 00000000

```

(a) c

```

RV32I Single Cycle CPU
pc: 00000344    inst: 30200073

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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000002
s9: 00000000    s10:000002C8    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000666

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: 00000302
a_val: 00000000 b_val: 00000000 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000348

mem_wen: 1    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: 000002C8    mtval: 00000000
mtvec: 00000320    mie: 00000000    mip: 00000000

```

(b) d

图 10: 发生 ecall

如图 c 所示，在进行了 trap 程序后，相应的 x22-x26(s6-mstatus;s7-mtvec;s8-mcause;s9-mtval;s10-mepc) 的值被写入；注意到此时 s10=mepc+4；这是因为判断了 mcause 不是 1，所以不是外部中断，所以 mepc+4；在 b 图中这个值被同步给了 mepc；

最后在时钟上升沿到来，mret 指令执行；返回到 0x2c8 处，即异常指令的下一条指令。

```
RV32I Single Cycle CPU
pc: 00002C8      inst: 06608093

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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000002
s9: 00000000    s10:00002C8    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 00002B0     t6: 00000666

rs1: 01 rs1_val: FFFFFFFF
rs2: 06 rs2_val: C0000000
rd:  01 reg_i_data: 00000065  reg_wen: 1

is_imm: 1      is_auiopc: 0    is_lui: 0      imm: 00000066
a_val: FFFFFFFF b_val: 00000066 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000065  cmp_res: 0

is_branch: 0   is_jal: 0      is_jalr: 0
do_branch: 0   pc_branch: 00002CC

mem_wen: 0     mem_ren: 0
dmem_o_data: D7BD65D9  dmem_i_data: 00000000  dmem_addr: 00000065

csr_wen: 0     csr_ind: 000   csr_ctrl: 0     csr_r_data: 00000000
mstatus: 00000000  mcause: 00000000  mepc: 00000000  mtval: 00000000
mtvec:  00000320   mie: 00000000   mip: 00000000
```

图 11: mret 返回

验证 illegal 指令

```
RV32I Single Cycle CPU
pc: 00002CC    inst: 00000FF

x0: 00000000   ra: 00000065   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: 00000001
a5: 00000000   a6: 00000000   a7: 00000000   s2: 00000020   s3: 00000000
s4: 000002A4   s5: 000002A4   s6: 00000001   s7: 00000320   s8: 00000002
s9: 00000000   s10:000002C8  s11:00000000  t3: 00000000  t4: 00D0CBA0
t5: 000002B0   t6: 00000666

rs1: 01 rs1_val: 00000065
rs2: 00 rs2_val: 00000000
rd: 01 reg_i_data: 00000065    reg_wen: 0

is_imm: 0    is_auipc: 0    is_lui: 0    imm: 00000000
a_val: 00000065 b_val: 00000000 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000065    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 000002D0

mem_wen: 0    mem_ren: 0
dmem_o_data: D7BD65D9    dmem_i_data: 00000000    dmem_addr: 00000065

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000000    mtval: 00000000
mtvec: 00000320    mie: 00000000    mip: 00000000
```

(a) a

```
RV32I Single Cycle CPU
pc: 00000320    inst: 30003B73

x0: 00000000   ra: 00000065   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: 00000001
a5: 00000000   a6: 00000000   a7: 00000000   s2: 00000020   s3: 00000000
s4: 000002A4   s5: 000002A4   s6: 00000001   s7: 00000320   s8: 00000002
s9: 00000000   s10:000002C8  s11:00000000  t3: 00000000  t4: 00D0CBA0
t5: 000002B0   t6: 00000666

