



## 本科实验报告

课程名称: 计算机体系结构

设计名称: Pipelined CPU

姓 名: 曾帅王异鸣

学 号: 3190105729 3190102780

学 院: 计算机学院

专 业: 计算机科学与技术

班 级: 图灵 1901

指导老师: 姜晓红

2021 年 10 月 16 日

# 目录

1 实验目的 . . . . .	3
2 实验内容 . . . . .	3
3 实验原理 . . . . .	3
3.1 CSR 指令 . . . . .	3
3.2 CSR 寄存器 . . . . .	4
3.3 处理过程 . . . . .	8
3.4 精确 Exception . . . . .	8
3.5 数据通路 . . . . .	12
4 实验步骤与调试 . . . . .	20
4.1 仿真 . . . . .	20
4.1.1 异常处理 . . . . .	20
4.1.2 中断处理 . . . . .	23
4.2 综合 . . . . .	25
4.3 实现 . . . . .	25
4.4 验证设计 . . . . .	25
4.5 生成二进制文件 . . . . .	25
4.6 烧写上板 . . . . .	26
5 实验结果与分析 . . . . .	26
5.1 异常与中断跳转 . . . . .	26
6 讨论与心得 . . . . .	30

# 1 实验目的

本次实验要求我们实现支持中断与异常的 RV32IZicsr 流水线 CPU

1. 理解 CPU 中断异常的原理并掌握它的执行过程
2. 掌握设计 RV32I 流水线支持中断与异常的 CPU 的方法
3. 掌握验证 CPU 正确性的方式

## 2 实验内容

实验的基本要求是实现 RISC-V 架构下，指令集为 RV32I 并支持中断与异常的流水线 CPU。

本实验已给出主要框架，需要完成的实验内容如下：

1. 补充修改数据通路
2. 设计中断异常控制模块
3. 利用测试程序验证 CPU 并观察指令的执行情况

## 3 实验原理

### 3.1 CSR 指令

实现 RSIC-V 的中断与异常首先就需要能够对 CSR 进行读写操作，M 特权模式提供了如下六种指令进行操作。

31	25 24	20 19	15 14	12 11	7 6	0	
	csr		rs1	001	rd	1110011	I csrrw
	csr		rs1	010	rd	1110011	I csrrs
	csr		rs1	011	rd	1110011	I csrrc
	csr		zimm	101	rd	1110011	I csrrwi
	csr		zimm	110	rd	1110011	I csrrsi
	csr		zimm	111	rd	1110011	I csrrci

图 3.1: CSR 指令

介绍完指令格式之后，CSR 每个指令的作用如下

操作名	使用方式	操作方式
csrrw	csrrw rd, csr, rs1	$t = \text{CSRs}[csr]; \text{CSRs}[csr] = x[rs1]; x[rd] = t$
csrrs	csrrs rd, csr, rs1	$t = \text{CSRs}[csr]; \text{CSRs}[csr] = t   x[rs1]; x[rd] = t$
csrrc	csrrc rd, csr, rs1	$t = \text{CSRs}[csr]; \text{CSRs}[csr] = t \& x[rs1]; x[rd] = t$
csrrwi	csrrwi rd, csr, zimm[4:0]	$x[rd] = \text{CSRs}[csr]; \text{CSRs}[csr] = zimm$
csrrsi	csrrsi rd, csr, zimm[4:0]	$t = \text{CSRs}[csr]; \text{CSRs}[csr] = t   zimm; x[rd] = t$
csrrci	csrrci rd, csr, zimm[4:0]	$t = \text{CSRs}[csr]; \text{CSRs}[csr] = t \& zimm; x[rd] = t$

## 3.2 CSR 寄存器

RISC-V 定义了一些控制和状态寄存器 (CSR), 用于配置或记录一些运行的状态, 而 CSR 寄存器是处理器内核内部的寄存器, 使用专有的 12 位地址编码空间, 此次实验使用到了下述寄存器

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	<code>mvendorid</code>	Vendor ID.
0xF12	MRO	<code>marchid</code>	Architecture ID.
0xF13	MRO	<code>mimpid</code>	Implementation ID.
0xF14	MRO	<code>mhartid</code>	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	<code>mstatus</code>	Machine status register.
0x301	MRW	<code>misa</code>	ISA and extensions
0x302	MRW	<code>medeleg</code>	Machine exception delegation register.
0x303	MRW	<code>mideleg</code>	Machine interrupt delegation register.
0x304	MRW	<code>mie</code>	Machine interrupt-enable register.
0x305	MRW	<code>mtvec</code>	Machine trap-handler base address.
0x306	MRW	<code>mcounteren</code>	Machine counter enable.
Machine Trap Handling			
0x340	MRW	<code>mscratch</code>	Scratch register for machine trap handlers.
0x341	MRW	<code>mepc</code>	Machine exception program counter.
0x342	MRW	<code>mcause</code>	Machine trap cause.
0x343	MRW	<code>mtval</code>	Machine bad address or instruction.
0x344	MRW	<code>mip</code>	Machine interrupt pending.

图 3.2: CSR 寄存器

下面对各个控制与状态寄存器做具体的描述:

**mtvec** Riscv 处理器 trap 后跳入的 PC 地址由一个叫做机器模式异常入口地址寄存器 mtvec 的 csr 寄存器指定。mtvec 是一个可读可写的寄存器, 软件可以编程设定它的值。

xlen-1												2	1	0
Interrupt												BASE[xlen-1:2](WARL)		
												xlen-2		2

图 3.3: mtvec

- 假设 mode 的值为 0，则所有的异常响应时处理器均跳转到 base 值指示的 pc 地址
- 假设 mode 的值为 1，则狭义的异常发生时候，处理器均跳转到 base 值指示的 pc 地址。狭义的中断发生时候，处理器跳转到  $\text{base} + 4 * \text{cause}$  值指示的 pc 地址。  
cause 的值表示中断对应的异常编号 (exception code)。

**mcause** Riscv 架构规定，进入异常时候，机器模式异常原因寄存器 mcause 被同时更新，以反映当前的异常种类，软件可以通过读此寄存器查询造成异常的具体原因。

xlen-1	xlen-2													0
Interrupt													异常编码(WLRL)	
1													xlen-1	

图 3.4: mcause

异常编号域定义了中断和异常类型，如下表所示：

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

图 10.3: RISC-V 异常和中断的原因。中断时 mcause 的最高有效位置 1，同步异常时置 0，且低有效位标识了中断或异常的具体原因。只有在实现了监管者模式时才能处理监管者模式中断和页面错误异常

图 3.5: cause

**mepc** Riscv 架构定义异常的返回地址由机器模式异常 PC 寄存器 mepc 保存。在进入异常时候，硬件将自动更新 mepc 寄存器的值为当前遇到异常的指令 PC 值（即当前程序的停止执行点）。该寄存器的值将作为异常的返回地址，在异常结束后，能够使用它保存的 pc 值返回之前停止执行的程序点。注意：mepc 虽然被自动更新，但它是可读可写的，软件可以直接读写该寄存器的值。

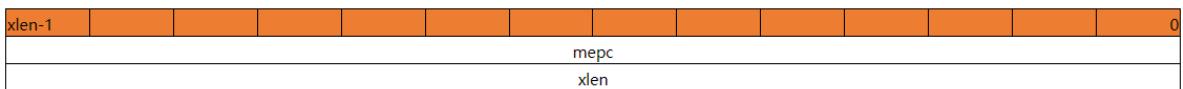


图 3.6: mepc

出现中断时候，中断返回地址 mepc 的值被更新为下一条尚未执行的指令。出现异常时候，中断返回地址 mepc 的值被更新为当前发生异常的指令 pc。注意：如果异常是有 ecall 和 ebreak 产生，由于 mepc 的值被更新为 ecall 或者 ebreak 指令自己的 PC。因此，在异常返回时候，如果直接使用 mepc 保存的 pc 值作为返回地址，则会再次进入异常，形成死循环。正确的做法是在异常处理程序中软件改变 mepc 指向下一条指令。

**mtval** Riscv 规定，在进入异常时候，硬件将自动更新机器模式异常值寄存器 mtval，以反映引起当前异常的存储器访问地址或者指令编码。

mtval	xlen-1																								0
		mtval																							
			xlen																						

图 3.7: mtval

如果是由访问存储器造成的异常，比如硬件断点，取指令，存储器读写造成的异常，则将存储器访问的地址更新到 mtval。如果是由非法指令造成的异常，则将该指令的指令编码更新到 mtval 寄存器中。

**mstatus** mstatus 是机器模式下的状态寄存器。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SD		WPRI		TSR	TW	TVM	MXR	SUM	MPRV	XSI[1]	XSI[0]	FSI[1]	FSI[0]	MPP[1]	MPP[0]	WPRI	SPP	MPIE	WPRI	SPIE	UPIE	MIE	WPRI	SIE	UIE						

