计算机体系结构
实验报告

Lab3: pipeline CPU with forwarding pathes and predict-not-taken

孔伊宁 3170103294

2019 年 12 月 30 日

content

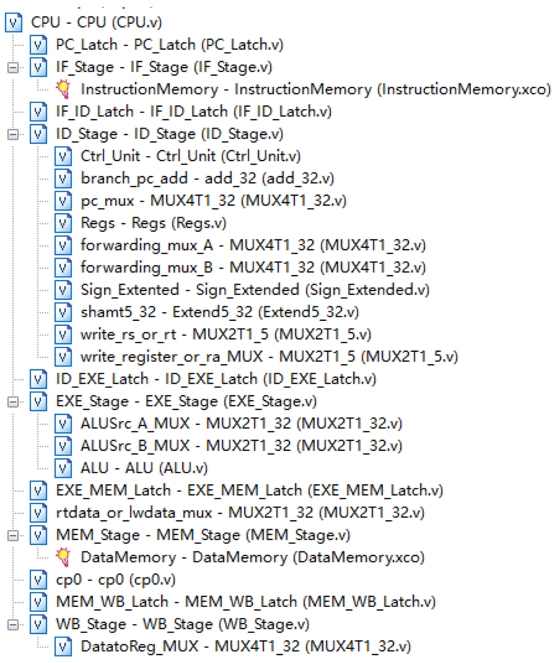
1 Experimental Purpose

Purpose:

- Understand the principle of CPU Interrupt and its processing procedure.
- Understand the function of CP0 coprocessor. Master the design methods of pipelined CPU supporting simple interrupt.
- master methods of program verification of Pipelined CPU supporting interrupt.

Task:

- Design of Pipelined CPU supporting Interrupt.
  - Design co-processor CP0
  - Design CPU Controller
  - Design datapath
- Understand the function of CP0 coprocessor. Master the design methods of pipelined CPU supporting simple interrupt.



```
CPU - CPU (CPU.v)
    PC_Latch - PC_Latch (PC_Latch.v)
    IF_Stage - IF_Stage (IF_Stage.v)
        InstructionMemory - InstructionMemory (InstructionMemory.xco)
    IF_ID_Latch - IF_ID_Latch (IF_ID_Latch.v)
    ID_Stage - ID_Stage (ID_Stage.v)
        Ctrl_Unit - Ctrl_Unit (Ctrl_Unit.v)
        branch_pc_add - add_32 (add_32.v)
        pc_mux - MUX4T1_32 (MUX4T1_32.v)
        Regs - Regs (Regs.v)
        forwarding_mux_A - MUX4T1_32 (MUX4T1_32.v)
        forwarding_mux_B - MUX4T1_32 (MUX4T1_32.v)
        Sign_Extented - Sign_Extended (Sign_Extended.v)
        shamt5_32 - Extend5_32 (Extend5_32.v)
        write_rs_or_rt - MUX2T1_5 (MUX2T1_5.v)
        write_register_or_ra_MUX - MUX2T1_5 (MUX2T1_5.v)
    ID_EXE_Latch - ID_EXE_Latch (ID_EXE_Latch.v)
    EXE_Stage - EXE_Stage (EXE_Stage.v)
        ALUSrc_A_MUX - MUX2T1_32 (MUX2T1_32.v)
        ALUSrc_B_MUX - MUX2T1_32 (MUX2T1_32.v)
        ALU - ALU (ALU.v)
    EXE_MEM_Latch - EXE_MEM_Latch (EXE_MEM_Latch.v)
    rtdata_or_lwdata_mux - MUX2T1_32 (MUX2T1_32.v)
    MEM_Stage - MEM_Stage (MEM_Stage.v)
        DataMemory - DataMemory (DataMemory.xco)
    cp0 - cp0 (cp0.v)
    MEM_WB_Latch - MEM_WB_Latch (MEM_WB_Latch.v)
    WB_Stage - WB_Stage (WB_Stage.v)
        DatatoReg_MUX - MUX4T1_32 (MUX4T1_32.v)
```