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 16 reg_i_data: 00000001    reg_wen: 1

is_imm: 0    is_auipc: 0    is_lui: 0    imm: 00000300
a_val: 00000000 b_val: 00000000 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 00000324

mem_wen: 1    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: 000002CC    mtval: 000000FF
mtvec: 00000320    mie: 00000000    mip: 00000000
```

(b) b

图 12: 发生 illegal 指令

如图 a, 返回之后, 将 x1 的值加上 0x66(原来是-1), 变成 0x65, 然后遇到了一条 illegal 指令, 在时钟上升沿到来之后, 进入 trap: 如图 b; mcause 为 3, 代表非法指令; mepc 为 0x2cc, 代表 a 中发生非法指令的地址, mtval 为非法指令的内容; 接下来要做的事与 ecall 的处理类似, mepc=mepc+4; 最后 mret 返回到 0x2d0 处。

```

RV32I Single Cycle CPU
pc: 000002D0    inst: 66000F93

x0: 00000000    ra: 00000065    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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000003
s9: 000080FF    s10:000002D0    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000666

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 1F reg_i_data: 00000660    reg_wen: 1

is_imm: 1    is_auiipc: 0    is_lui: 0    imm: 00000660
a_val: 00000000 b_val: 00000660 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000660    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 000002D4

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 00000660

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000    mepc: 00000000 mtval: 00000000
mtvec: 00000320    mie: 00000000    mip: 00000000

```

(a) mret 返回

```

RV32I Single Cycle CPU
pc: 000002D0    inst: 66000F93

x0: 00000000    ra: 00000065    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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000003
s9: 000080FF    s10:000002D0    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000666

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 1F reg_i_data: 00000660    reg_wen: 1

is_imm: 1    is_auiipc: 0    is_lui: 0    imm: 00000660
a_val: 00000000 b_val: 00000660 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000660    cmp_res: 0

is_branch: 0    is_jal: 0    is_jalr: 0
do_branch: 0    pc_branch: 000002D4

mem_wen: 0    mem_ren: 0
dmem_o_data: 00000000    dmem_i_data: 00000000    dmem_addr: 00000660

csr_wen: 0    csr_ind: 000    csr_ctrl: 0    csr_r_data: 00000000
mstatus: 00000000 mcause: 00000000    mepc: 00000000 mtval: 00000000
mtvec: 00000320    mie: 00000000    mip: 00000000

```

(b) 最后一条指令

图 13: 测试结束

可以看到，mret 返回到 0x2d0，即将执行最后一条指令，将 x31 的值设置为 0x660，当时钟上升沿到来时，x31 变为 0x660；结束，下一条指令为进入 dummy 循环；

验证外部中断 INV

最后，我们来验证外部中断，即 SW[13] 拉高后，代表外部中断发生

```

RV32I Single Cycle CPU

pc: 00000010    inst: 00000013

x0: 00000000    ra: 00000065    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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000001
s9: 00000000    s10:00000010    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000660

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000    reg_wen: 1

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

is_branch: 0    is_jal: 0        is_jalr: 0
do_branch: 0    pc_branch: 00000014

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: 00000320    mie: 00000000    mip: 00000000

```

图 14: 外部中断

此时我们所处的指令地址为 0x10;

```

RV32I Single Cycle CPU

pc: 00000320    inst: 30003B73

x0: 00000000    ra: 00000065    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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000003
s9: 000080FF    s10:000002D0    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000660

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 16 reg_i_data: 00000001    reg_wen: 1

is_imm: 0        is_auiopc: 0    is_lui: 0        imm: 00000300
a_val: 00000000 b_val: 00000000 alu_ctrl: 0    cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0    is_jal: 0        is_jalr: 0
do_branch: 0    pc_branch: 00000324

mem_wen: 1        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: 00000010    mtval: 00000000
mtvec: 00000320    mie: 00000000    mip: 00000000

```

图 15: 外部中断

时钟上升沿到来时，进入 trap 处理程序，mcause 为 1，代表外部中断；mepc 为

0x10, 代表外部中断发生的地址; mtval 为 0, 代表没有附加信息; mtvec 为 0x320, 代表异常处理程序的入口地址; mstatus 为 0x00000001, 代表此时正在处理异常, 不再接受新的异常程序;

