浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验名称：Lab2:Pipelined CPU supporting exception & interrupt

指导教师：何水兵 完成日期：2023 年 11 月 7 日

实验分工

姓名：蒋奕 3210103803

同组学生姓名：黄俊涛 3210101831

分工情况：

蒋奕:ExceptionUnit 模块 + 仿真设计状态机进行 STATE_IDLE,STATE_MEPC,STATE_MCAUSE 之间的转换（时钟正边沿），进行 epc 和 mcause 寄存器的记录，生成 reg 的 flush 信号（时钟负边沿）。最后仿真，验证并且得到结果。

黄俊涛：ExceptionUnit 模块 + 仿真

1 实验目的和要求

- Understand the principle of CPU exception & interrupt and its processing procedure.
- Master the design methods of pipelined CPU supporting exception & interrupt.
- master methods of program verification of Pipelined CPU supporting exception & interrupt.

2 实验内容和原理

2.1 实验内容

- Design of Pipelined CPU supporting exception & interrupt including:

  1. Design datapath

  2. Design Co-processor & Controller

- Verify the Pipelined CPU with program and observe the execution of program

2.2 实验原理

- CSR instruction

  In this experiment, the following CSR instructions need to be supported The following is a representation of the input and output signals for a given CSRRegs module:

csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		uimm	101	rd	1110011	CSRRWI
csr		uimm	110	rd	1110011	CSRRSI
csr		uimm	111	rd	1110011	CSRRCI
000000000000		00000	000	00000	1110011	ECALL
0011000	00010	00000	000	00000	1110011	MRET

表 1: CSRRegs

Signal Name	**Signal Type**	**Bit Width**	**Description**
clk	Input	-	Clock signal
rst	Input	-	Reset signal
raddr	Input	12 bits	Read address
waddr	Input	12 bits	Write address
wdata	Input	32 bits	Write data
csr_w	Input	-	Control signal indicating CSR write operation
csr_wsc_mode	Input	2 bits	Mode of CSR write operation
rdata	Output	32 bits	Read data
mstatus	Output	32 bits	Value of the M-status register
mepc	Input	32 bits	Machine Exception Program Counter
mcause	Input	32 bits	Machine Exception Cause Register

- exception detection

  The top-level data path has already passed in the exception detection signal to ExceptionUnit. It only needs to judge the exception based on the input signal. Then go to the exception handling part. The following are the input signals for exception detection:

表 2: Signal Descriptions

Signal Name	**Description**
input illegal_inst	Illegal instruction
input l_access_fault	Load instruction access fault
input s_access_fault	Store instruction access fault
input ecall_m	Environment call (ecall)

- Exception handling

  We need to handle exceptions when processing request:

  1. Access error exception: The physical memory address does not support access
  2. Environment call exception: Occurs when executing the ecall instruction
  3. Illegal instruction exception: Invalid opcode found during decoding phase

  When an exception/interruption occurs, the hardware status transition is as follows:

1. The PC of the exception instruction is saved in `mepc`, and the PC is set to `mtvec`. `mepc` points to the instruction that caused the exception; for interrupts, it points to the location where execution should resume after the interrupt is handled.
2. Set `mcause` according to the source of the exception, and set `mtval` to the address of the error or other information words applicable to the specific exception.
3. Set the `MIE` bit in the control status register `mstatus` to zero to disable interrupts and retain the previous `MIE` value in `MPIE`.
4. The permission mode before the exception occurs is retained in the `MPP` domain of `mstatus`, and then the permission mode is changed to `M`.

表 3: Address, Name, and Description for regs

Address	Name	Description
0x300	mstatus	Machine status register
0x305	mtvec	Machine trap-handler base address
0x341	mepc	Machine exception program counter
0x342	mcause	Machine trap cause
0x343	mtval	Machine bad address or instruction

Exception Handling Steps: When an exception occurs:

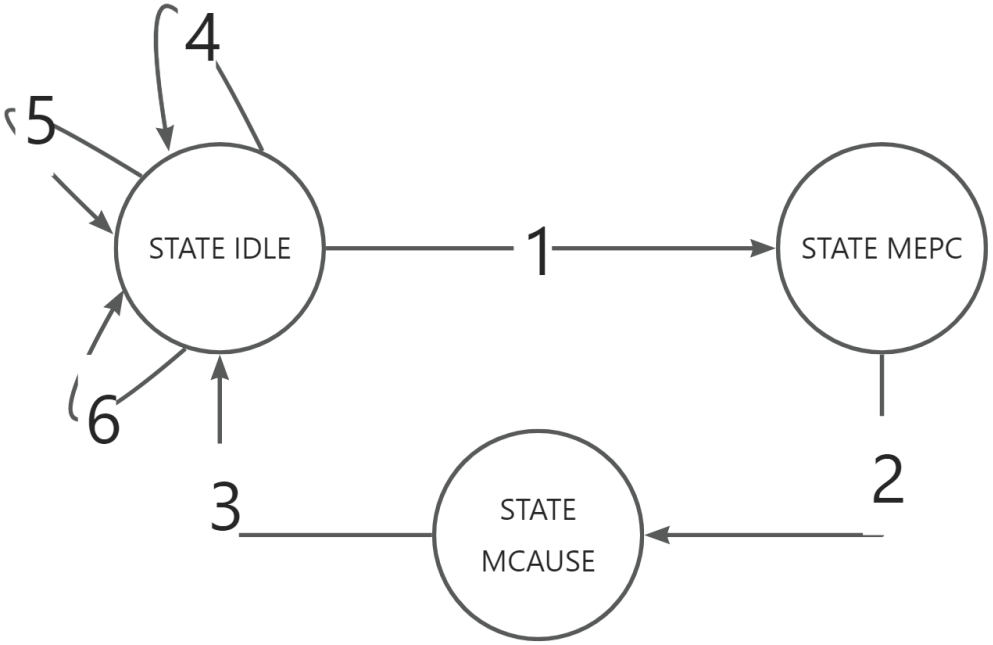
1. `mstatus`: Set `MIE` to 0 to disable interrupts and save the original `MIE` value in `MPIE`.
2. `mtvec`: Set the current PC as `mtvec`.
3. `mepc`: Save the address of the instruction that caused the exception.
4. `mcause`: For exceptions, set the highest bit to 0, and the lower bits as follows:
   - Illegal instruction exception: Set the lower bits to 2.
   - Load access exception: Set the lower bits to 5.
   - Store access exception: Set the lower bits to 7.
   - Ecall environment call exception: Set the lower bits to 11.
5. `mtval`: - For access exceptions: Store the address that caused the error.
   - For an illegal instruction exception: Store the illegal instruction itself. - For other exceptions: Set to 0.

After handling the exception: When detecting the `mret` instruction, perform the following steps:

1. Set the PC register to `mepc` (which now holds the address after incrementing by 4, not the exception instruction address).
2. Copy the `MPIE` field from `mstatus` to the `MIE` field to restore the previous interrupt enable settings (this is the reverse operation of what was done when the exception occurred).

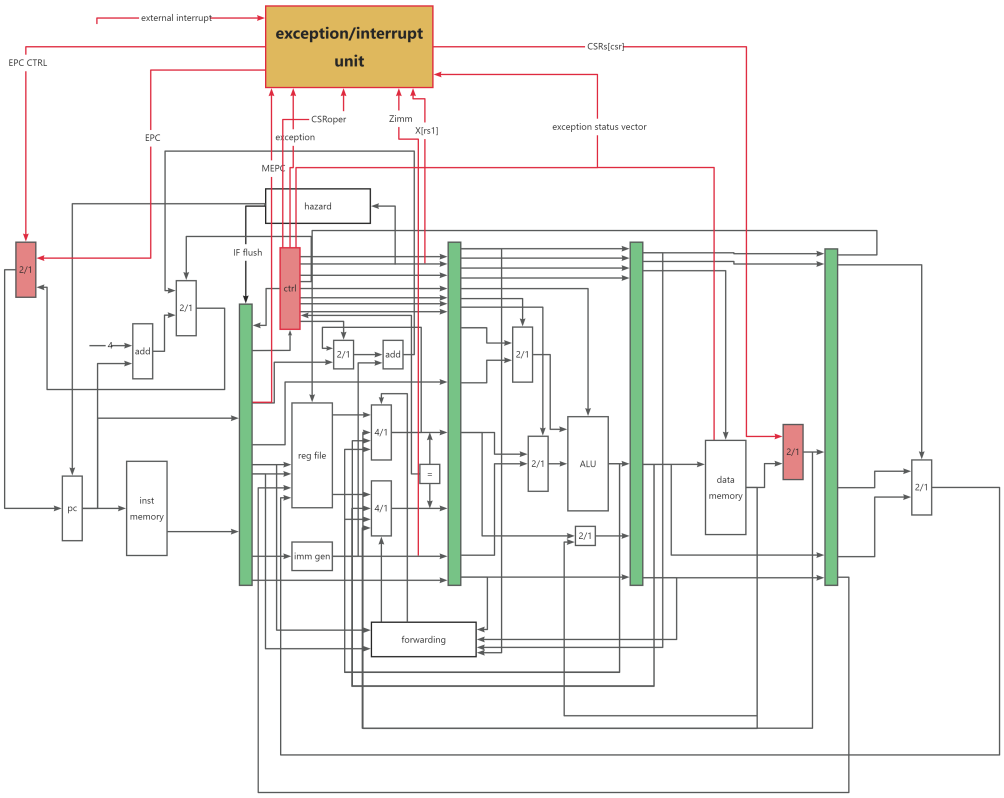
In all, we use state machine to represent this:

Exception Handling and Instruction Flow:



1. **STATE_IDLE → (exception or interruption) STATE_MEPC**
   - Write to `mstatus`.
   - Flush all pipeline registers.
   - If it's an exception (not an interrupt), cancel register writes.
   - Record `epc` and `cause`.
2. **STATE_MEPC → STATE_MCAUSE**
   - Write `epc` to `mepc`.
   - Read `mtvec`.
   - Set the redirect PC mux (next cycle PC `mtvec`).
3. **STATE_MCAUSE → STATE_IDLE**
   - Write `cause` to `mcause`.
4. **STATE_IDLE → (mret) STATE_IDLE**
   - Write to `mstatus`.
   - Read `mepc`.
   - Set the redirect PC mux (next cycle PC `mepc`).
   - Flush pipeline registers (EM, DE, FD).
5. **STATE_IDLE → (CSR instructions) STATE_IDLE**
   - Perform CSR (Control and Status Register) operations.
6. **STATE_IDLE → (other) STATE_IDLE**

The schematic is as follows:



3 实验过程和数据记录及结果分析

3.1 实验过程

The details of exception unit can be seen in comments of the corresponding code.

- Complete the ExceptionUnit.v file

```
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
19      input mret,
20
21      input[31:0] epc_cur,
22      input[31:0] epc_next,
23      output[31:0] PC_redirect,
24      output redirect_mux,
25
26      output reg_FD_flush, reg_DE_flush, reg_EM_flush, reg_MW_flush,
27      output RegWrite_cancel
28  );
29
30      reg[11:0] csr_waddr = 0;
31      wire[11:0] csr_raddr;
32      reg[31:0] csr_wdata = 0;
33      reg csr_w = 0;// csr write enable
34      reg[1:0] csr_wsc = 0;
35      parameter CSR_WRITE_MODE = 2'b01;
36      parameter CSR_SET_MODE = 2'b10;
37      parameter CSR_CLEAR_MODE = 2'b11;
38      parameter WRITE_ENABLE = 1'b1;
39      parameter WRITE_DISABLE = 1'b0;
40      wire[31:0] mstatus;
41      parameter IS_CSR_INSTRUCTIONS = 1'b1;
42      parameter MSTATUS_ADDR = 12'h300;
43      parameter MTVEC_ADDR = 12'h305;
44      parameter MEPC_ADDR = 12'h341;
45      parameter MCAUSE_ADDR = 12'h342;
46      parameter MTVAL_ADDR = 12'h343;
47      parameter MIP_ADDR = 12'h344;
48      parameter STATE_IDLE = 2'b00;
49      parameter STATE_MEPC = 2'b01;
50      parameter STATE_MCAUSE = 2'b10;
51      reg[1:0] currentState = STATE_IDLE;
52      CSRRegs csr(.clk(clk),.rst(rst),.csr_w(csr_w),.raddr(csr_raddr)
    ↪  ,.waddr(csr_waddr),
53          .wdata(csr_wdata),.rdata(csr_r_data_out),.mstatus(mstatus),
    ↪  .csr_wsc_mode(csr_wsc));
54      reg[31:0] mcauseRecorder = 0;
55      reg[31:0] epcRecorder = 0;
```

```
56      // do things when negedge
57      always @(negedge clk) begin
58          case(currentState)
59          STATE_IDLE: begin
60          // write mstatus
61          // flush all the pipeline registers
62          // if exception (not interrupt), cancal regwrite
63          // record epc and cause
64              //exception or interrupt
65              if((mstatus[3] && interrupt) | illegal_inst |
    ↪ l_access_fault | s_access_fault | ecall_m) begin
66                  //write mstatus
67                  csr_waddr = MSTATUS_ADDR;
68                  csr_wdata = {mstatus[31:8], mstatus[3],
    ↪ mstatus[6:4], 1'b0, mstatus[2:0]};
69                  csr_w = WRITE_ENABLE;
70                  csr_wsc = CSR_WRITE_MODE;
71                  //record epc & mcause
72                  epcRecorder = epc_cur;
73                  if(interrupt) begin
74                      mcauseRecorder = 1 << 31;
75                  end
76                  else if(illegal_inst) begin
77                      mcauseRecorder = 2;
78                  end
79                  else if(l_access_fault) begin
80                      mcauseRecorder = 5;
81                  end
82                  else if(s_access_fault) begin
83                      mcauseRecorder = 7;
84                  end
85                  else if(ecall_m) begin
86                      mcauseRecorder = 11;
87                  end
88                  else begin
89                      mcauseRecorder = 0;
90                  end
91              end
92              //mret
93              else if(mret) begin
94                  // write mstatus
95                  // read mepc
```

```
96                  // set redirect pc mux (next cycle pc -> mepc)
97                  // flush pipeline registers (EM, DE, FD)
98                  csr_waddr = MSTATUS_ADDR;
99                  csr_wdata = {mstatus[31:8], mstatus[3],
    ↪ mstatus[6:4], 1'b0, mstatus[2:0]};
100                 csr_w = WRITE_ENABLE;
101                 csr_wsc = CSR_WRITE_MODE;
102             end
103             //csr instructions