Figure 1.1: CPU Architecture directory

2 Experimental Principle and Content

2.1 CPU design

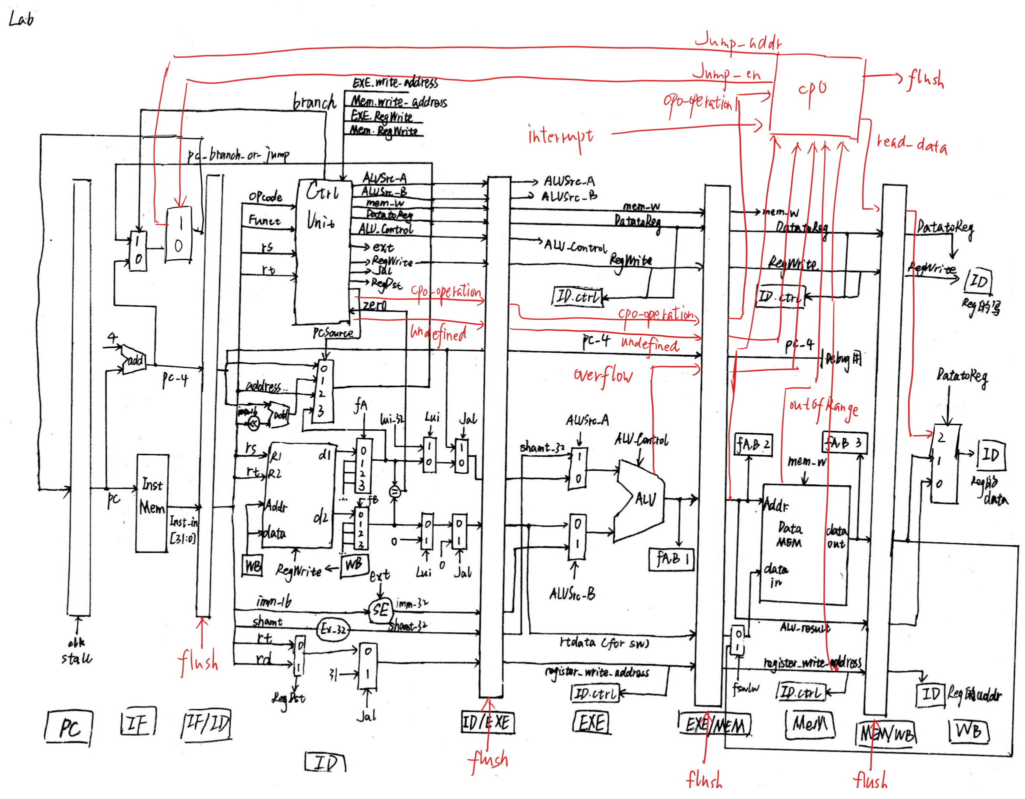


Figure 2.1: CPU design drawing

CPU consists of six main block(IF, ID, EXE, MEM, WB, cp0) and five latches(PC, IF/ID, ID/EXE, EXE/MEM, MEM/WB). CPU supports stall, forwarding, predit-not-taken and interrupt.

It supports 26 instructions, as followed:

instruction type	instructions
R-type	add,sub,and,or,xor,nor,slt,srl,sll,jr,jarl
I-type	addi,andi,ori,xori,slti,sw,lw,lui,beq,bne
J-type	j,jal
Privileged instructions	mfc0, mtc0, eret

2.2 Module cp0

cp0 has four registers, EHB register, EPC register, cause register and interrupt priority register.

EHB is set constantly as 32'd4 where is the location of Exception handler. When interrupt occurs, program will jump to 0000 0004 to handle exception.

EPC is the return address that instruction eret needs. It differs on the situation of Exception and Interrupt. When Exception occurs, it will be set by the PC of the instruction on the stage EXE. When interrupt occurs, it checks whether signal mem_m in the stage MEM is true. If true, EPC will be set by the PC of the instruction on the stage EXE, otherwise, it will be set by the PC of the instruction on the stage MEM.

Cause register indicates the type of the current interrupt. We hava five type, listing by priority from high to low, that is, overflow, undefided, outOfRange, INT1(SW[2]=1), INT2(SW[1]=1). The value of cause register also represents for priority of the interrupt.

Interrupt priority register saves the priority before the interrupt occurs. It's used to compare with cause register when two interrupt occur in sequence, and judge whether the current interruption should be interrupt by the later one or not.