图 3.8: mstatus

MIE 域表示全局中断使能。当该 MIE 域值为 1 时，表示所有中断的全局开关打开，当 MIE 域的值为 0 时候，表示全局关闭所有中断。MPIE 用于保存进入异常之前 MIE 域的值。MPP 用于保存进入异常之前特权模式的值。处理器进入异常时候：MPIE 域的值被更新为 MIE 的值。MIE 的值被更新为 0（意味着进入异常，中断被屏蔽）。MPP 的值被更新为异常发生前的模式（如果只实现机器模式，则 MPP 的值永远为 11）。

**mie/mip** Riscv 架构上定义的异常是不可屏蔽的，但狭义上的中断是可以屏蔽的，通过设置 mie 寄存器来屏蔽中断。mip 寄存器用于查询中断的等待状态，软件可以通过读 mip 寄存器达到查询中断状态的结果。

mip	xlen-1	11	10	9	8	7	6	5	4	3	2	1	0
	WPRI	MEIP	WPRI	SEIP	UEIP	MTIP	WPRI	STIP	UTIP	MSIP	WPRI	SSIP	USIP
xlen-12		1	1	1	1	1	1	1	1	1	1	1	1
mie	xlen-1	11	10	9	8	7	6	5	4	3	2	1	0
	WPRI	MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE
xlen-12		1	1	1	1	1	1	1	1	1	1	1	1

图 3.9: mipmie

机器模式下外部中断的屏蔽由 csr 寄存器 mie 中 MEIE 域控制，等待标志 (pending) 则反映在 csr 寄存器 mip 的 MEIP 域。mip 和 mie 寄存器的高 20 位可以用于扩展其它的自定义中断类型。

### 3.3 处理过程

发生异常/中断时，硬件自动经历如下的状态转换：

1. 异常指令的 PC 被保存在 mepc 中，PC 被设置为 mtvec。mepc 指向导致异常的指令；对于中断，它指向中断处理后应该恢复执行的位置。
2. 根据异常来源设置 mcause，并将 mtval 设置为出错的地址或者其它适用于特定异常的信息字。
3. 把控制状态寄存器 mstatus 中的 MIE 位置零以禁用中断，并把先前的 MIE 值保留到 MPIE 中。
4. 发生异常之前的权限模式保留在 mstatus 的 MPP 域中，再把权限模式更改为 M。

从中断与异常处理返回时需要用到 mret 指令，处理过程如下：

1. PC 被设置为 mepc
2. 恢复 mstatus 的 MIE 位
3. 恢复保存在 mstatus 的 MPP 域中的权限模式

### 3.4 精确 Exception

在本次实验中，我们实现的是精确异常。也即异常所有之前的指令必须回收，异常之后的指令一律不得回收，即不可以更新体系结构状态。为实现这一要求，我们做了如下设计：

1. 所以异常的跳转均需要在 WB 阶段进行，也就是说前几个阶段侦测到的异常不会立刻进入异常处理，而是要流入 WB 后再同一处理
2. 如果多阶段产生了异常，处理最早产生的异常
3. 使用异常向量来完成这一操作：如果异常被检测到异常信号将会被加入异常向量并且取消所有修改系统状态的写信号

根据上述原理，我组设计的中断异常模块如下：

```
1 `timescale 1ns / 1ps
2
3 module ExceptionUnit(
4     input clk , rst ,
```

```

5   input csr_rw_in ,
6   input [1:0] csr_wsc_mode_in ,
7   input csr_w_imm_mux,
8   input [11:0] csr_rw_addr_in ,
9   input [31:0] csr_w_data_reg ,
10  input [4:0] csr_w_data_imm ,
11  output [31:0] csr_r_data_out ,
12
13  input interrupt ,
14  input illegal_inst ,
15  input l_access_fault ,
16  input s_access_fault ,
17  input ecall_m ,
18  input [31:0] faultaddr ,
19
20  input mret ,
21
22  input [31:0] epc_cur ,
23  input inst_is_zero ,
24  input [31:0] epc_next ,
25  output [31:0] PC_redirect ,
26  output redirect_mux ,
27
28  output reg_FD_flush , reg_DE_flush , reg_EM_flush , reg_MW_flush ,
29  output RegWrite_cancel
30 );
31
32  wire [31:0] wdata;
33
34  reg [31:0] CSR [0:15];
35
36  wire ex_in=interrupt | illegal_inst | l_access_fault | s_access_fault |
37    ecall_m ;
38  localparam mstatus = 0;
39  localparam mie      = 4;
40  localparam mtvec     = 5;
41  localparam mepc      = 9;
42  localparam mcause     = 10;
43  localparam mtval      = 11;
44  localparam mip       = 12;
45
// Address mapping. The address is 12 bits , but only 4 bits are

```

```

        used in this module.

46   wire raddr_valid = csr_rw_addr_in[11:7] == 5'h6 && csr_rw_addr_in
47     [5:3] == 3'h0;
48   wire [3:0] raddr_map = (csr_rw_addr_in[6] << 3) + csr_rw_addr_in
49     [2:0];
50   wire waddr_valid = csr_rw_addr_in[11:7] == 5'h6 && csr_rw_addr_in
51     [5:3] == 3'h0;
52   wire [3:0] waddr_map = (csr_rw_addr_in[6] << 3) + csr_rw_addr_in
53     [2:0];

54   assign csr_r_data_out = CSR[raddr_map];
55   assign wdata = csr_w_imm_mux ? csr_w_data_imm : csr_w_data_reg;

56   //According to the diagram, design the Exception Unit
57   always@(posedge clk or posedge rst) begin
58     if(rst) begin
59       CSR[0] = 32'h88;
60       CSR[1] = 0;
61       CSR[2] = 0;
62       CSR[3] = 0;
63       CSR[4] = 32'hfff;
64       CSR[5] = 0;
65       CSR[6] = 0;
66       CSR[7] = 0;
67       CSR[8] = 0;
68       CSR[9] = 0;
69       CSR[10] = 0;
70       CSR[11] = 0;
71       CSR[12] = 0;
72       CSR[13] = 0;
73       CSR[14] = 0;
74       CSR[15] = 0;
75     end
76     else if(csr_rw_in && ~ex_in) begin
77       case(csr_wsc_mode_in)
78         2'b01: CSR[waddr_map] = wdata;
79         2'b10: CSR[waddr_map] = CSR[waddr_map] |
80                   wdata;
81         2'b11: CSR[waddr_map] = CSR[waddr_map] & ~
82                   wdata;
83         default: CSR[waddr_map] = wdata;
84       endcase

```

```

81      end
82      else if(ex_in && CSR[mstatus][3]==1 && ~inst_is_zero )
83          begin
84              CSR[mepc] = interrupt ? epc_next : epc_cur ;
85              if (interrupt) begin
86                  CSR[mcause]=32'h800b ;
87              end
88              else begin
89                  if(ecall_m) CSR[mcause]=32'hb;
90                  else if(s_access_fault) CSR[mcause]=32'h6;
91                  else if(l_access_fault) CSR[mcause]=32'h4;
92                  else if(illegal_inst) CSR[mcause]=32'h2;
93              end
94              if (~interrupt) begin
95                  if(ecall_m) CSR[mtval]=0;
96                  else if(s_access_fault) CSR[mtval]=
97                      faultaddr;
98                  else if(l_access_fault) CSR[mtval]=
99                      faultaddr;
100                 else if(illegal_inst) CSR[mtval]=epc_cur;
101             end
102             CSR[mstatus][7]=CSR[mstatus][3];
103             CSR[mstatus][3]=0;
104         end
105     end
106
107     assign PC_redirect = (ex_in & CSR[mstatus][3] & ~inst_is_zero) ?
108         CSR[mtvec] :
109             mret ? CSR[mepc] : 0;
110     assign redirect_mux = (ex_in & CSR[mstatus][3] & ~inst_is_zero) |
111         mret;
112     assign reg_FD_flush= (ex_in & CSR[mstatus][3] & ~inst_is_zero) |
113         mret;
114     assign reg_DE_flush= (ex_in & CSR[mstatus][3] & ~inst_is_zero) |
115         mret;
116     assign reg_EM_flush= (ex_in & CSR[mstatus][3] & ~inst_is_zero) |
117         mret;
118     assign reg_MW_flush= (ex_in & CSR[mstatus][3] & ~inst_is_zero) |
119         mret;

```

```

114     assign RegWrite_cancel= ex_in & CSR[mstatus][3] & ~inst_is_zero;
115
116 endmodule

