中断与异常实现（断点，时间中断，地址错误）

姓名：梁朝阳　**学号**：2311561

目录

摘要

CP0 模块实现部分：硬件上单独拎出来了 CP0，补充完整了 STATUS 寄存器（寄存器 12）CAUSE 寄存器（寄存器 13）EPC 寄存器（寄存器 14）BADVADDR 寄存器（寄存器 8）COUNT 寄存器（寄存器 9）COMPARE 寄存器（寄存器 11）。实现了具体的检测逻辑。然后进行了总线上的一些调整等。

软件异常处理程序部分：不同类别的异常和终端必须采用硬件和软件结合实现，所以弄了几个非常简单的异常处理程序，对不同类型的中断和异常进行不同的处理。

关键词： 协处理器、检测、处理

1 实验目的与要求

1. 例外检测与实现
2. 自行修改 inst_rom 中的指令，复现例外
3. 实验报告中体现实现支持之前的波形，和之后的波形，并详细分析实验结果。

2 实验过程

2.1 CP0 模块总体架构

2.1.1 模块接口设计

CP0 模块（cp0.v）是异常处理系统的核心，其接口定义如下：

Listing 1: CP0 模块接口定义

```
module cp0(
    input           clk,      // 时钟
    input           resetn,   // 复位信号，低电平有效

    // 来自WB级的控制信号
    input           mtc0,     // MTC0指令标识
    input           mfc0,     // MFC0指令标识
    input     [ 7:0] cp0r_addr, // CP0寄存器地址 {寄存器号[4:0], 选择域[2:0]}
    input     [31:0] wdata,   // 写入CP0的数据

    // 异常相关信号
    input           syscall, // SYSCALL指令标识
    input           eret,     // ERET指令标识
    input     [31:0] pc,      // 当前PC值（用于保存到EPC）
    input           wb_valid, // WB级有效信号
    input           wb_over, // WB级完成信号

    // 统一异常总线（来自WB的最终裁决）
    input           ex_valid_i,      // 异常有效
    input     [ 4:0] ex_code_i,      // 异常编码
    input           ex_bd_i,         // 延迟槽异常
    input     [31:0] ex_pc_i,        // 发生异常的PC
    input           badvaddr_valid_i, // 错误地址有效
    input     [31:0] badvaddr_i,     // 错误地址

    // CP0寄存器读数据输出
```

```
    output    [31:0] cp0r_rdata,     // CP0寄存器读数据（用于MFC0）

    // 异常处理输出
    output           cancel,   // 取消流水线信号
    output           exc_valid, // 异常有效信号
    output    [31:0] exc_pc,  // 异常入口地址或ERET返回地址

    // 寄存器值输出（用于异常处理）
    output    [31:0] cp0r_status,    // STATUS寄存器值
    output    [31:0] cp0r_cause,      // CAUSE寄存器值
    output    [31:0] cp0r_epc,        // EPC寄存器值

    // 中断输出
    output           c0_int           // 中断有效信号
);
```

2.1.2 设计原则

CP0 模块的设计遵循以下原则：

1. **统一异常总线**：所有异常（包括 SYSCALL、BREAK、地址错误、中断）都通过统一的异常总线（`ex_valid_i`，`ex_code_i` 等）传递到 CP0，由 CP0 统一处理。
2. **寄存器访问控制**：通过 MTC0/MFC0 指令访问 CP0 寄存器，使用写掩码（WMASK）控制可写位域。
3. **异常优先级**：在 WB 级进行异常仲裁，确定优先级后统一传递给 CP0。
4. **中断检测**：CP0 内部检测定时器中断条件，输出中断信号。

2.2 CP0 寄存器实现

2.2.1 寄存器列表

本实现包含以下 CP0 寄存器：

Table 1: CP0 寄存器列表

寄存器号	选择域	名称	功能描述
12	0	STATUS	系统状态寄存器
13	0	CAUSE	异常原因寄存器
14	0	EPC	异常程序计数器
8	0	BADVADDR	错误虚拟地址寄存器
9	0	COUNT	定时器计数寄存器
11	0	COMPARE	定时器比较寄存器

2.2.2 STATUS 寄存器（寄存器 12）

STATUS 寄存器用于控制系统状态和中断使能。

Listing 2: STATUS 寄存器位域定义

```
// STATUS寄存器位域
wire status_ie;     // bit 0: 全局中断使能 (IE)
wire status_exl;    // bit 1: 异常级别 (EXL)
wire [7:0] status_im; // bit 15:8: 中断屏蔽位 (IM)

// STATUS寄存器写掩码（支持IE、EXL、IM位）
wire [31:0] STATUS_WMASK;
assign STATUS_WMASK = 32'h0000_8103; // bit 0(IE), bit 1(EXL), bit 15:8(IM)
```

关键位域说明：

- IE (bit 0)：全局中断使能位。当 IE=0 时，所有中断被屏蔽。
- EXL (bit 1)：异常级别位。当 EXL=1 时，CPU 处于异常处理模式，新的中断和异常被屏蔽。
- IM[7:0] (bit 15:8)：中断屏蔽位。每一位对应一个中断源，IM[7] 对应定时器中断。