The code of cp0 is as followed:

```
module cp0 (
  input wire clk,  // main clock
  // debug
  `ifdef DEBUG
  input wire [4:0] debug_addr,  // debug address
  output reg [31:0] debug_data,  // debug data
  `endif
  // operations (read in ID stage and write in EXE stage)
  input wire [1:0] operation,  // CP0 operation type
  input wire [4:0] read_address_rd,  // read address
  output reg [31:0] read_data,  // read data
  input wire [4:0] write_address_rd,  // write address
  input wire [31:0] write_data,  // write data
  // control signal
  input wire rst,  // synchronous reset
  input wire overflow,
  input wire undefined,
  input wire outOfRange,
  input wire ir_en,  // interrupt enable
  input wire [1:0] SW_INT,  // external interrupt input
  input wire [31:0] EPC_interrupt,  // target instruction address to store when interrupt
   occurred
  input wire [31:0] EPC_exception,  // target instruction address to store when exception
   occurred
  output reg flush,   // flush the inst before MEM stage when INT or EXCEPTION or ERET
   occur
  output reg jump_en,  // force jump enable signal when interrupt authorised or ERET
   occurred
  output reg [31:0] jump_addr  // target instruction address to jump to
);

  reg[31:0] EHB = 32'd4;
  reg[31:0] cause;       //$13 the interrupt cause
  reg[31:0] EPC;         //$14
  reg[31:0] INT_priority; //$15 5>4>3>2>1>0
  wire [31:0] cause_req;
  wire interruptRequest;

  initial begin
    cause = 32'd0;
    EPC = 32'd0;
    INT_priority = 32'd0;
  end

  assign cause_req = overflow    ? 32'd5 :
              undefined   ? 32'd4 :
              outOfRange ? 32'd3 :
              SW_INT[1]   ? 32'd2 :
              SW_INT[0]   ? 32'd1 : 32'b0;
  assign interruptRequest = cause_req == 32'b0 ? 0 : 1;

  //read
  always @* begin
    case(read_address_rd)
      32'd13: read_data = cause;
      32'd14: read_data = EPC;
      32'd15: read_data = INT_priority;
      default: read_data = 32'b0;
    endcase
  end

  //jump
  always @* begin
    jump_en = 0;
    flush = 0;
    //interrupt
    if(interruptRequest && cause_req > INT_priority) begin
      jump_addr <= 32'h4;
      jump_en <= 1;
      flush <= 1;
    end
    //eret
    if(operation == 2'b11) begin
      jump_addr <= EPC;
      jump_en <= 1;
      flush <= 1;
    end
  end

  always @(posedge clk) begin
    if(rst) begin
      EPC <= 0;
      cause <= 0;
      INT_priority <= 0;
    end
    else if(interruptRequest && cause_req > INT_priority) begin
      if(cause_req <= 2)
        EPC <= EPC_interrupt;
      else
        EPC <= EPC_exception;
      cause <= cause_req;
    end
  end

  //write
  always @(negedge clk) begin
    if(operation == 2'b10) begin
      case(write_address_rd)
        32'd13: cause <= write_data;
        32'd14: EPC <= write_data;
        32'd15: INT_priority <= write_data;
      endcase
    end
  end
endmodule
```

2.3 Instruction mft0

This instruction reads data from cp0'rd and write to Regs'rt.

In ID stage, Instruction is decoded. When the instruction is recognized as "mfc0", Ctrl Unit will output a signal **cp0_operation**, it indicates the operation that will be done to cp0. Here cp0_operation = 2'b01 which means "read data". Meanwhile, **read_addr**(which means the address of the register in cp0 that is read) and **write_addr**(which means the address of the register in Regs that the read data will write to) are output. When this signals are flying in MEM stage, cp0 gets cp0_operation and read_addr. Then cp0 will output the data in the corresponding register. In WB stage, the read data will write back to Regs.

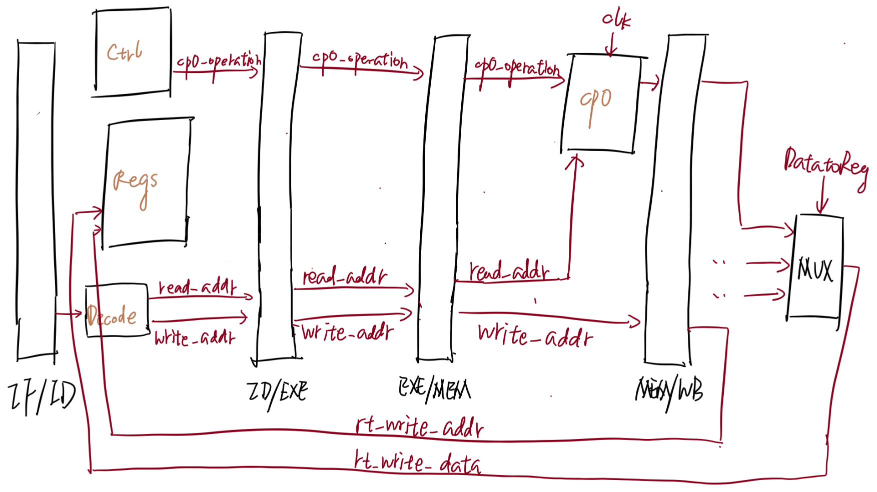


Figure 2.2: mfc0 datapath

2.4 Instruction mtc0

This instruction writes data from Regs'rt to cp0's rd.