```

### 3.5 数据通路

完成了个部件的设计，数据通路就是将各个部分连接起来，保证各功能正常实现的关键部分，下面是总体的原理图

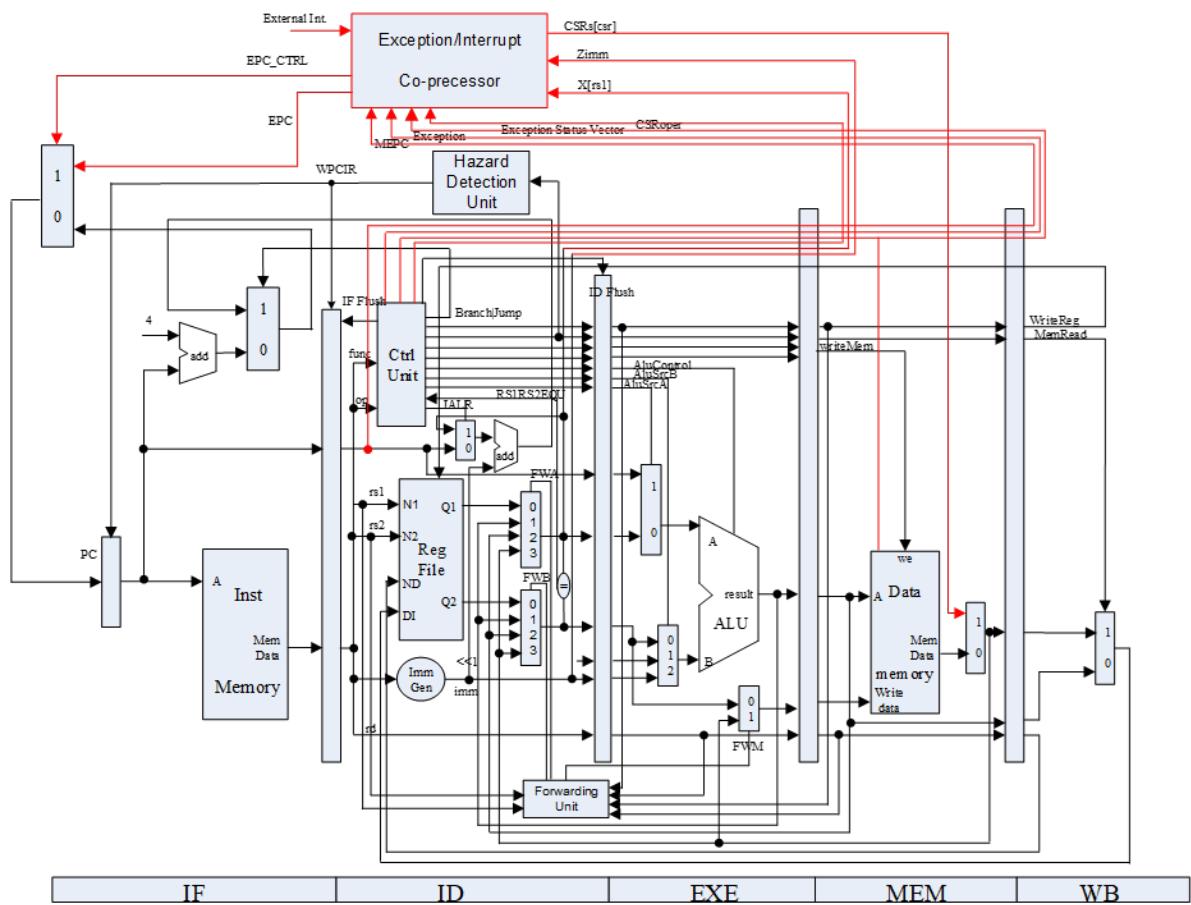


图 3.10: DataPath

数据通路的补充就是根据原理图连线即可，具体的源代码如下

```

1 `timescale 1ns / 1ps
2
3
4 module RV32core(
5     input debug_en, // debug enable
6     input debug_step, // debug step clock

```

```

7      input [6:0] debug_addr, // debug address
8      output[31:0] debug_data, // debug data
9      input clk, // main clock
10     input rst, // synchronous reset
11     input interrupter // interrupt source, for future use
12 );
13
14 wire debug_clk;
15
16 debug_clk clock (.clk(clk), .debug_en(debug_en), .debug_step(debug_step),
17 ,.debug_clk(debug_clk));
18
19 wire Branch_ctrl, JALR, RegWrite_ctrl, mem_w_ctrl, mem_r_ctrl,
20     ALUSrc_A_ctrl, ALUSrc_B_ctrl, DatatoReg_ctrl, rs1use_ctrl,
21     rs2use_ctrl,
22     MRET, csr_rw_ctrl, csr_w_imm_mux_ctrl;
23 wire[1:0] hazard_optype_ctrl, exp_vector_ctrl;
24 wire[2:0] ImmSel_ctrl, cmp_ctrl;
25 wire[3:0] ALUControl_ctrl;
26
27 wire forward_ctrl_ls;
28 wire[1:0] forward_ctrl_A, forward_ctrl_B;
29
30 wire PC_EN_IF;
31 wire [31:0] PC_IF, next_PC_IF, PC_4_IF, inst_IF, final_PC_IF;
32
33 wire reg_FD_stall, reg_FD_flush, isFlushed_ID, cmp_res_ID;
34 wire [31:0] jump_PC_ID, PC_ID, inst_ID, Debug_regs, rs1_data_reg,
35     rs2_data_reg,
36     Imm_out_ID, rs1_data_ID, rs2_data_ID, addA_ID;
37
38 wire reg_DE_flush, isFlushed_EXE, RegWrite_EXE, mem_w_EXE, mem_r_EXE,
39     ALUSrc_A_EXE, ALUSrc_B_EXE, ALUzero_EXE, ALUoverflow_EXE,
40     DatatoReg_EXE,
41     csr_rw_EXE, csr_w_imm_mux_EXE, mret_EXE;
42 wire[1:0] exp_vector_EXE;
43 wire[2:0] u_b_h_w_EXE;
44 wire[3:0] ALUControl_EXE;
45 wire[4:0] rs1_EXE, rs2_EXE, rd_EXE;
46 wire[31:0] ALUout_EXE, PC_EXE, inst_EXE, rs1_data_EXE, rs2_data_EXE,
47     Imm_EXE,
48     ALUA_EXE, ALUB_EXE, Dataout_EXE;

```

```

44
45   wire redirect_mux_exp , reg_FD_flush_exp , reg_DE_flush_exp ,
46     reg_EM_flush_exp , reg_MW_flush_exp , RegWrite_cancel_exp ;
47   wire [31:0] PC_redirect_exp ;
48
49   wire isFlushed_MEMORY , RegWrite_MEMORY , DatatoReg_MEMORY , mem_w_MEMORY , mem_r_MEMORY ,
50     csr_rw_MEMORY , csr_w_imm_mux_MEMORY , mret_MEMORY , l_access_fault_MEMORY ,
51     s_access_fault_MEMORY ;
52   wire [1:0] exp_vector_MEMORY ;
53   wire [2:0] u_b_h_w_MEMORY ;
54   wire [4:0] rs1_MEMORY , rd_MEMORY ;
55   wire [31:0] ALUout_MEMORY , PC_MEMORY , inst_MEMORY , Dataout_MEMORY , Datain_MEMORY ,
56     rs1_data_MEMORY , CSRout_MEMORY , RAMout_MEMORY ;
57
58   wire isFlushed_WB , RegWrite_WB , DatatoReg_WB ;
59   wire [3:0] exp_vector_WB ;
60   wire [4:0] rd_WB ;
61   wire [31:0] wt_data_WB , PC_WB , inst_WB , ALUout_WB , Datain_WB ;
62
63
64 // IF
65 REG32 REG_PC(.clk(debug_clk), .rst(rst), .CE(PC_EN_IF), .D(final_PC_IF),
66   Q(PC_IF));
67
68 add_32 add_IF(.a(PC_IF), .b(32'd4), .c(PC_4_IF));
69
70 MUX2T1_32 mux_IF(.I0(PC_4_IF), .I1(jump_PC_ID), .s(Branch_ctrl), .o(
71   next_PC_IF));
72
73 MUX2T1_32 redirectPC (.I0(next_PC_IF) ,. I1 (PC_redirect_exp) ,. s (
74   redirect_mux_exp) ,. o (final_PC_IF));
75
76 ROM_D inst_rom (.a(PC_IF[8:2]), .spo(inst_IF));
77
78
79 // ID
80 REG_IF_ID reg_IF_ID (.clk(debug_clk), .rst(rst), .EN(1'b1), .Data_stall(
81   reg_FD_stall),
82   .flush(reg_FD_flush | reg_FD_flush_exp), .PCOUT(PC_IF), .IR(inst_IF)
83   ,
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

80     .IR_ID(inst_ID) ,. PCurrent_ID(PC_ID) ,. isFlushed(isFlushed_ID));
81
82 CtrlUnit ctrl(.inst(inst_ID) ,. cmp_res(cmp_res_ID) ,. Branch(Branch_ctrl)
83     ,. ALUSrc_A(ALUSrc_A_ctrl) ,
84     .ALUSrc_B(ALUSrc_B_ctrl) ,. DatatoReg(DatatoReg_ctrl) ,. RegWrite(
85         RegWrite_ctrl) ,
86     .mem_w(mem_w_ctrl) ,. mem_r(mem_r_ctrl) ,. rs1use(rs1use_ctrl) ,. rs2use
87         (rs2use_ctrl) ,
88     .hazard_optype(hazard_optype_ctrl) ,. ImmSel(ImmSel_ctrl) ,. cmp_ctrl(
89         cmp_ctrl) ,
90     .ALUControl(ALUControl_ctrl) ,. JALR(JALR) ,. MRET(MRET) ,. csr_rw(
91         csr_rw_ctrl) ,
92     .csr_w_imm_mux(csr_w_imm_mux_ctrl) ,. exp_vector(exp_vector_ctrl));
93
94
95 Regs register(.clk(debug_clk) ,.rst(rst) ,.L_S(RegWrite_WB & ~
96     RegWrite_cancel_exp) ,
97     .R_addr_A(inst_ID[19:15]) ,.R_addr_B(inst_ID[24:20]) ,
98     .rdata_A(rs1_data_reg) ,.rdata_B(rs2_data_reg) ,
99     .Wt_addr(rd_WB) ,.Wt_data(wt_data_WB) ,
100    .Debug_addr(debug_addr[4:0]) ,.Debug_regs(Debug_regs));
101
102
103 ImmGen imm_gen(.ImmSel(ImmSel_ctrl) ,.inst_field(inst_ID) ,.Imm_out(
104     Imm_out_ID));
105
106 MUX4T1_32 mux_forward_A(.I0(rs1_data_reg) ,.I1(ALUout_EXE) ,.I2(
107     ALUout_MEM) ,.I3(Datain_MEM) ,
108     .s(forward_ctrl_A) ,.o(rs1_data_ID));
109
110 MUX4T1_32 mux_forward_B(.I0(rs2_data_reg) ,.I1(ALUout_EXE) ,.I2(
111     ALUout_MEM) ,.I3(Datain_MEM) ,
112     .s(forward_ctrl_B) ,.o(rs2_data_ID));
113
114 MUX2T1_32 mux_branch_ID(.I0(PC_ID) ,.I1(rs1_data_ID) ,.s(JALR) ,.o(
115     addA_ID));
116
117 add_32 add_branch_ID(.a(addA_ID) ,.b(Imm_out_ID) ,.c(jump_PC_ID));
118
119 cmp_32 cmp_ID(.a(rs1_data_ID) ,.b(rs2_data_ID) ,.ctrl(cmp_ctrl) ,.c(
120     cmp_res_ID));
121
122 HazardDetectionUnit hazard_unit(.clk(debug_clk) ,.Branch_ID(Branch_ctrl)
123     ,.rs1use_ID(rs1use_ctrl) ,