写入控制：

Listing 3: STATUS 寄存器写入逻辑

```
if (status_wen) begin
    status <= (status & ~STATUS_WMASK) | (wdata & STATUS_WMASK);
end
```

2.2.3 CAUSE 寄存器（寄存器 13）

CAUSE 寄存器记录异常原因和中断状态。

Listing 4: CAUSE 寄存器位域定义

```
// CAUSE寄存器位域
wire cause_bd;      // bit 31: 延迟槽标志 (BD)
wire cause_ti;      // bit 30: 定时器中断标志 (TI)
wire [7:0] cause_ip; // bit 15:8: 中断挂起位 (IP)
wire [4:0] cause_excode; // bit 6:2: 异常编码 (ExcCode)

// CAUSE寄存器写掩码（支持IP[1:0]位）
wire [31:0] CAUSE_WMASK;
assign CAUSE_WMASK = 32'h0000_0300; // bit 9:8(IP[1:0])
```

关键位域说明：

- `BD (bit 31)`：延迟槽标志。当异常发生在分支指令的延迟槽中时，BD=1。
- `TI (bit 30)`：定时器中断标志。当 COUNT == COMPARE 时，TI=1。
- `IP[7:0] (bit 15:8)`：中断挂起位。IP[7] 对应定时器中断，由硬件自动更新。
- `ExcCode (bit 6:2)`：异常编码，标识异常类型。

异常编码表：

Table 2: 异常编码表

异常编码	异常类型
0	中断 (Interrupt)
4	地址错误-加载 (AdEL)
5	地址错误-存储 (AdES)
8	系统调用 (SYSCALL)
9	断点 (BREAK)
12	算术溢出 (OV)

2.2.4 EPC 寄存器（寄存器 14）

EPC 寄存器保存发生异常时的程序计数器值。

Listing 5: EPC 寄存器写入逻辑

```
// 统一异常处理：所有异常都通过ex_valid_i传递
if (ex_valid_i && wb_valid) begin
    status[1] <= 1'b1;                // EXL
    cause[31] <= ex_bd_i;             // BD
    cause[6:2] <= ex_code_i;          // ExcCode
    epc <= ex_bd_i ? ex_pc_i : ex_pc_i; // 写入分支PC或出错PC
    if (badvaddr_valid_i) begin
        badvaddr <= badvaddr_i;
    end
end
```

说明：

- 当异常发生时，EPC 保存发生异常的指令地址。
- 如果异常发生在延迟槽中（BD=1），EPC 保存分支指令的地址。
- ERET 指令执行时，CPU 跳转到 EPC 保存的地址继续执行。

2.2.5 BADVADDR 寄存器（寄存器 8）

BADVADDR 寄存器保存导致地址错误的虚拟地址。

Listing 6: BADVADDR 寄存器写入逻辑

```
if (badvaddr_valid_i) begin
    badvaddr <= badvaddr_i;
end
```

说明：

- 仅在地址错误异常（AdEL/AdES）时写入。
- 由 MEM 级检测地址对齐错误，通过异常总线传递到 CP0。

2.2.6 COUNT 寄存器（寄存器 9）

COUNT 寄存器是定时器计数器，用于定时器中断。

Listing 7: COUNT 寄存器实现

```
// 定时器时钟分频（每两个时钟周期翻转一次，降低计数频率）
reg time_tick;
always @(posedge clk) begin
    if (!resetn) begin
        time_tick <= 1'b0;
    end else begin
        time_tick <= ~time_tick;
    end
end

// COUNT寄存器：可写，或每两个时钟周期自增
always @(posedge clk) begin
    if (!resetn) begin
        count <= 32'd0;
    end else begin
        if (count_wen) begin
            count <= wdata;
        end else if (time_tick) begin
            count <= count + 1'b1;
        end
    end
end
```

说明：

- COUNT 寄存器可通过 MTC0 指令写入。
- 正常情况下，每两个时钟周期自增 1（通过 `time_tick` 分频）。

- 当 COUNT == COMPARE 时，触发定时器中断。

2.2.7 COMPARE 寄存器（寄存器 11）

COMPARE 寄存器是定时器比较值，用于定时器中断。

Listing 8: COMPARE 寄存器实现

```
// COMPARE寄存器：可写，写入时清除定时器中断
always @(posedge clk) begin
    if (!resetn) begin
        compare <= 32'd0;
    end else begin
        if (compare_wen) begin
            compare <= wdata;
        end
    end
end

// 定时器中断标志（cause_ti_reg）
always @(posedge clk) begin
    if (!resetn) begin
        cause_ti_reg <= 1'b0;
    end else begin
        if (compare_wen) begin
            cause_ti_reg <= 1'b0; // 写入COMPARE时清除
        end else if (count_eq_compare) begin
            cause_ti_reg <= 1'b1; // COUNT == COMPARE时置位
        end
    end
end
```

说明：

- COMPARE 寄存器可通过 MTC0 指令写入。
- 写入 COMPARE 寄存器会清除定时器中断标志（TI 位）。
- 当 COUNT == COMPARE 时，置位 TI 标志，触发中断。