In ID stage, Instruction is decoded. When the instruction is recognized as "mtc0", Ctrl Unit will output a signal **cp0_operation**, it indicates the operation that will be done to cp0. Here cp0_operation = 2'b10 which means "write data". The written data is from Regs'rt, so Regs outputs rs(set as 0) and rt and adds them in EXE stage and implements write operation in MEM stage. Write address is decoded from instruction.

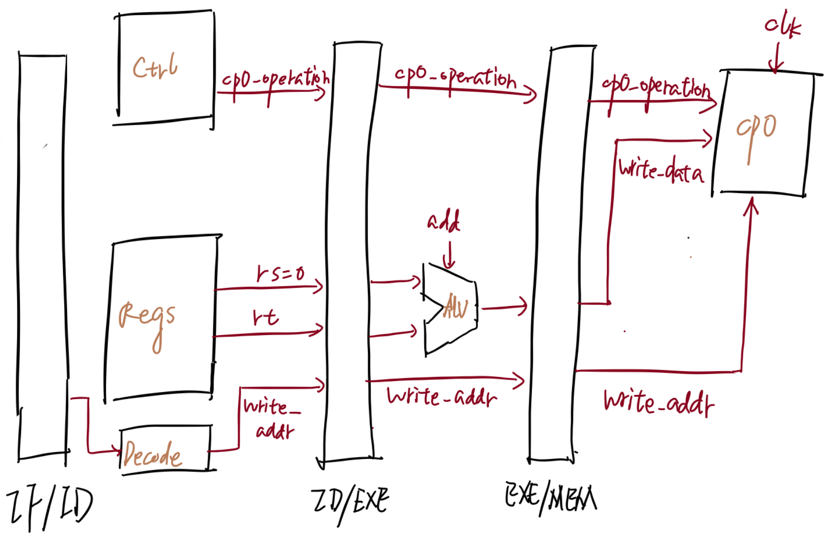


Figure 2.3: mtc0 datapath

2.5 Instruction eret

This instruction will set PC to EPC, program returnning to the instruction after the instruction where the interrupt/exception occurs. Everytime interrupt/exception occurs, setting PC to EPC. And when handler is over, set PC to EPC that has been stored before.

Figure 2.4: eret datapath

2.6 Interrupt/Exception

SW[2:1] stands for INT1 and INT2, which are the lowest priority interrupt. Three exception undefined, overflow, outOfRange hava higher priority, they can interrupt lower priority interrupt.

Undefined signal is set true when Decoder in ID stage find the instruction is not any expected one and can't understand. Overflow signal is set true when overflow happens in ALU in EXE stage. OutOfRange signal is set true when the address given to Data Memory is out of range.

Interrupt is a global signal that controled bt SW[2:1]. Once any one of them turns to high, interrupt signal will be sent to cp0. It accepts the input at the posedge of clk. Once an interruption happens and it is not happen in a time when a higher exception is processing, cp0 will set output signal **flush**true, save the current PC to EPC and set PC to EHB.

Exception is checked at the MEM stage. cp0 accepts the input of signal undefined, overflow and outOfRange. Once an exception is received, cp0 will set output signal **flush** true, save the current PC to EPC and set PC to EHB.

flush signal is accepts by the latches IF/ID, ID/EXE, EXE/MEM and MEM/WB. When flush signal is set, the latches will produce a bubble.

3 Experimental Phenomena

An instruction ”sw $t4, 1024($zero)” occurs exception of outOfRange. PC jumps to EHB.