```

```

110    .rs2use_ID(rs2use_ctrl), .hazard_optype_ID(hazard_optype_ctrl), .
111      rd_EXE(rd_EXE),
112      .rd_MEM(rd_MEM), .rs1_ID(inst_ID[19:15]), .rs2_ID(inst_ID[24:20]), .
113        rs2_EXE(rs2_EXE),
114      .PC_EN_IF(PC_EN_IF), .reg_FD_stall(reg_FD_stall), .reg_FD_flush(
115        reg_FD_flush),
116      .reg_DE_flush(reg_DE_flush), .forward_ctrl_ls(forward_ctrl_ls), .
117        forward_ctrl_A(forward_ctrl_A),
118      .forward_ctrl_B(forward_ctrl_B));
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
// EX
REG_ID_EX reg_ID_EX(.clk(debug_clk), .rst(rst), .EN(1'b1),
. flush(reg_DE_flush | reg_DE_flush_exp | isFlushed_ID),
.IR_ID(inst_ID), .PCurrent_ID(PC_ID), .rs1_addr(inst_ID[19:15]), .
rs2_addr(inst_ID[24:20]),
.rs1_data(rs1_data_ID), .rs2_data(rs2_data_ID), .Imm32(Imm_out_ID), .
rd_addr(inst_ID[11:7]),
.ALUSrc_A(ALUSrc_A_ctrl), .ALUSrc_B(ALUSrc_B_ctrl), .ALUC(
    ALUControl_ctrl), .DatatoReg(DatatoReg_ctrl),
.RegWrite(RegWrite_ctrl), .WR(mem_w_ctrl), .u_b_h_w(inst_ID[14:12]),
    ,.mem_r(mem_r_ctrl),
.csr_rw(csr_rw_ctrl), .csr_w_imm_mux(csr_w_imm_mux_ctrl), .mret(MRET),
    ,
.exp_vector(exp_vector_ctrl),
.PCurrent_EX(PC_EXE), .IR_EX(inst_EXE), .rs1_EX(rs1_EXE), .rs2_EX(
    rs2_EXE),
.A_EX(rs1_data_EXE), .B_EX(rs2_data_EXE), .Imm32_EX(Imm_EXE), .rd_EX(
    rd_EXE),
.ALUSrc_A_EX(ALUSrc_A_EXE), .ALUSrc_B_EX(ALUSrc_B_EXE), .ALUC_EX(
    ALUControl_EXE),
.DatatoReg_EX(DatatoReg_EXE), .RegWrite_EX(RegWrite_EXE), .WR_EX(
    mem_w_EXE),
.u_b_h_w_EX(u_b_h_w_EXE), .mem_r_EX(mem_r_EXE), .isFlushed(
    isFlushed_EXE),
.csr_rw_EX(csr_rw_EXE), .csr_w_imm_mux_EX(csr_w_imm_mux_EXE), .
mret_EX(mret_EXE),
.exp_vector_EX(exp_vector_EXE));
MUX2T1_32 mux_A_EXE(.I0(rs1_data_EXE), .I1(PC_EXE), .s(ALUSrc_A_EXE), .o(
    ALUA_EXE));

```

```

136
137 MUX2T1_32 mux_B_EXE(. I0(rs2_data_EXE) ,. I1(Imm_EXE) ,. s(ALUSrc_B_EXE) ,. o
138 (ALUB_EXE)) ;
139
140 ALU alu (.A(ALUA_EXE) ,.B(ALUB_EXE) ,. Control(ALUControl_EXE) ,
141 . res(ALUout_EXE) ,. zero(ALUzero_EXE) ,. overflow(ALUoverflow_EXE)) ;
142
143
144
145 // MEM
146 REG_EX_MEM reg_EXE_MEM(. clk(debug_clk) ,. rst(rst) ,. EN(1'b1) ,. flush(
147 reg_EM_flush_exp | isFlushed_EXE) ,
148 .IR_EX(inst_EXE) ,. PCurrent_EX(PC_EXE) ,. ALUO_EX(ALUout_EXE) ,. B_EX(
149 Dataout_EX) ,
150 .rd_EX(rd_EX) ,. rs1_EX(rs1_EX) ,. rs1_data_EX(rs1_data_EX) ,.
151 DatatoReg_EX(DatatoReg_EX) ,
152 .RegWrite_EX(RegWrite_EX) ,. WR_EX(mem_w_EXE) ,. u_b_h_w_EX(
153 u_b_h_w_EXE) ,. mem_r_EX(mem_r_EXE) ,
154 .csr_rw_EX(csr_rw_EXE) ,. csr_w_imm_mux_EX(csr_w_imm_mux_EXE) ,.
155 mret_EX(mret_EXE) ,
156 .exp_vector_EX(exp_vector_EX) ,
157
158 .PCurrent_MEM(PC_MEM) ,. IR_MEM(inst_MEM) ,. ALUO_MEM(ALUout_MEM) ,.
159 Datao_MEM(Dataout_MEM) ,
160 .rd_MEM(rd_MEM) ,. rs1_MEM(rs1_MEM) ,. rs1_data_MEM(rs1_data_MEM) ,.
161 DatatoReg_MEM(DatatoReg_MEM) ,
162 .RegWrite_MEM(RegWrite_MEM) ,. WR_MEM(mem_w_MEM) ,. u_b_h_w_MEM(
163 u_b_h_w_MEM) ,. mem_r_MEM(mem_r_MEM) ,
164 .isFlushed(isFlushed_MEM) ,. csr_rw_MEM(csr_rw_MEM) ,.
165 csr_w_imm_mux_MEM(csr_w_imm_mux_MEM) ,
166 .mret_MEM(mret_MEM) ,. exp_vector_MEM(exp_vector_MEM)) ;
167
168
169 RAM_B data_ram (. addra(ALUout_MEM) ,. clka(debug_clk) ,. dina(Dataout_MEM) ,
170 . wea(mem_w_MEM) ,. rea(mem_r_MEM) ,. douta(RAMout_MEM) ,. mem_u_b_h_w(
171 u_b_h_w_MEM) ,
172 . l_access_fault(l_access_fault_MEM) ,. s_access_fault(
173 s_access_fault_MEM)) ;
174
175
176 ExceptionUnit exp_unit (. clk(debug_clk) ,. rst(rst) ,. csr_rw_in(csr_rw_MEM)
177 ,. csr_wsc_mode_in(inst_MEM[13:12]) ,