2.3 异常处理流程

2.3.1 异常检测与仲裁

异常检测分布在流水线的不同阶段：

1. **ID 级**：检测 SYSCALL、BREAK 指令。

2. **MEM 级**：检测地址对齐错误（AdEL/AdES）。

3. **WB 级**：统一仲裁所有异常，确定优先级。

4. **CP0**：检测定时器中断。

2.3.2 WB 级异常仲裁

WB 级负责统一仲裁所有异常，确定优先级后传递给 CP0：

Listing 9: WB 级异常仲裁逻辑

```
// 异常仲裁逻辑（优先级：中断 > 地址错 > BREAK > SYSCALL）
// 非中断异常（地址错、BREAK、SYSCALL）
assign wb_ex_valid_no_int = (mem_ex_adel_wb | mem_ex_ades_wb | brk_wb | syscall)
    ? WB_valid : 1'b0;
assign wb_ex_code_no_int = mem_ex_adel_wb ? 5'd4 : // AdEL
                           mem_ex_ades_wb ? 5'd5 : // AdES
                           brk_wb ? 5'd9 :       // BREAK
                           syscall ? 5'd8 : 5'd0; // SYSCALL

// 最终异常有效信号（中断优先级最高）
assign wb_ex_valid = (cp0_int && WB_valid) | wb_ex_valid_no_int;
assign wb_ex_code = cp0_int ? 5'd0 : wb_ex_code_no_int; // 中断异常码为0
assign wb_ex_bd = 1'b0;      // 延迟槽后续接入
assign wb_ex_pc = pc;        // 异常PC（地址错与syscall均取当前pc）
```

异常优先级：

1. 定时器中断（最高优先级）

2. 地址错误（AdEL/AdES）

3. BREAK 异常

4. SYSCALL 异常

2.3.3 异常处理流程

当异常发生时，CP0 执行以下操作：

1. **设置 EXL 位**：STATUS[1] = 1，进入异常处理模式。

2. **保存 EPC**：将发生异常的 PC 保存到 EPC 寄存器。

3. **设置 CAUSE**：写入异常编码（ExcCode）和延迟槽标志（BD）。

4. **保存 BADVADDR**：如果是地址错误，保存错误地址。

5. **跳转异常入口**：CPU 跳转到异常入口地址（0x0）。

Listing 10: CP0 异常处理逻辑

```
// 统一异常处理：所有异常都通过ex_valid_i传递
if (ex_valid_i && wb_valid) begin
    status[1] <= 1'b1;                // EXL
    cause[31] <= ex_bd_i;             // BD
    cause[6:2] <= ex_code_i;          // ExcCode
    epc <= ex_bd_i ? ex_pc_i : ex_pc_i; // 写入分支PC或出错PC
    if (badvaddr_valid_i) begin
        badvaddr <= badvaddr_i;
    end
end
```

2.3.4 ERET 指令处理

ERET 指令用于从异常处理返回：

Listing 11: ERET 指令处理

```
// ERET指令：清除EXL位
if (eret && wb_valid) begin
    status[1] <= 1'b0; // 清EXL
end

// 异常/返回对外信号
assign exc_pc = eret ? epc : `EXC_ENTER_ADDR;
```

说明：

- ERET 执行时，清除 STATUS[1]（EXL 位），退出异常处理模式。
- CPU 跳转到 EPC 保存的地址继续执行。

2.4 中断处理

2.4.1 定时器中断检测

定时器中断由 CP0 内部检测：

Listing 12: 定时器中断检测逻辑

```
// COUNT == COMPARE检测
assign count_eq_compare = (count == compare);

// 定时器中断标志（cause_ti_reg）
always @(posedge clk) begin
```

```
    if (!resetn) begin
        cause_ti_reg <= 1'b0;
    end else begin
        if (compare_wen) begin
            cause_ti_reg <= 1'b0; // 写入COMPARE时清除
        end else if (count_eq_compare) begin
            cause_ti_reg <= 1'b1; // COUNT == COMPARE时置位
        end
    end
end

// CAUSE寄存器位域更新
always @(posedge clk) begin
    // cause[30]: TI位（定时器中断标志）
    if (!ex_valid_i || !wb_valid) begin
        cause[30] <= cause_ti_reg;
    end

    // cause[15:8]: IP位（中断挂起位）
    // IP[7] = TI（定时器中断）
    if (!ex_valid_i || !wb_valid) begin
        cause[15:8] <= {cause_ti_reg, 5'd0, cause[9:8]};
    end
end
```

2.4.2 中断使能条件

中断需要满足以下条件才能触发：

Listing 13: 中断检测逻辑

```
// 中断检测逻辑
// 中断条件：有中断挂起 && 对应中断使能 && 全局中断使能 && 不在异常级别
assign c0_int = |(cause_ip[7:0] & status_im[7:0]) & status_ie & !status_exl;
```