104             else if(csr_rw_in == IS_CSR_INSTRUCTIONS) begin
105                 //csr operations
106                 csr_waddr = csr_rw_addr_in;
107                 if (csr_w_imm_mux) begin
108                     // write immediate number
109                     csr_wdata = {27'b0, csr_w_data_imm};
110                 end
111                 else begin
112                     // write reg data
113                     csr_wdata = csr_w_data_reg;
114                 end
115                 csr_w = csr_rw_in;
116                 csr_wsc = csr_wsc_mode_in;
117             end
118             //other
119             else begin
120                 csr_w = WRITE_DISABLE;
121             end
122         end
123
124         STATE_MEPC: begin
125             // write epc to mepc
126             // read mtvec
127             // set redirect pc mux (next cycle pc -> mtvec)
128             // assign redirect_mux = (((currentState ==
    ↪ STATE_IDLE)) & mret)|(currentState == STATE_MEPC);
129             csr_waddr = MEPC_ADDR;
130             csr_wdata = epcRecorder;
131             csr_w = WRITE_ENABLE;
132             csr_wsc = CSR_WRITE_MODE;
133         end
134
135         STATE_MCAUSE: begin
```

```
136             // write cause to mcause
137             csr_waddr = MCAUSE_ADDR;
138             csr_wdata = mcauseRecorder;
139             csr_w = WRITE_ENABLE;
140             csr_wsc = CSR_WRITE_MODE;
141         end
142         endcase
143     end
144     //state change when posedge
145     always @(posedge clk) begin
146         case(currentState)
147         STATE_IDLE: begin
148             // exception or interrupt
149             if((mstatus[3] && interrupt) | illegal_inst |
    ↪ l_access_fault | s_access_fault | ecall_m) begin
150                 currentState <= STATE_MEPC;
151             end
152             // mret
153             else if(mret) begin
154                 currentState <= STATE_IDLE;
155             end
156             // csr instructions
157             else if(csr_rw_in) begin
158                 currentState <= STATE_IDLE;
159             end
160             // other
161             else begin
162                 currentState <= STATE_IDLE;
163             end
164         end
165         STATE_MEPC: begin
166             currentState <= STATE_MCAUSE;
167         end
168         STATE_MCAUSE: begin
169             currentState <= STATE_IDLE;
170         end
171         endcase
172     end
173
174     assign csr_raddr = ((currentState == STATE_IDLE)) ?
175                         (((mstatus[3] && interrupt) | illegal_inst
    ↪ | l_access_fault | s_access_fault | ecall_m) ? //exception or
    ↪ interrupt
```

```
176                         0 : mret ? MEPC_ADDR // read mepc in
    ↪ STATE_IDLE -> (mret) STATE_IDLE
177                         : csr_rw_in ? csr_rw_addr_in : 0 ) : // csr
    ↪ instructions read address of csr regs
178                         ((currentState == STATE_MEPC) ? MTVEC_ADDR
    ↪ : 0); // read mtvec in STATE_MEPC -> STATE_MCAUSE
179     assign PC_redirect = csr_r_data_out;
180     // STATE_IDLE -> (mret) STATE_IDLE ==> set redirect pc mux
    ↪ (next cycle pc -> mepc)
181     // STATE_MEPC -> STATE_MCAUSE  ==> set redirect pc mux (next
    ↪ cycle pc -> mtvec)
182     assign redirect_mux = (((currentState == STATE_IDLE)) && mret)
    ↪ |
183                             (currentState == STATE_MEPC);
184     // flush all the pipeline registers when STATE_IDLE -->
    ↪ STATE_MEPC
185     assign reg_FD_flush = reg_EM_flush;
186     assign reg_DE_flush = reg_FD_flush;
187     assign reg_EM_flush = reg_MW_flush ||
188                         (((currentState == STATE_IDLE)) && mret);
189     // flush pipeline registers (EM, DE, FD) when STATE_IDLE -->
    ↪ STATE_IDLE (mret)
190     assign reg_MW_flush = (((currentState == STATE_IDLE)) &&
    ↪ ((mstatus[3] && interrupt) |
191                              illegal_inst |
192                              l_access_fault |
193                              s_access_fault |
194                              ecall_m)) ||
195                             (currentState == STATE_MEPC);
196     // if exception (not interrupt), cancal regwrite when
    ↪ STATE_IDLE --> STATE_IDLE (exception or interrupt)
197     assign RegWrite_cancel = ((currentState == STATE_IDLE)) &&
198                             (illegal_inst |
199                             l_access_fault |
200                             s_access_fault |
201                             ecall_m);
202 endmodule
```