```

```

164     .csr_w_imm_mux(csr_w_imm_mux_MEMORY) ,.csr_rw_addr_in(inst_MEMORY[31:20])
165     ,
166     .csr_w_data_reg(rs1_data_MEMORY) ,.csr_w_data_imm(rs1_MEMORY) ,
167     .csr_r_data_out(CSRout_MEMORY) ,
168
169     .interrupt(interrupter) ,
170     .illegal_inst(~isFlushed_WB & exp_vector_WB[3]) ,
171     .ecall_m(~isFlushed_WB & exp_vector_WB[2]) ,
172     .l_access_fault(~isFlushed_WB & exp_vector_WB[1]) ,
173     .s_access_fault(~isFlushed_WB & exp_vector_WB[0]) ,
174     .mret(mret_MEMORY) ,
175     /*自行添加*/
176     .faultaddr(ALUout_WB) ,
177     .inst_is_zero(inst_WB ? 0 : 1) ,
178     /*自行添加*/
179     .epc_cur(PC_WB) ,
180     .epc_next(~isFlushed_MEMORY ? PC_MEMORY : ~isFlushed_EXE ? PC_EXE :
181     ~isFlushed_ID ? PC_ID : PC_IF) ,
182     .PC_redirect(PC_redirect_exp) ,.redirect_mux(redirect_mux_exp) ,
183     .reg_FD_flush(reg_FD_flush_exp) ,.reg_DE_flush(reg_DE_flush_exp) ,
184     .reg_EM_flush(reg_EM_flush_exp) ,.reg_MW_flush(reg_MW_flush_exp) ,
185     .RegWrite_cancel(RegWrite_cancel_exp)) ;
186
187 MUX2T1_32 mux_csrout (.I0(RAMout_MEMORY) ,.I1(CSRout_MEMORY) ,.s(csr_rw_MEMORY) ,.o
188 (Datain_MEMORY)) ;
189
190 // WB
191 REG_MEMORY_WB reg_MEMORY_WB(.clk(debug_clk) ,.rst(rst) ,.EN(1'b1) ,.flush(
192     reg_MW_flush_exp | isFlushed_MEMORY) ,
193     .IR_MEMORY(inst_MEMORY) ,.PCurrent_MEMORY(PC_MEMORY) ,.ALUO_MEMORY(ALUout_MEMORY) ,.
194     Datai(Datain_MEMORY) ,
195     .rd_MEMORY(rd_MEMORY) ,.DatatoReg_MEMORY(DatatoReg_MEMORY) ,.RegWrite_MEMORY(
196     RegWrite_MEMORY) ,
197     .exp_vector_MEMORY({exp_vector_MEMORY, l_access_fault_MEMORY ,
198         s_access_fault_MEMORY}) ,
199     .PCurrent_WB(PC_WB) ,.IR_WB(inst_WB) ,.ALUO_WB(ALUout_WB) ,.MDR_WB(
200         Datain_WB) ,
201     .rd_WB(rd_WB) ,.DatatoReg_WB(DatatoReg_WB) ,.RegWrite_WB(RegWrite_WB
202         ) ,
203     .isFlushed(isFlushed_WB) ,.exp_vector_WB(exp_vector_WB));

```

```

198
199 MUX2T1_32 mux_WB(. I0 (ALUout_WB) ,. I1 (Datain_WB) ,. s (DatatoReg_WB) ,. o (
200      wt_data_WB) );
201
202
203     wire [31:0] Test_signal;
204     assign debug_data = debug_addr[5] ? Test_signal : Debug_regs;
205
206     CPUTEST    U1_3(. PC_IF(PC_IF),
207                      .PC_ID(PC_ID),
208                      .PC_EXE(PC_EXE),
209                      .PC_MEM(PC_MEM),
210                      .PC_WB(PC_WB),
211                      .PC_next_IF(next_PC_IF),
212                      .PCJump(jump_PC_ID),
213                      .inst_IF(inst_IF),
214                      .inst_ID(inst_ID),
215                      .inst_EXE(inst_EXE),
216                      .inst_MEM(inst_MEM),
217                      .inst_WB(inst_WB),
218                      .PCEN(PC_EN_IF),
219                      .Branch( Branch_ctrl),
220                      .PCSource( Branch_ctrl),
221                      .RS1DATA(rs1_data_reg),
222                      .RS2DATA(rs2_data_reg),
223                      .Imm32(Imm_out_ID),
224                      .ImmSel(ImmSel_ctrl),
225                      .ALUC(ALUControl_ctrl),
226                      .ALUSrc_A(ALUSrc_A_ctrl),
227                      .ALUSrc_B(ALUSrc_B_ctrl),
228                      .A(ALUA_EXE),
229                      .B(ALUB_EXE),
230                      .ALU_out(ALUout_MEM),
231                      .Datai(Datain_MEM),
232                      .Datao(Dataout_MEM),
233                      .Addr(Addr),
234                      .WR(MWR),
235                      .MIO(mem_r_MEM),
236                      .WDATA(wt_data_WB),
237                      .DatatoReg(DatatoReg_WB),
238                      .RegWrite(RegWrite_WB),
239                      .data_hazard(reg_FD_stall),

```

```
239     .control_hazard(Branch_ctrl),
240     .exp_sig({csr_rw_MEM, 2'b0, csr_w_imm_mux_MEM,
241               isFlushed_ID, isFlushed_EXE, isFlushed_MEM,
242               isFlushed_WB,
243               interrupter, 3'b0, exp_vector_WB,
244               4'b0,
245               3'b0, redirect_mux_exp,
246               reg_FD_flush_exp, reg_DE_flush_exp, reg_EM_flush_exp
247               , reg_MW_flush_exp,
248               3'b0, RegWrite_cancel_exp}),
249     .Debug_addr(debug_addr[4:0]),
250     .Test_signal(Test_signal)
251   );
252
253 endmodule
```

## 4 实验步骤与调试

### 4.1 仿真

根据已经写好的代码，进行仿真模拟

#### 4.1.1 异常处理

当程序遇到 exception 时，跳入处理程序。程序在 WB 阶段接收 exception 信号，并跳入异常处理程序，将还在执行的命令全部清空，并取消寄存器的写入。