中断使能条件：

1. 有中断挂起（IP 位为 1）
2. 对应中断屏蔽位使能（IM 位为 1）
3. 全局中断使能（IE=1）
4. 不在异常级别（EXL=0）

2.5 总线排布

2.5.1 MEM->WB 总线

MEM 级通过总线将异常信息传递给 WB 级：

Listing 14: MEM->WB 总线定义

```
`define MEM_WB_BUS_WIDTH 153

// 扩展MEM->WB总线，新增：mem_ex_adel, mem_ex_ades, badvaddr(dm_addr)
assign MEM_WB_bus = {rf_wen,rf_wdest,            // WB需要使用的信号
                     mem_result,                 // 最终要写回寄存器的数据
                     lo_result,                  // 乘法低32位结果
                     hi_write,lo_write,          // HI/LO写使能
                     mfhi,mflo,                  // WB需要使用的信号
                     mtc0,mfc0,cp0r_addr,syscall,brk,eret, // WB需要使用的信号
                     mem_ex_adel, mem_ex_ades,   // 地址异常标志（新增）
                     dm_addr,                    // BADVADDR（新增）
                     pc};                        // PC值
```

总线位域说明：

- `mem_ex_adel`：Load 地址错误标志
- `mem_ex_ades`：Store 地址错误标志
- `dm_addr`：访存地址（用于 BADVADDR）
- `syscall, brk, eret`：异常指令标识

2.5.2 WB->CP0 异常总线

WB 级通过异常总线将异常信息传递给 CP0：

Listing 15: WB->CP0 异常总线

```
// 统一异常总线（传递给CP0）
assign wb_ex_valid = (cp0_int && WB_valid) | wb_ex_valid_no_int;
assign wb_ex_code = cp0_int ? 5'd0 : wb_ex_code_no_int;
assign wb_ex_bd = 1'b0;
assign wb_ex_pc = pc;
assign wb_badvaddr_valid = mem_ex_adel_wb | mem_ex_ades_wb;
assign wb_badvaddr = mem_badvaddr_wb;
```

2.6 各异常类型的实现

2.6.1 SYSCALL 异常

检测位置：ID 级（指令译码）

Listing 16: SYSCALL 指令检测

```
// decode.v
assign inst_SYSCALL = (op == 6'b000000) & (funct == 6'b001100);
```

处理流程：

1. ID 级检测到 SYSCALL 指令，将 `syscall` 信号传递到 WB 级。
2. WB 级仲裁，设置异常码为 8。
3. CP0 保存 EPC（SYSCALL 指令地址），设置 EXL=1，跳转到异常入口。
4. 异常处理程序读取 CAUSE，判断为 SYSCALL，执行相应处理。
5. 处理完成后，EPC += 4，ERET 返回。

2.6.2 BREAK 异常

检测位置：ID 级（指令译码）

Listing 17: BREAK 指令检测

```
// decode.v
assign inst_BREAK = (op == 6'b000000) & (funct == 6'b001101);
```

处理流程：

1. ID 级检测到 BREAK 指令，将 `brk` 信号传递到 WB 级。
2. WB 级仲裁，设置异常码为 9。
3. CP0 保存 EPC（BREAK 指令地址），设置 EXL=1，跳转到异常入口。
4. 异常处理程序读取 CAUSE，判断为 BREAK，执行相应处理。
5. 处理完成后，EPC += 4，ERET 返回。

2.6.3 地址错误异常（AdEL/AdES）

检测位置：MEM 级（访存阶段）

Listing 18: 地址对齐错误检测

```
// mem.v
wire mem_ex_adel; // load 地址错
wire mem_ex_ades; // store 地址错
```

```
assign mem_ex_adel = MEM_valid && inst_load && ls_word && (dm_addr[1:0]!=2'b00);
assign mem_ex_ades = MEM_valid && inst_store && ls_word && (dm_addr[1:0]!=2'b00);
```

处理流程：

1. MEM 级检测到地址不对齐（低 2 位不为 00），设置 `mem_ex_adel` 或 `mem_ex_ades`。
2. 将错误地址（`dm_addr`）和异常标志传递到 WB 级。
3. WB 级仲裁，设置异常码为 4（AdEL）或 5（AdES）。
4. CP0 保存 EPC（出错指令地址），保存 BADVADDR（错误地址），设置 EXL=1，跳转到异常入口。
5. 异常处理程序读取 CAUSE 和 BADVADDR，执行相应处理。
6. 处理完成后，EPC += 4，ERET 返回。

2.6.4 定时器中断

检测位置：CP0 内部

Listing 19: 定时器中断检测

```
// cp0.v
assign count_eq_compare = (count == compare);
assign c0_int = |(cause_ip[7:0] & status_im[7:0]) & status_ie & !status_exl;
```

处理流程：

1. CP0 检测到 COUNT == COMPARE，置位 TI 标志。
2. 检查中断使能条件（IE=1, IM[7]=1, EXL=0）。
3. 如果条件满足，输出 `c0_int` 信号。
4. WB 级仲裁，设置异常码为 0（中断）。
5. CP0 保存 EPC（被中断指令地址），设置 EXL=1，跳转到异常入口。
6. 异常处理程序读取 CAUSE，检查 TI 位，执行定时器中断处理。
7. 处理完成后，写入 COMPARE 清除 TI 位，ERET 返回。

2.7 CP0 寄存器访问

2.7.1 MTC0 指令（写入 CP0 寄存器）

MTC0 指令用于写入 CP0 寄存器：

Listing 20: MTC0 指令处理

```
// 写允许信号
wire mtc0_wr; // MTC0写使能（排除异常时写入）
assign mtc0_wr = mtc0 && wb_valid && !ex_valid_i; // 异常时不写入

assign status_wen = mtc0_wr && sel_status;
assign cause_wen = mtc0_wr && sel_cause;
assign epc_wen   = mtc0_wr && sel_epc;
assign count_wen = mtc0_wr && sel_count;
assign compare_wen = mtc0_wr && sel_compare;
assign badvaddr_wen = mtc0_wr && sel_badvaddr;

// 写入逻辑（使用写掩码）
if (status_wen) begin
    status <= (status & ~STATUS_WMASK) | (wdata & STATUS_WMASK);
end
```