3.2 数据记录及结果分析

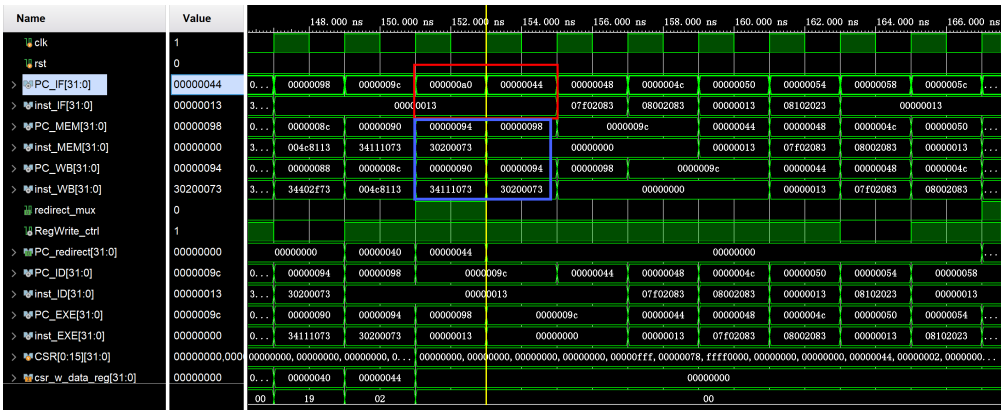
Since there are many cases in simulation wave, I only take some examples to show.For the convenience of explanation, all my explanations below are for the positions corresponding to the **yellow vertical lines** in the simulation diagram.

- illegal instruction

The related code block is as follows:

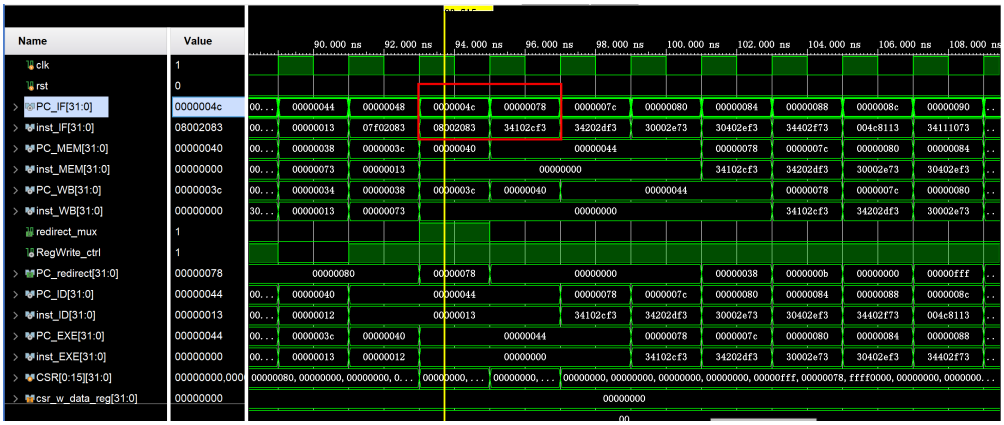
PC_IF	Instruction	Function
0x40	addi x0, x0, 0	0x12 is illegal decode of instruction
0x78	csrr x25, 0x341	label is trap

PC_IF jump from 0x40 to 0x78. When we decode the instruction, we find 0x00000012 for addi x0,x0,0 is illegal instruction, then write mstatus , flush all the pipeline registers, cancal regwrite, record epc = current epc and cause = 2.