```

RV32I Single Cycle CPU
pc: 00000344    inst: 30200073

x0: 00000000    ra: 00000065    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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000001
s9: 00000000    s10:00000010   s11:000000D0   t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000660

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

is_imm: 0        is_auiipc: 0      is_lui: 0        imm: 00000302
a_val: 00000000 b_val: 00000000 alu_ctrl: 0      cmp_ctrl: 0
alu_res: 00000000    cmp_res: 0

is_branch: 0     is_jal: 0        is_jalr: 0
do_branch: 0     pc_branch: 00000348

mem_wen: 1       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: 00000010    mtval: 00000000
mtvec: 00000320    mie: 00000000    mip: 00000000

RV32I Single Cycle CPU
pc: 00000010    inst: 00000013

x0: 00000000    ra: 00000065    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: 00000001
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000001    s7: 00000320    s8: 00000001
s9: 00000000    s10:00000010   s11:000000D0   t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000660

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 00 reg_i_data: 00000000    reg_wen: 1

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

is_branch: 0     is_jal: 0        is_jalr: 0
do_branch: 0     pc_branch: 00000014

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: 00000320    mie: 00000000    mip: 00000000

```

图 16: 外部中断

接下来, 将 x22-x26 的值写入相应的 CSR 寄存器中, 进行是否为外部中断的判

断之后发现为真，直接跳转到 `mret(mepc=mepc)`; 最后返回到外部中断发生的地址 (`0x10`);