说明:

- 异常发生时，禁止写入 CP0 寄存器（`!ex_valid_i`）。
- 使用写掩码（WMASK）控制可写位域。
- STATUS 和 CAUSE 寄存器只有部分位可写。

2.7.2 MFC0 指令（读取 CP0 寄存器）

MFC0 指令用于读取 CP0 寄存器:

Listing 21: MFC0 指令处理

```
// MFC0读
assign cp0r_rdata = sel_status ? status :
                    sel_cause  ? cause  :
                    sel_epc    ? epc    :
                    sel_count  ? count  :
                    sel_compare ? compare :
                    sel_badvaddr? badvaddr : 32'd0;
```

说明:

- 根据 CP0 寄存器地址（`cp0r_addr`）选择对应的寄存器。
- 读数据通过 `cp0r_rdata` 输出，写回到通用寄存器。

2.8 流水线控制信号

2.8.1 cancel 信号

`cancel` 信号用于取消流水线中已取出的指令：

Listing 22: cancel 信号生成

```
assign cancel = (ex_valid_i | eret | c0_int) && wb_over;
```

说明：

- 当异常或 ERET 发生时，需要取消流水线中已取出的指令。
- `cancel` 信号在 WB 级完成时（`wb_over`）发出。

2.8.2 exc_valid 和 exc_pc 信号

`exc_valid` 和 `exc_pc` 信号用于控制异常跳转：

Listing 23: 异常跳转信号生成

```
assign exc_valid = (ex_valid_i | eret | c0_int) && wb_valid;
assign exc_pc  = eret ? epc : `EXC_ENTER_ADDR;
```

说明：

- `exc_valid`：异常有效信号，控制是否跳转。
- `exc_pc`：跳转目标地址，ERET 时返回 EPC，异常时跳转到异常入口（0x0）。

3 软件异常处理程序与仿真结果

本节分别给出四种异常类型（SYSCALL、BREAK、地址错误、定时器中断）的完整软件异常处理程序。每个程序均由两部分组成：

- **异常处理程序**：位于 ROM 起始（0x00 起），处理对应异常；
- **测试主程序**：位于复位地址（如 0x30 或 0x40 起），触发并验证异常；

所有指令均为 32 位机器码，每条占 4 字节，地址按 4 递增。

3.1 SYSCALL 异常测试程序

测试目的：验证 SYSCALL 异常（异常码 8）在触发后能正确保存 EPC、修改并恢复程序执行。
程序说明：执行 `syscall` 指令触发异常，异常处理程序判断 ExcCode=8 后进行处理，返回后继续执行主程序。

Table 3: SYSCALL 异常测试程序指令表

序号	地址 (Hex)	机器码	汇编 / 说明
1	0x00	AC010000	`sw $1, 0($0)` 保存 `$1` 到 `Mem[0]`
2	0x04	40016800	`mfc0 $1, $13` 读取 CAUSE 寄存器到 `$1`
3	0x08	00010882	`srl $1, $1, 2` 右移 2 位，将异常码移到低位
4	0x0C	3421001F	`andi $1, $1, 0x1F` 取低 5 位异常码
5	0x10	24020008	`addiu $2, $0, 8` `$2 = 8`（SYSCALL 异常码）
6	0x14	10220002	`beq $1, $2, 2` 若为 SYSCALL，分支到地址 0x20
7	0x18	40017000	`mfc0 $1, $14` 从 EPC 读出异常发生地址
8	0x1C	24210004	`addiu $1, $1, 4` EPC += 4
9	0x20	40817000	`mtc0 $1, $14` 将更新后的 EPC 写回
10	0x24	8C010000	`lw $1, 0($0)` 恢复 `$1`
11	0x28	42000018	`eret` 从异常返回
12	0x2C	40017000	`mfc0 $1, $14` SYSCALL 分支：读 EPC
13	0x30	24210004	`addiu $1, $1, 4` SYSCALL：EPC += 4
14	0x34	40817000	`mtc0 $1, $14` SYSCALL：写回 EPC
15	0x38	8C010000	`lw $1, 0($0)` SYSCALL：恢复 `$1`
16	0x3C	42000018	`eret` SYSCALL：从异常返回
17	0x40	24010001	`addiu $1, $0, 1` `$1 = 1`，初始化
18	0x44	40816000	`mtc0 $1, $12` 写 STATUS 寄存器
19	0x48	0000000C	`syscall` 触发 SYSCALL 异常
20	0x4C	24010001	`addiu $1, $0, 1` 异常返回后 `$1 = 1`
21	0x50	24020002	`addiu $2, $0, 2` `$2 = 2`
22	0x54	00411821	`addu $3, $2, $1` `$3 = $2 + $1 = 3`
23	0x58	AC030000	`sw $3, 0($0)` 把结果 `$3` 存到 `Mem[0]`
24	0x5C	08000013	`j 0x4C` 跳回 0x4C 形成死循环