When we finish handling illegal instruction, we execute mret to return to the instruction just after the illegal address because the trap program does.

- load access fault



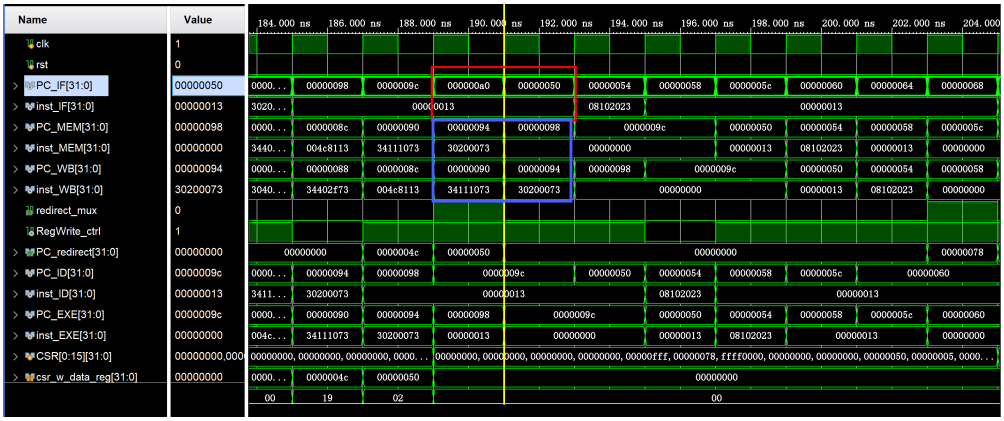
The related code block is as follows:

PC_IF	Instruction	Function
0x4C	lw x1, 128(x0)	l access fault
0x78	csrr x25, 0x341	label is trap

While the RAM is as follows:

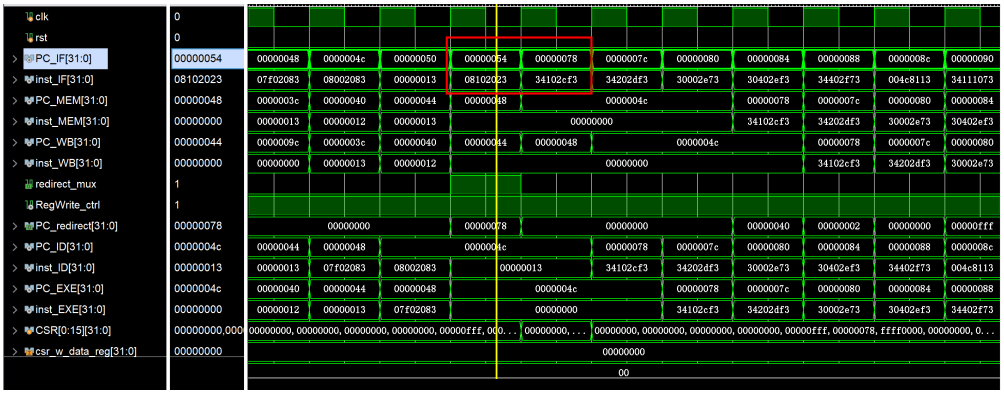
NO.	Data	Addr.	Comment	NO.	Data	Addr.	Comment
0	000080BF	0		16	00000000	40	
1	00000008	4		17	00000000	44	
2	00000010	8		18	00000000	48	
3	00000014	C		19	00000000	4C	
4	FFFF0000	10		20	A3000000	50	
5	0FFF0000	14		21	27000000	54	
6	FF000F0F	18		22	79000000	58	
7	F0F0F0F0	1C		23	15100000	5C	
8	00000000	20		24	00000000	60	
9	00000000	24		25	00000000	64	
10	00000000	28		26	00000000	68	
11	00000000	2C		27	00000000	6C	
12	00000000	30		28	00000000	70	
13	00000000	34		29	00000000	74	
14	00000000	38		30	00000000	78	
15	00000000	3C		31	00000000	7C	

Because $128_{10} = 80_{16}$. 80_{16} is beyond the range of the RAM. PC_IF jump from 0x4C to 0x78. Then we write mstatus , flush all the pipeline registers, cancal regwrite, record epc = current epc and cause = 5.