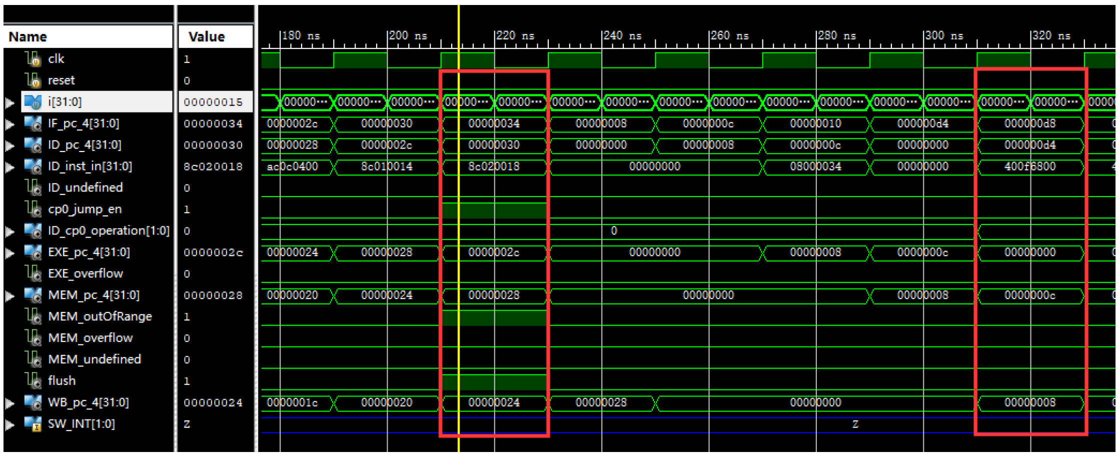
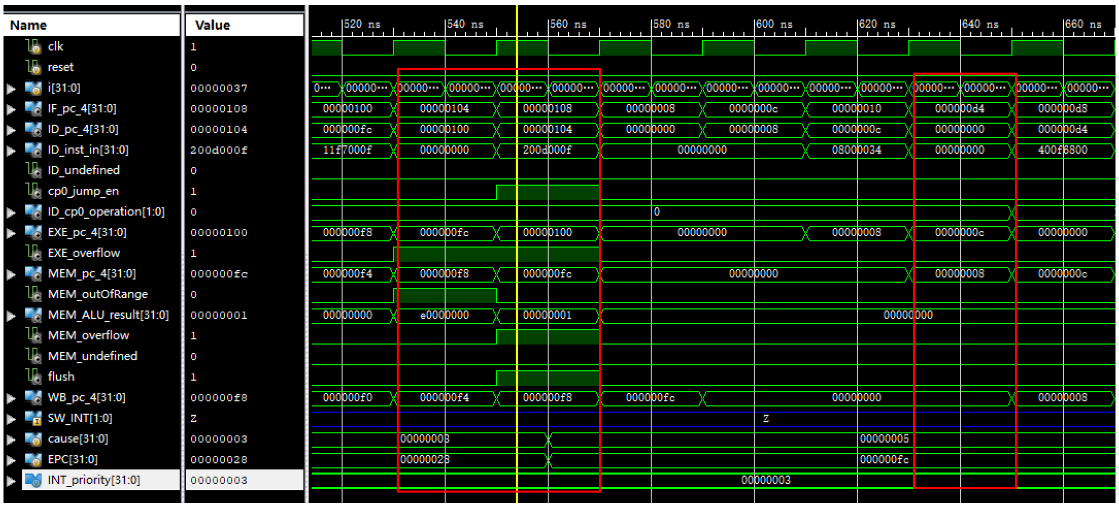