程序入口： 0x2C **异常入口：** 0x00 **测试结果：** Mem[0] 中应写入 3。

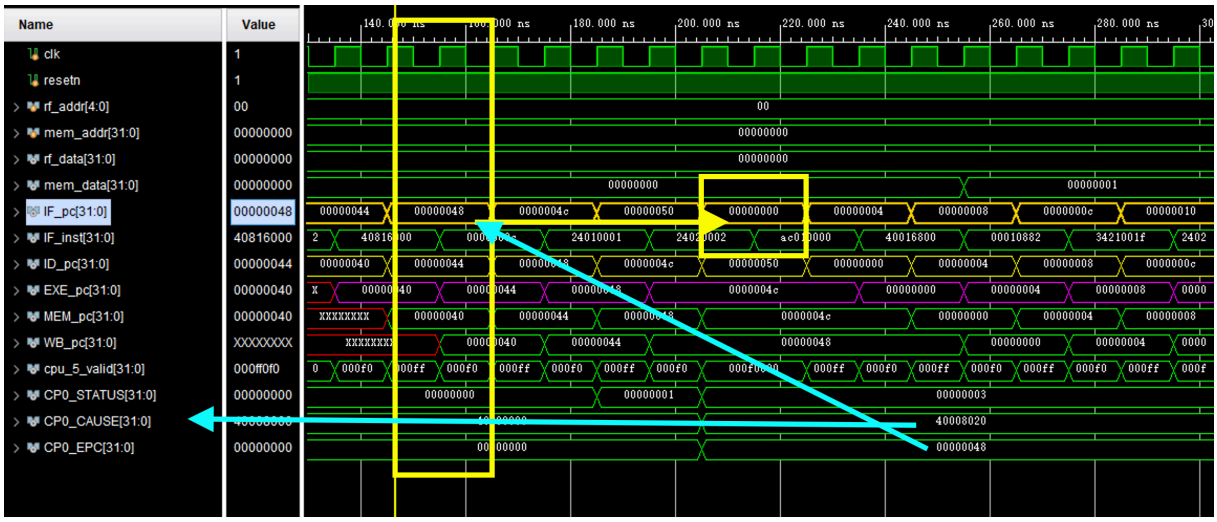

图 3.1: syscall

这个不太好对比，因为之前就也是正确的，这里 48H 是 syscall 的地址

3.2 AdEL 异常测试程序

测试目的：验证 AdEL 异常（异常码 9）触发与恢复机制。

Table 4: AdEL 异常测试程序指令表

序号	地址	机器码	汇编 / 说明
1	0x00	AC010000	`sw $1, 0($0)` 保存 $1
2	0x04	40016800	`mfc0 $1, $13` 读取 CAUSE
3	0x08	00010882	`srl $1, $1, 2` 提取 ExcCode
4	0x0C	3421001F	`andi $1, $1, 0x1F` 取异常码
5	0x10	24020004	`addiu $2, $0, 4` $2=4（AdEL 异常码）
6	0x14	10220003	`beq $1,$2,3` 若 ExcCode=4，跳到 AdEL 分支
7	0x18	40017000	`mfc0 $1,$14` 默认异常：读 EPC
8	0x1C	24210004	`addiu $1,$1,4` EPC += 4
9	0x20	40817000	`mtc0 $1,$14` 写回 EPC
10	0x24	8C010000	`lw $1,0($0)` 恢复 $1
11	0x28	42000018	`eret` 返回（默认异常）
12	0x2C	40017000	`mfc0 $1,$14` AdEL 分支：读 EPC
13	0x30	24210004	`addiu $1,$1,4` EPC += 4
14	0x34	40817000	`mtc0 $1,$14` 写回 EPC
15	0x38	8C010000	`lw $1,0($0)` 恢复 $1
16	0x3C	42000018	`eret` 返回（AdEL 异常）
17	0x40	24010001	`addiu $1,$0,1` 程序初始化 $1=1
18	0x44	40816000	`mtc0 $1,$12` 写 STATUS 寄存器
19	0x48	240200AA	`addiu $2,$0,0xAA` $2=0xAA
20	0x4C	AC020000	`sw $2,0($0)` 把 $2 存入 Mem[0]
21	0x50	8C010001	`lw $1,1($0)` 非对齐访问，触发 AdEL 异常
22	0x54	24010001	`addiu $1,$0,1` 异常返回后继续执行
23	0x58	24020002	`addiu $2,$0,2` $2=2
24	0x5C	00411821	`addu $3,$2,$1` $3=3
25	0x60	AC030004	`sw $3,4($0)` 存结果到 Mem[4]
26	0x64	08000019	`j 0x64` 死循环