When we finish handling load access fault, we execute mret to return to the instruction just after the load access fault because the trap program does.

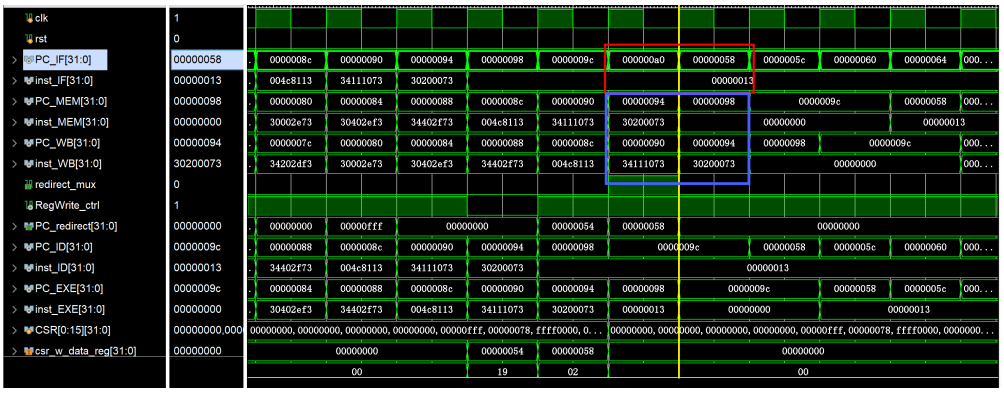
- store access fault



The related code block is as follows:

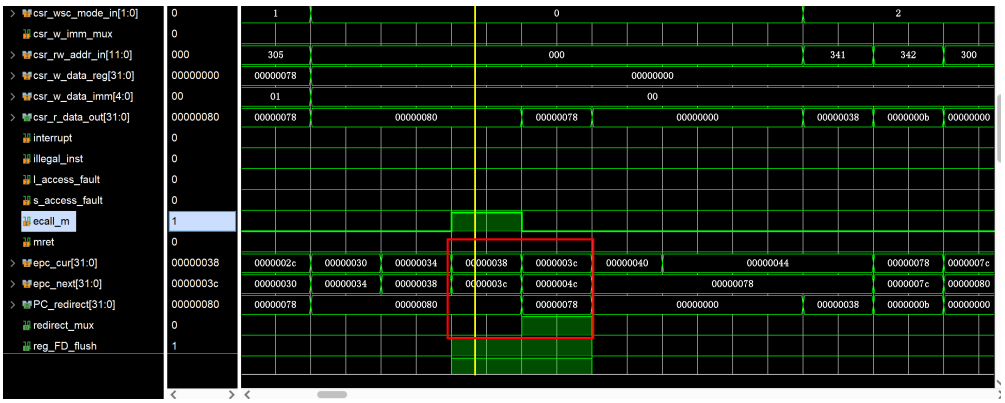
PC_IF	Instruction	Function
0x54	sw x1, 128(x0)	s access fault
0x78	csrr x25, 0x341	label is trap

Because $128_{10} = 80_{16}$. 80_{16} is beyond the range of the ram. PC_IF jump from 0x54 to 0x78. Then we write mstatus , flush all the pipeline registers, cancal regwrite, record epc = current epc and cause = 7.



When we finish handling store access fault, we execute mret to return to the instruction just after the store access fault because the trap program does.