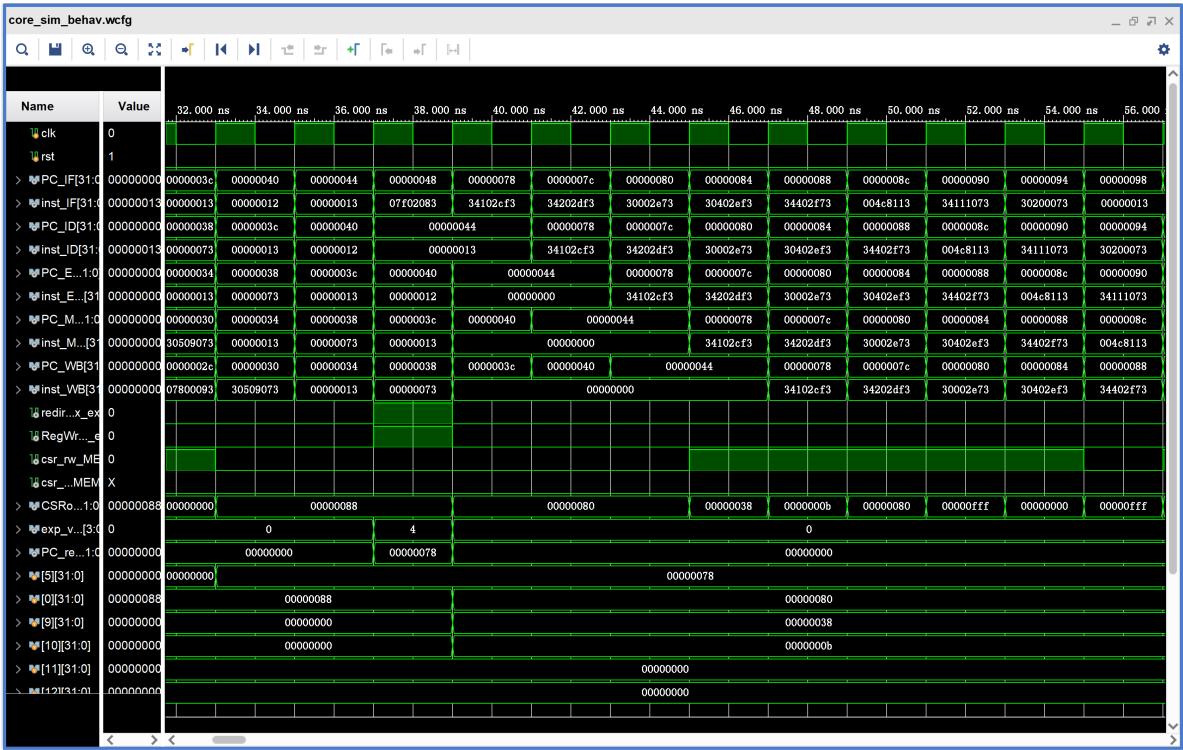


图 4.11: 仿真结果图 1

当异常处理程序运行至末尾时，程序读取到 mret 指令并回到原程序中，由于在异常处理程序中给 mepc 加上了 4，实际回到的是异常发生指令的下一条指令。

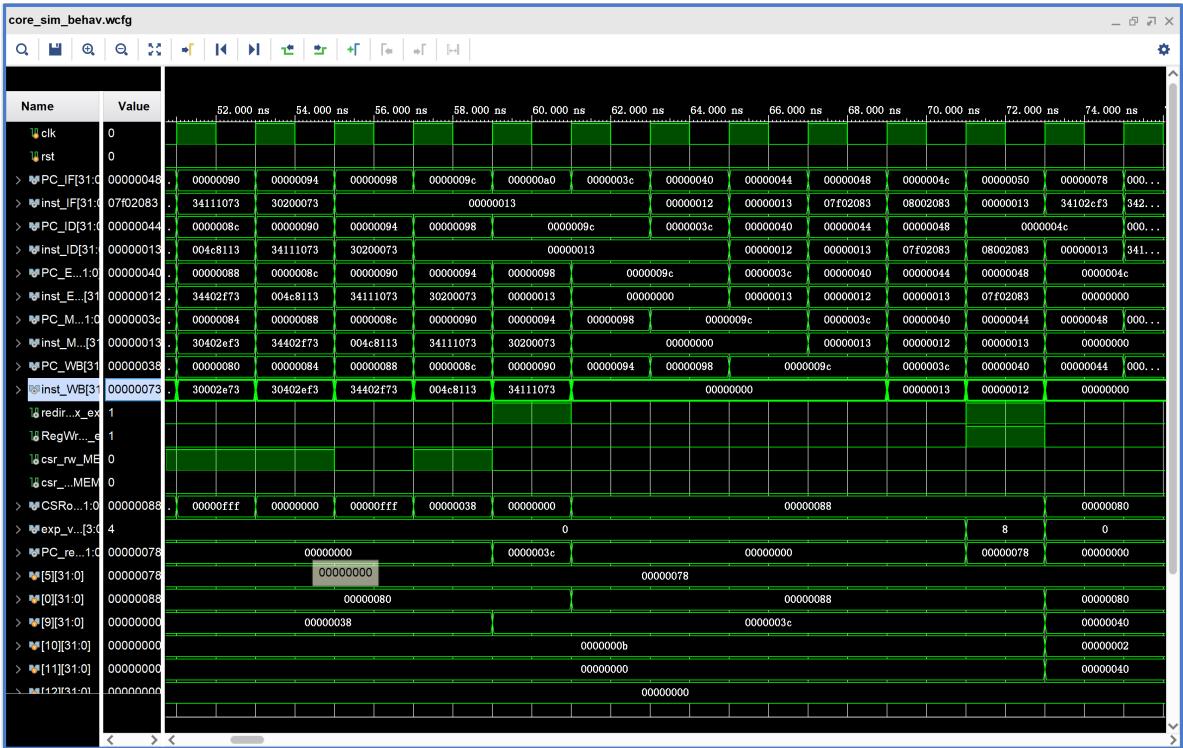


图 4.12: 仿真结果图 2

之后程序继续运行直到遇到下一个异常，并以相同的方式处理该异常。

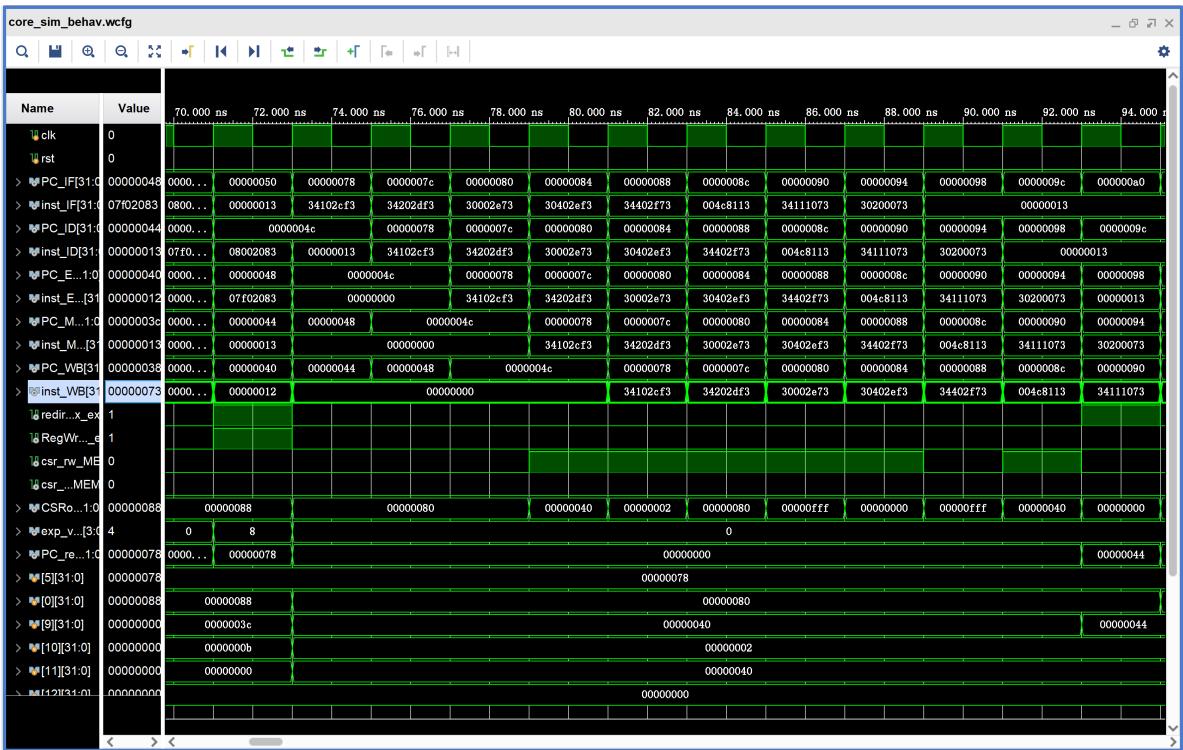


图 4.13: 仿真结果图 3

### 4.1.2 中断处理

当程序遇到外部中断时，进入中断处理程序。在中断过程中，虽然中断信号一直拉起，但是由于 mstatus 的 mie 位此时为零，因此无法在中断中触发中断。

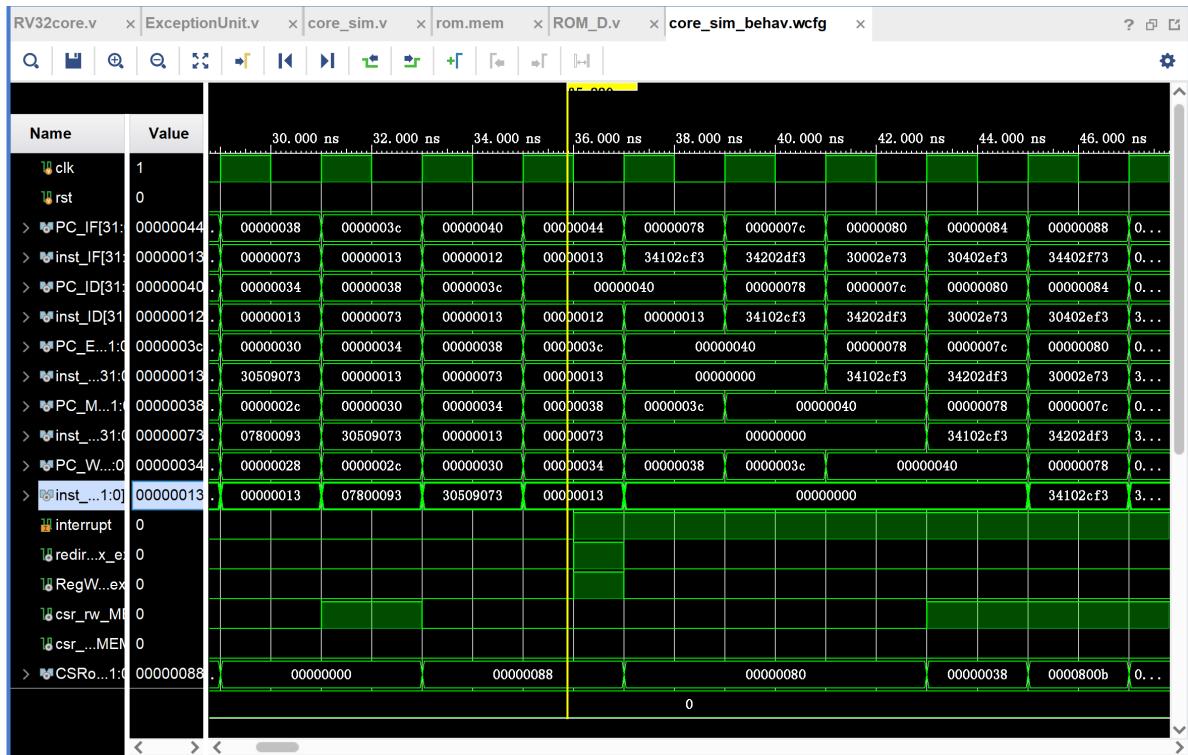


图 4.14: 仿真结果图 4

返回时回到需要执行的下一条指令

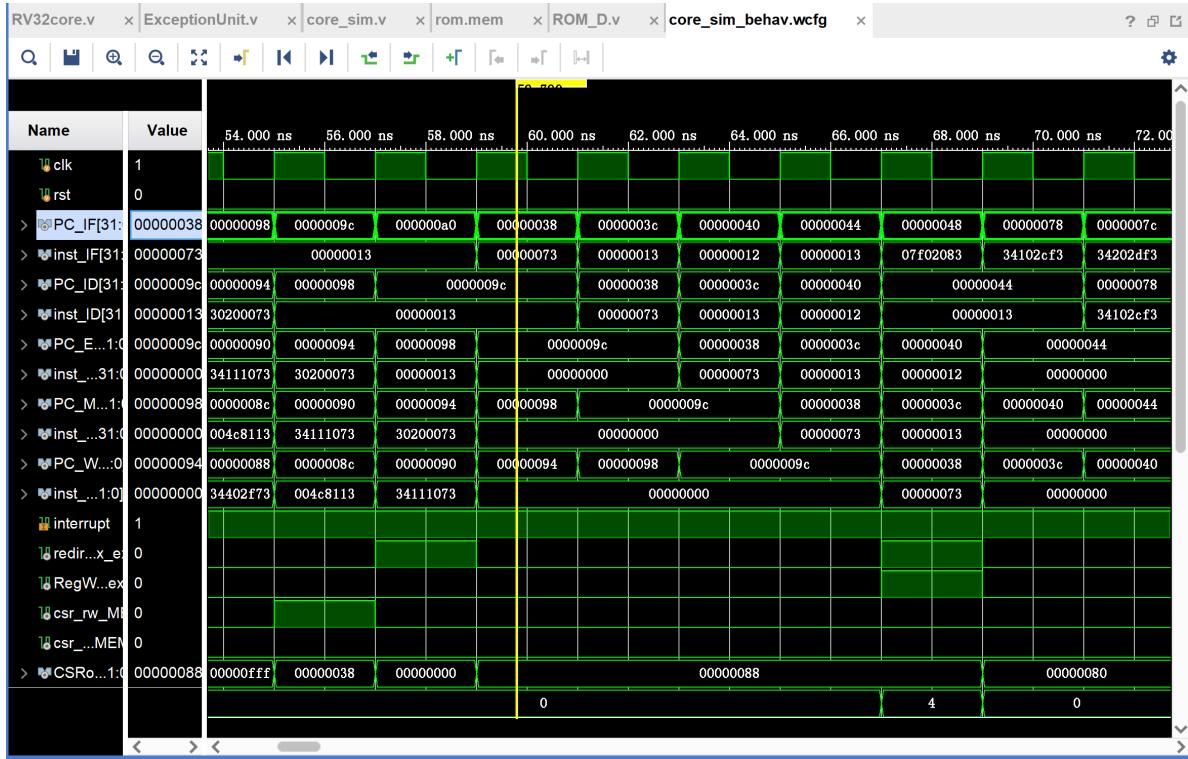


图 4.15: 仿真结果图 5

返回后中断信号依旧处于拉起状态，在程序完全离开中断处理程序后，再次进入中断处理程序。

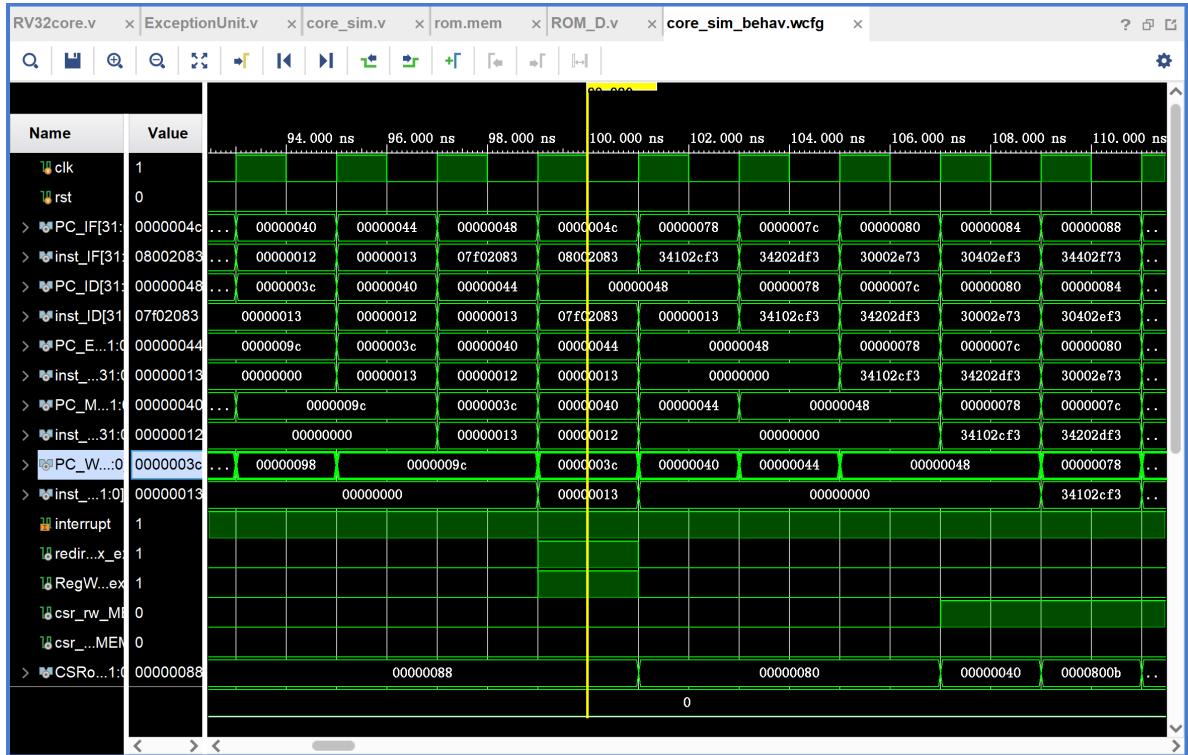


图 4.16: 仿真结果图 6

## 4.2 综合

选择左侧面板的 Run Synthesis 或者点击上方的绿色小三角，选择 Synthesis

## 4.3 实现

选择左侧面板的 Run Implementation 或者点击上方的绿色小三角，选择 Implementation。值得注意的是执行 implementation 之前应该确保引脚约束存在且正确，同时之前已经综合过最新的代码。

## 4.4 验证设计

选择左侧面板的 Open Elaborated Design，输出的结果如下，根据原理图来判断，基本没有问题

## 4.5 生成二进制文件

选择左侧面板的 Generate Bitstream 或者点击上方的绿色二进制标志。同时生成 Bitstream 前要确保：之前已经综合、实现过最新的代码。如没有，直接运行会默认从综合、

实现开始。此过程还要注意生成的 bit 文件默认存放在.runs 下相应的 implementation 文件夹中

## 4.6 烧写上板

点击左侧的 Open Hardware Manager → 点击 Open Target → Auto Connect → 点击 Program Device → 选择 bistream 路径，烧写。验证结果见实验结果部分。

# 5 实验结果与分析

经过综合实现，我们可以在开发板上看到测试的结果。我们分以下几个部分验证实验结果的正确性

## 5.1 异常与中断跳转

当触发异常或中断时，清空尚未结束的所有指令，并跳转至中断/异常处理程序  
遇到 ecall 指令触发异常

Zhejiang University Computer Organization Experimental  
SOC Test Environment (With RISC-V)

	x0:zero 00000000	x01: ra 00000078	x02: sp 00000000	x03: gp 00000000
x04: tp	00000010	x05: t0 00000014	x06: t1 FFFF0000	x07: t2 0FFF0000