Figure 3.1: simulate: outOfRange

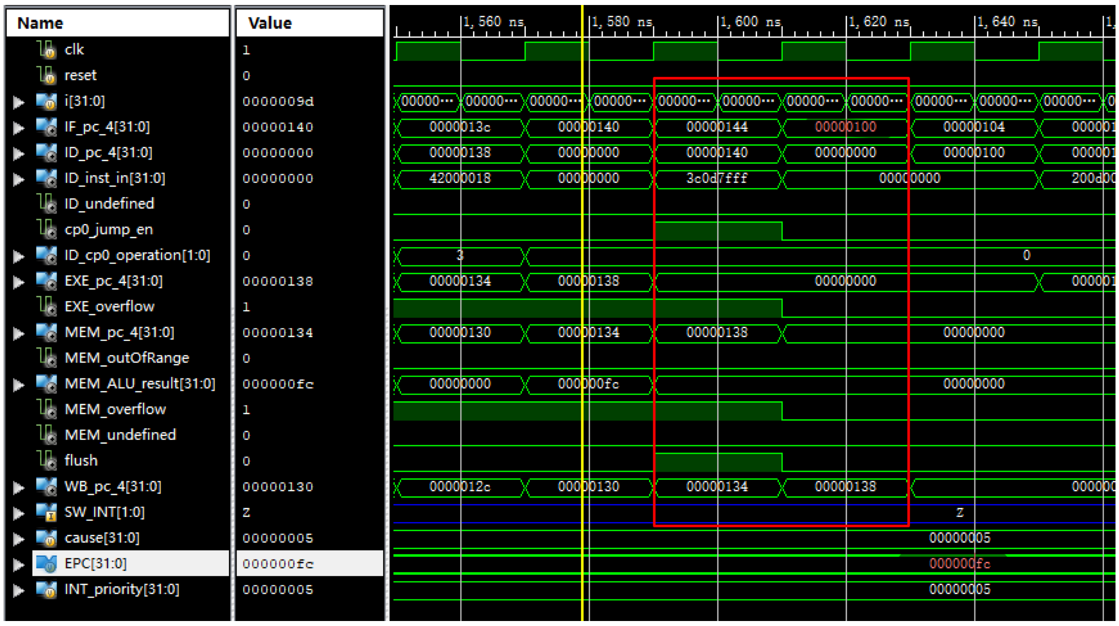
When in the handler, outOfRange occurs again, but this time as the priority of the new exception is the same as the processed exception, so no jump happens. But later, overflow occurs, overflow has higher priority than the processed exception, so it interrupts it and jumps to EHB again.



Name	Value
clk	1
reset	0
i[31:0]	00000037
IF_pc_4[31:0]	00000108
ID_pc_4[31:0]	00000104
ID_inst_in[31:0]	200d000f
ID_undefined	0
cp0_jump_en	1
ID_cp0_operation[1:0]	0
EXE_pc_4[31:0]	00000100
EXE_overflow	1
MEM_pc_4[31:0]	000000fc
MEM_outOfRange	0
MEM_ALU_result[31:0]	00000001
MEM_overflow	1
MEM_undefined	0
flush	1
WB_pc_4[31:0]	000000f8
SW_INT[1:0]	z
cause[31:0]	00000003
EPC[31:0]	00000028
INT_priority[31:0]	00000003

Figure 3.2: simulate: high priority interrupt

When the handler is over, eret is called. Set PC to EPC.



Name	Value
clk	1
reset	0
i[31:0]	0000009d
IF_pc_4[31:0]	00000140
ID_pc_4[31:0]	00000000
ID_inst_in[31:0]	00000000
ID_undefined	0
cp0_jump_en	0
ID_cp0_operation[1:0]	0
EXE_pc_4[31:0]	00000138
EXE_overflow	1
MEM_pc_4[31:0]	00000134
MEM_outOfRange	0
MEM_ALU_result[31:0]	000000fc
MEM_overflow	1
MEM_undefined	0
flush	0
WB_pc_4[31:0]	00000130
SW_INT[1:0]	z
cause[31:0]	00000005
EPC[31:0]	000000fc
INT_priority[31:0]	00000005

Figure 3.3: simulate: eret

The part of asm code is as follows:

```
j prepare
j int
prepare:
lui $t6, 57344; // 7 segment
```

```
addi $t4, $zero, 1
addi $s7, $zero, 2
addi $sp, $zero, 1024 // sp
mtc0 $zero, $12 //
sw $t4, 1024($zero)// mem exception: Figure 3.1

...

int:
...

sw $t4, 1024($zero)// mem exception
lui $t5, 0x7fff//
add $t5, $t5, $t5 // overflow: Figure 3.2
syscall //unrecognized
j recover

...

recover:
mtc0 $t4, $12
lw $t9, 0($sp)
lw $t8, 4($sp)
lw $t5, 8($sp)
addi $sp, $sp, 12
mtc0 $t9, $15 // mtc0 $t7, $cause
mtc0 $t8, $14 // mtc0 $t8, $epc
eret // set PC to EPC: Figure 3.3
```

4 Discussion and Review

At the beginning, I was hesitating to decide where to put the cp0, at the stage MEM or WB? PPT tells me to put on the stage WB, but I thought it may cause some problem when the instruction that occurs interrupt/exception is ”sw”. And I thought to let writing not happen is troublesome in stage WB. So I finally put it on the stage MEM. It still leaves me lots of confusion.

And the forwarding also troubles me a lot, It’s a mess to think about it.

I have not been to lab to load .bit. I use SW[2:1] to stand for interrupt, but I’m not sure whether it works. In the report I use different priority exception to represent for it.