- ecall

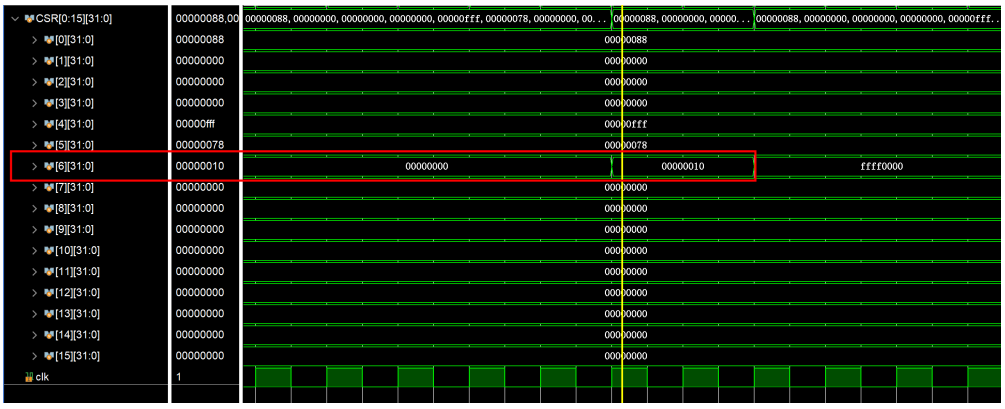


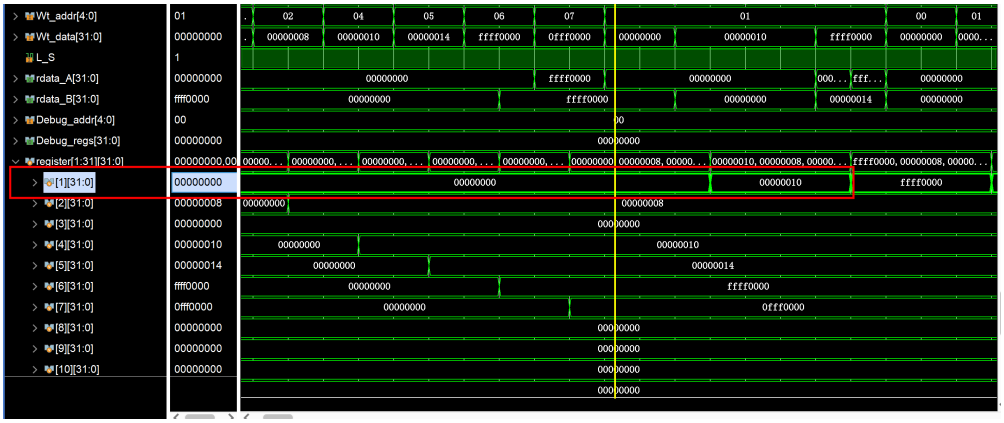
The related code block is as follows:

PC_IF	Instruction	Function
0x38	ecall	ecall
0x78	csrr x25, 0x341	label is trap

PC_IF jump from 0x38 to 0x78. Then we write mstatus , flush all the pipeline registers, cancal regwrite, record epc = current epc and cause = 11.

- CSR instructions

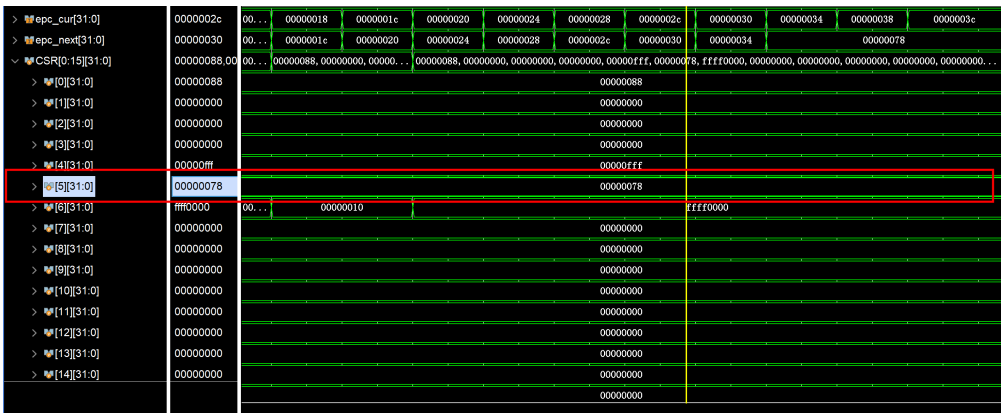




The related code block is as follows:

PC_IF	Instruction	Function
0x18	csrrwi x1, 0x306, 16	x1<=CSR[6],CSR[6]<=16
0x1C	csrr x1, 0x306	x1<=CSR[6]

Since 0x306 correspond with CSR[6], we can see that firstly, x1 set to 0 and CSR[6] set to 16, secondly, x1 set to 16.



The related code block is as follows:

PC_IF	Instruction	Function
0x30	csrw 0x305, x1	CSR[5]<=x1

Since 0x305 correspond with CSR[5], we can see that firstly, x1 set to CSR[5](0x78).

4 讨论与心得

This experiment mainly focuses on the handler after triggering the exception, which involves the modification of some specific registers. We need to add the function of supporting exception and interrupt processing to the pipeline CPU, which is mainly divided into: The first is to support the normal execution of CSR-type RISC-V instructions. This part does not include exception handling, so we can use normal instructions in the Trap test program; the second part is exception. Just read and write CSRRegs according to the general steps of exception handling; the last part is exception return.When the mret signal is reached, read mepc+4 and jump. The overall idea of Experiment 2 is very clear and has a close connection with the os lab.