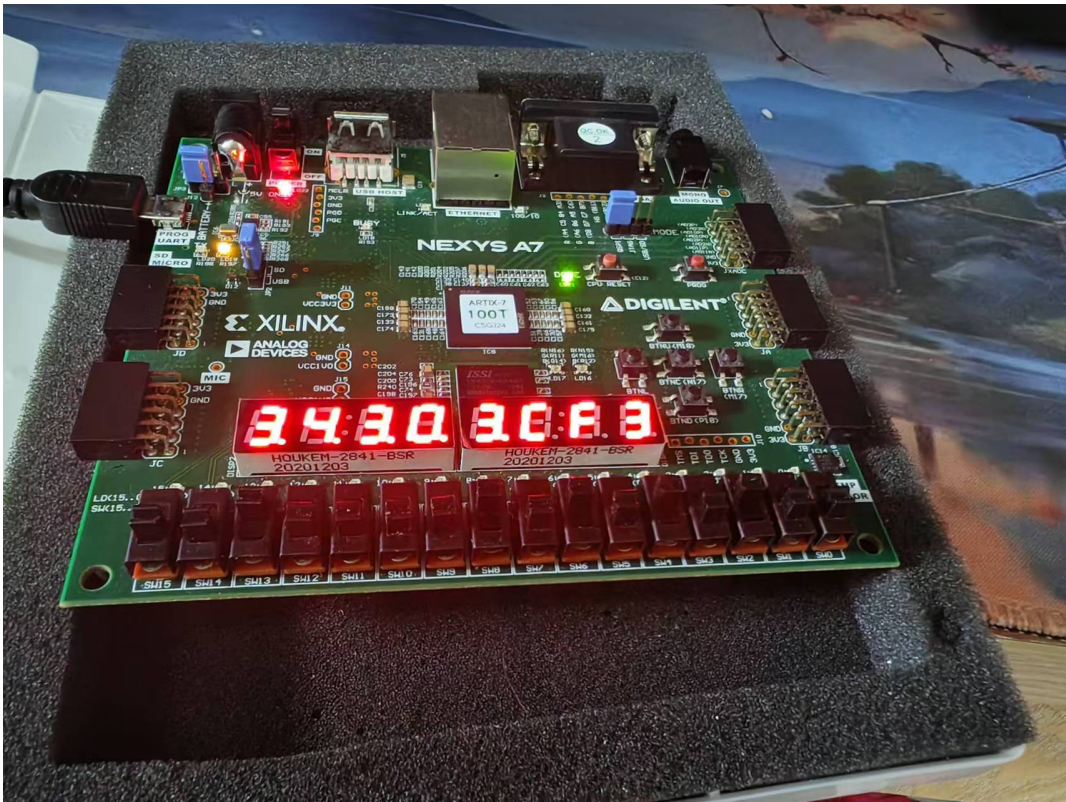


图 17: 外部中断

需要注意的是，在这一过程中我始终保持了 `SW[13]` 为高电平，这里也是为了验证我们在 `mstatus` 为 1 时不再接受新的异常，可以看到，此时执行的指令为中断处理程序内部的程序 (`34303CF3,csrrc x25 0x343 x0`), 而不是跳到新的 `trap` 程序开头 (`30003B73,csrrc x22 0x300 x0`)。

当返回到 `0x10` 时, `msatus` 为 0, 才能继续接受新的异常。

最后，我还测试了当 `ecall` 和外部中断同时发生时，会优先处理外部中断，然后 `mret` 到 `ecall` 的地址，此时如果外部中断还在发生，会再次处理外部中断，直到外部中断结束，才会继续处理 `ecall`。

三、讨论与实验心得

我原本的计划是在期中考试之前把 lab4 写完，但是很显然并没有做到，这导致我 lab4-3 和 lab4-4 之间隔了两周，在期中周正式结束之后，我回到寝室一鼓作气连续写了 8 个小时，把中断基本写完了（多么吉列的豆蒸）

过程中我遇到的一个主要的问题是，在设计 RV_int 模块时，对于旁路输入的赋值，我也使用了时钟上升沿的触发，但是这样会有一个问题，因为我的 CSR 寄存器堆是在 RV_int 的外部初始化的；也是时钟上升沿写，但是写之前其信号并没有准备好；

所以最后我取消了 RV_int 模块的时钟，改为了 always@(*) 的赋值，这样就解决了这个问题，以及一开始我没有完全考虑到各种旁路输入在不同情况下的值，导致除了需要改变的 CSR 寄存器之外，其它寄存器的值很随机，后面我重新审视了一下代码，发现了这个问题，对与每一种情况下的旁路输入都进行了赋值，这样就解决了这个问题。

以及验收的时候，对于旁路输入的写使能，我没有考虑到 mstatus，所以在 trap 程序中处理的时候，如果外部中断打开，mepc 的更新值会使用旁路输入的值而不是我 CSR 指令的值，这会导致我返回不到 mepc+4 的值，最后才改掉了；

总的来说，有了 lab4-3 的基础，lab4-4 的过程就没这么痛苦了，是一个不错的体验。

思考题

为了将 0xDEADBEEF，存入寄存器 t1 中，我们首先看

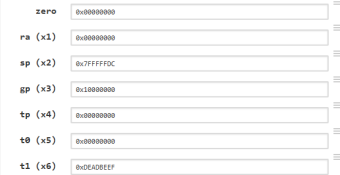
```
1 lui t1, 0xDEADB
2 addi t1, t1, -273 // 0xEEF
```

由于 addi 指令的立即数范围为-2048 到 2047，直接使用加上-273 实际上不会把后面的 0xEEF 加上去，而是将其减去 273，得到的结果是 0xDEADAEEF，可以看到，向 B 借位减了 1，变成 A；所以只需要修改第一条指令即可：

```
1 lui t1, 0xDEADC
2 addi t1, t1, -273 // 0xEEF
```

这样我们就会得到正确的结果

PC	Machine Code	Basic Code	Original Code
0x0	0xDEADC337	lui x6 912092	lui t1, 0xDEADC
0x4	0xEEF30313	addi x6 x6 -273	addi t1, t1, -273



zero 0x00000000
ra (x1) 0x00000000
sp (x2) 0x77ffffdc
gp (x3) 0x10000000
tp (x4) 0x00000000
t0 (x5) 0x00000000
t1 (x6) 0xDEADAEEF

图 18: 正确结果