50H 是异常处理地址，可以看到 50H 处，后面的指令在 50H 走到 EXE 阶段后，跳转为了 00H（异常处理成功，也就是成功检测出了异常）：

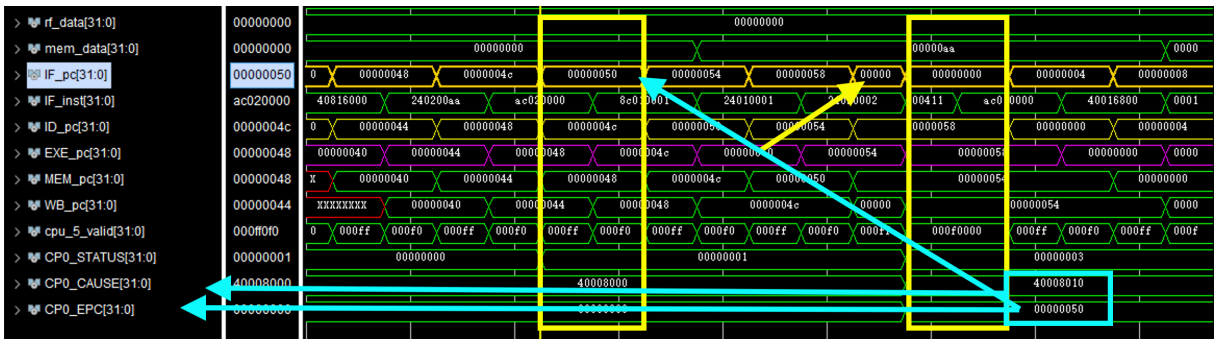

图 3.2: 错误地址

同时设置 Cause 寄存器中的 ExcCode 位为和 EPC 为 50H。操作系统/异常处理函数后续可以根据这两个寄存器的值做更详细的处理。

3.3 定时器中断触发波形分析

下图为定时器中断测试程序在仿真时的关键波形。从图中可以看到，当 CPU 执行主程序到初始化阶段时，COUNT 与 COMPARE 相等，触发了定时器中断信号，CP0 模块响应后进入异常处理流程。

Table 5: 定时器中断测试程序指令表

序号	地址	机器码	汇编指令 / 说明
1	0x00	AC010000	`sw $1, 0($0)` 备份 `$1` 到内存
2	0x04	40016800	`mfc0 $1, $13` 读 CAUSE
3	0x08	00010882	`srl $1, $1, 2` CAUSE 右移 2，取 ExcCode 到低位
4	0x0C	3421001F	`andi $1, $1, 0x1F` 取异常码低 5 位
5	0x10	24020000	`addiu $2, $0, 0` `$2=0`（中断异常码）
6	0x14	10220006	`beq $1, $2, 6` 是中断则跳到 0x34
7	0x18	40017000	`mfc0 $1, $14` 默认异常：读 EPC
8	0x1C	24210004	`addiu $1, $1, 4` EPC += 4
9	0x20	40817000	`mtc0 $1, $14` 写回 EPC
10	0x24	8C020004	`lw $2, 4($0)` 恢复 `$2`
11	0x28	8C010000	`lw $1, 0($0)` 恢复 `$1`
12	0x2C	42000018	`eret` 默认异常返回
13	0x34	40016800	`mfc0 $1, $13` 中断分支：再读 CAUSE
14	0x38	0001089E	`srl $1, $1, 30` 取 TI 位到 bit0
15	0x3C	34210001	`andi $1, $1, 1` 提取 TI 标志
16	0x40	24020001	`addiu $2, $0, 1` `$2=1`
17	0x44	10220003	`beq $1, $2, 3` TI=1 跳到 0x54
18	0x48	8C020004	`lw $2, 4($0)` 其他中断：恢复 `$2`
19	0x4C	8C010000	`lw $1, 0($0)` 其他中断：恢复 `$1`
20	0x50	42000018	`eret` 其他中断返回
21	0x54	4001B800	`mfc0 $1, $11` 定时器：读 COMPARE 寄存器
22	0x58	4081B800	`mtc0 $1, $11` 写回 COMPARE 清 TI 位
23	0x5C	2401009C	`addiu $1, $0, 0x009C` 设置 EPC=0x9C
24	0x60	40817000	`mtc0 $1, $14` 写入 EPC
25	0x64	8C020004	`lw $2, 4($0)` 恢复 `$2`
26	0x68	8C010000	`lw $1, 0($0)` 恢复 `$1`
27	0x6C	42000018	`eret` 定时器中断返回
28	0x70	24018001	`addiu $1, $0, 0x8001` 设置 STATUS：IM7=1, IE=1
29	0x74	40816000	`mtc0 $1, $12` 写 STATUS
30	0x78	24010000	`addiu $1, $0, 0` COUNT=0
31	0x7C	40819000	`mtc0 $1, $9` 写 COUNT
32	0x80	24010005	`addiu $1, $0, 5` COMPARE=5
33	0x84	4081B800	`mtc0 $1, $11` 写 COMPARE
34	0x88	24010001	`addiu $1, $0, 1` `$1=1`
35	0x8C	24020002	`addiu $2, $0, 2` `$2=2`
36	0x90	24030000	`addiu $3, $0, 0` `$3=0`（计数器）
37	0x94	24630001	`addiu $3, $3, 1` 循环自增
38	0x98	08000025	`j 0x94` 等待中断（死循环）
39	0x9C	00411821	`addu $3, $2, $1` 中断返回后执行，加法运算
40	0xA0	AC030008	`sw $3, 8($0)` 存储结果
41	0xA4	08000029	`j 0xA4` 程序结束（死循环）