x08:fps0	00000000	x09: s1 00000000	x10: a0 00000000	x11: a1 00000000
x12: a2	00000000	x13: a3 00000000	x14: a4 00000000	x15: a5 00000000
x16: a6	00000000	x17: a7 00000000	x18: s2 00000000	x19: s3 00000000
x20: s4	00000000	x21: s5 00000000	x22: s6 00000000	x23: s7 00000000
x24: s8	00000000	x25: s9 00000000	x26:s10 00000000	x27:s11 00000000
x28: t3	00000000	x29: t4 00000000	x30: t5 00000000	x31: t6 00000000
PC---IF	00000048	INST- IF 07P02863	rs1Data 00000000	rs2Data 00000000
PC---ID	88888844	INST-ID 00000013	rs1addr 00000000	rs2addr 00000000
PC---EXE	00000048	INST-Ex 00000012	Exp-Sig 000401F1	PCJump 00000044
PC---MEM	0000003C	INST-M 00000013	B/PCE-S 00000100	PCOut 00000000
PC---WB	00000038	INST-WB 00000073	LAbSel 00010001	PCIPExt 0000004C
ALU-Ain	00000000	ALU-Out 00000000	CPUAddr 00000000	ALU-S1 00000001
ALU-Bin	00000000	WB-Data 00000000	CPU-Dai FFFFFFFBF	WB-D1 00000000
Imm-32ID	00000000	WB-Addr 00000000	CPU-Dao 00000000	RegA/DR 00000000
CODE-00	00000000	lw x01,x00,07FH		CODE-03 00000000
CODE-04	00000000	nop JStall:addi0		CODE-07 00000000
CODE-06	00000000	addi x00,x00,000H		CODE-08 00000000
CODE-0C	00000000	nop JStall:addi0		CODE-0F 00000000
CODE-18	00000000		ecall	CODE-13 00000000
CODE-14	00000000	CODE-15 00000000	CODE-16 00000000	CODE-17 00000000
CODE-18	00000000	CODE-19 00000000	CODE-1A 00000000	CODE-1B 00000000
CODE-1C	00000000	CODE-1D 00000000	CODE-1E 00000000	CODE-1F 00000000
CODE-20	00000000	CODE-21 00000000	CODE-22 00000000	CODE-23 00000000
CODE-24	00000000	CODE-25 00000000	CODE-26 00000000	CODE-27 00000000