在定时器中断的仿真波形中，可以看到 `CP0_EPC` 的值是 `0x00000078`。最开始我觉得中断应该

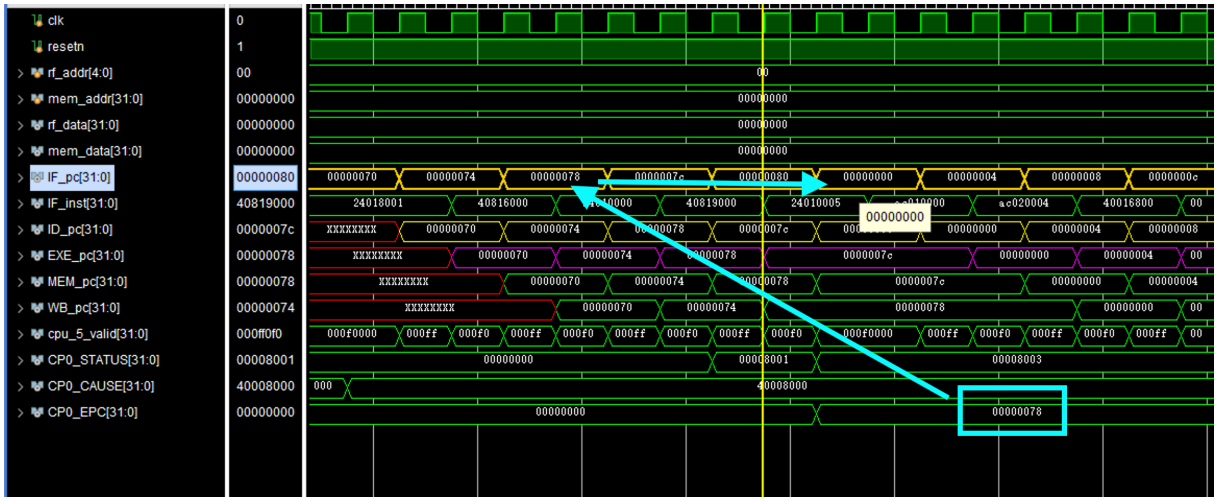

图 3.3: 定时器中断触发波形

在主程序的等待循环（比如 0x94 一带）发生，所以 EPC 应该是 0x94 才“正常”。后来详细分析后明白，EPC 为 0x78 是合理的。

这里需要注意一点：EPC 不是“我觉得中断大概是在某个位置触发”的那个地址，而是**硬件在检测到中断时，流水线里正在提交（WB 级）的那条指令的 PC**。定时器中断的条件是 `COUNT == COMPARE`，这个比较是在 CP0 里面按时钟节拍做的，什么时候这两个值相等，就在那一拍把 TI 位置 1，并把当时 WB 级里的 PC 送进 EPC。

本实验的主程序大致顺序是：

- 0x70 左右：配置 STATUS，打开中断；
- 0x78、0x7C、0x80、0x84：初始化 COUNT 和 COMPARE；
- 之后进入 0x94 的死循环，等定时器中断；

当 `COUNT == COMPARE` 的那一拍到来时，流水线里的指令已经“排队”流动了一段时间，并不保证此时 WB 级正好是 0x94 那条指令。本次仿真里，触发中断的时候，WB 级的 PC 恰好是 0x78，所以 EPC 里就记下了 0x78，这在时序上是说得通的。

4 实验总结

1. **CP0 模块设计**：成功实现了 CP0 模块，包含 6 个 CP0 寄存器（STATUS、CAUSE、EPC、BADVADDR、COUNT、COMPARE），每个寄存器都有明确的位域定义和访问控制机制。
2. **异常处理机制**：实现了统一的异常处理流程，通过 WB 级异常仲裁确定异常优先级，所有异常都通过统一的异常总线传递给 CP0，由 CP0 统一处理。异常发生时，CP0 自动保存 EPC、设置 EXL 位、写入 CAUSE 寄存器等。
3. **异常类型实现**：成功实现了 4 种异常类型：
    - SYSCALL 异常：在 ID 级检测，异常码为 8

- BREAK 异常：在 ID 级检测，异常码为 9
- 地址错误异常（AdEL/AdES）：在 MEM 级检测，异常码为 4/5
- 定时器中断：在 CP0 内部检测，异常码为 0

4. **中断处理**：实现了定时器中断机制，包括 COUNT/COMPARE 寄存器的实现、定时器中断检测逻辑、中断使能条件判断等。定时器中断可以在满足条件时异步触发，优先级最高。

5. **总线设计**：设计了 MEM->WB 总线和 WB->CP0 异常总线，实现了异常信息在流水线中的正确传递。总线宽度为 153 位，包含了所有必要的异常信号。

6. **寄存器访问**：实现了 MTC0/MFC0 指令，支持 CP0 寄存器的读写访问。使用写掩码（WMASK）控制可写位域，确保寄存器的安全性。

7. **流水线控制**：实现了 cancel、exc_valid、exc_pc 等流水线控制信号，确保异常发生时流水线能够正确响应，取消已取出的指令，跳转到异常入口或 ERET 返回地址。

8. **设计规范**：整个实现遵循 MIPS 架构规范，提供了完整的异常处理和中断支持，为系统软件提供了可靠的异常处理机制。