POWER VOL- VOL+ MENU CH- CH+ MODE

图 5.17: 验证结果图 1

进入 trap 时清空先前指令

```
Zhejiang University Computer Organization Experimental  
SOC Test Environment (With RISC-V)  
x0:zero 00000000    x01: ra 00000078    x02: sp 00000008    x03: gp 00000000  
x04: tp 00000010    x05: t0 00000014    x06: t1 FFFF0000    x07: t2 0FFF0000  
x8:fps0 00000000    x09: s1 00000000    x10: a0 00000000    x11: a1 00000000  
x12: a2 00000000    x13: a3 00000000    x14: a4 00000000    x15: a5 00000000  
x16: a6 00000000    x17: a7 00000000    x18: s2 00000000    x19: s3 00000000  
x20: s4 00000000    x21: s5 00000000    x22: s6 00000000    x23: s7 00000000  
x24: s8 00000000    x25: s9 00000000    x26:s10 00000000    x27:s11 00000000  
x28: t3 00000000    x29: t4 00000000    x30: t5 00000000    x31: t6 00000000  
PC-- IF 00000078 INST IF 34182CF3 rs1Data 00000000 rs2Data 00000000  
PC-- ID 00000041 INST ID 00000013 rs1Addr 00000000 rs2Addr 00000000  
PC-- EXE 00000041 INST EX 00000000 Exp-Sig 00000000 PCLength 00000044  
PC-- MDM 00000048 INST M 00000000 B/PCE-S 00000100 b1/b2 00000000  
PC-- WB 0000003C INST WB 00000000 L/ABSel 00010001 PCLength 0000007C  
ALU-Ain 00000000 ALU-Din 00000000 CPUAddr 00000000 ALU-S1 00000001  
ALU-Bin 00000000 WB-Data 00000000 CPU-Dai FFFFFFFDFD WB-WB 00000000  
Imm32ID 00000000 WB-Addr 00000000 CPU-Dlo 00000000 RegM/DR 00000000  
CODE-00 00000000 srs x19, 1,x00 CODE-03 00000000  
CODE-04 00000000    nop JStall:addi0 CODE-07 00000000  
CODE-08 00000000    nop DStall:lw 00 CODE-0B 00000000  
CODE-0C 00000000    nop DStall:lw 00 CODE-0F 00000000  
CODE-10 00000000    nop DStall:lw 00 CODE-13 00000000  
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000  
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000  
CODE-1C 00000000 CODE-1D 00000000 CODE-1E 00000000 CODE-1F 00000000  
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000  
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000
```

图 5.18: 验证结果图 2

trap 中指令的运行

暂停

```
Zhejiang University Computer Organization Experimental  
SOC Test Environment (With RISC-V)

x0:zero 00000000 x01: ra 00000078 x02: sp 00000004 x03: gp 00000000
x04: tp 00000010 x05: t0 00000014 x06: t1 FFFF0000 x07: t2 0FFF0000
x08:fps0 00000000 x09: s1 00000000 x10: a0 00000000 x11: a1 00000000
x12: a2 00000000 x13: a3 00000000 x14: a4 00000000 x15: a5 00000000
x16: a6 00000000 x17: a7 00000000 x18: s2 00000000 x19: s3 00000000
x20: s4 00000000 x21: s5 00000000 x22: s6 00000000 x23: s7 00000000
x24: s8 00000000 x25: s9 00000000 x26: s10 00000000 x27: s11 00000000
x28: t3 00000000 x29: t4 00000000 x30: t5 00000000 x31: t6 00000000
PC---IF 00000038 INST-IF 344B2F73 rs1Data 00000000 rs2Data 00000010
PC---ID 00000034 INST-ID 304B2EF3 rs1Addr 00000000 rs2Addr 00000004
PC---EXE 00000080 INST-EX 30002E73 Exp-Sig 00000000 PCLength 00000004
PC---MEM 0000007C INST-M 342B2DF3 B/PCE-S 00000100 B/PCE-R 00000000
PC---WB 00000078 INST-WB 34102CF3 I/ADSel 00000000 WBENM 0000000C
ALU-Ain 00000000 ALU-Out 00000000 CPUAddr 00000000 Q1/Q2/Q3 00000000
ALU-Bin 00000000 WB-Data 00000000 CPU-Dai 00000000 WR-B10 00000000
Im32ID 00000000 WB-Addr 00000019 CPU-Dao 00000000 RegW/DR 00000001
CODE-00 00000000 srrs x1E, 4,x00 CODE-03 00000000
CODE-04 00000000 srrs x1D, 4,x00 CODE-07 00000000
CODE-08 00000000 srrs x1C, 0,x00 CODE-0B 00000000
CODE-0C 00000000 srrs x1B, 2,x00 CODE-0F 00000000
CODE-10 00000000 srrs x19, 1,x00 CODE-13 00000000
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000
CODE-1C 00000000 CODE-1D 00000000 CODE-1E 00000000 CODE-1F 00000000
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000
```

POWER VOL- VOL+ MENU CH- CH+ MODE

图 5.19: 验证结果图 3

运行至 mret 后，清空后面的所有指令并跳转回原程序

暂停

```
Zhejiang University Computer Organization Experimental  
SOC Test Environment (With RISC-V)

x0:zero 00000000 x01: ra 00000078 x02: sp 00000004 x03: gp 00000000
x04: tp 00000010 x05: t0 00000014 x06: t1 FFFF0000 x07: t2 0FFF0000
x08:fps0 00000000 x09: s1 00000000 x10: a0 00000000 x11: a1 00000000
x12: a2 00000000 x13: a3 00000000 x14: a4 00000000 x15: a5 00000000
x16: a6 00000000 x17: a7 00000000 x18: s2 00000000 x19: s3 00000000
x20: s4 00000000 x21: s5 00000000 x22: s6 00000000 x23: s7 00000000
x24: s8 00000000 x25: s9 00000000 x26: s10 00000000 x27: s11 00000000
x28: t3 00000000 x29: t4 00000FFF x30: t5 0000000B x31: t6 00000000
PC---IF 00000044 INST-IF 00000013 rs1Data 00000000 rs2Data 00000000
PC---ID 00000040 INST-ID 00000012 rs1Addr 00000000 rs2Addr 00000000
PC---EXE 0000003C INST-EX 00000013 Exp-Sig 00000000 PCLength 00000004
PC---MEM 000000A8 INST-M 00000013 B/PCE-S 00000100 B/PCE-R 00000000
PC---WB 0000009C INST-WB 00000013 I/ADSel 00000000 WBENM 00000048
ALU-Ain 00000000 ALU-Out 00000000 CPUAddr 00000000 Q1/Q2/Q3 00000000
ALU-Bin 00000000 WB-Data 00000000 CPU-Dai FFFFFFFF WR-B10 00000000
Im32ID 00000000 WB-Addr 00000000 CPU-Dao 00000000 RegW/DR 00000000
CODE-00 00000000 nop JStall:addi0 CODE-03 00000000
CODE-04 00000000 addi x00, x00, 00001 CODE-07 00000000
CODE-08 00000000 nop JStall:addi0 CODE-0B 00000000
CODE-0C 00000000 nop JStall:addi0 CODE-0F 00000000
CODE-10 00000000 nop JStall:addi0 CODE-13 00000000
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000
CODE-1C 00000000 CODE-1D 00000000 CODE-1E 00000000 CODE-1F 00000000
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000
```

POWER VOL- VOL+ MENU CH- CH+ MODE

图 5.20: 验证结果图 4

## 6 讨论与心得

本实验要求在上一个实验的基础上实现中断与异常的相关功能。在本次实验中，我们再次复习了上学期与中断部分相关的内容，并学会了如何在使用流水线的情况下进行中断与异常的相关处理。在实验过程中，我们也遇到了一些问题，如中断与异常时处理 mepc 到底有哪些区别，中断长开时应该如何处理。许多内容在课上都没有提及，我们在查阅了相关资料后得到了答案。