MIPS 五级流水线 CPU 的 CP0 异常处理系统设计与实现

姓名：姓名　**专业**：专业

目录

摘要

CP0 模块实现部分：本文档详细描述了 MIPS 五级流水线 CPU 中协处理器 0（CP0）的设计与实现。CP0 是 MIPS 架构中用于系统控制和异常处理的关键模块，负责处理各种异常类型（如 SYSCALL、BREAK、地址错误等）和中断（如定时器中断），并维护系统状态寄存器。本文档涵盖了 CP0 模块的总体架构、6 个 CP0 寄存器的实现细节、异常处理流程和仲裁机制、4 种异常类型的实现、总线排布和信号传递、CP0 寄存器访问机制以及流水线控制信号等内容。

软件异常处理程序部分：本章节预留，用于描述异常处理程序的编写方法、测试程序的设计思路等。具体内容待补充。

关键词： MIPS、五级流水线、CP0、异常处理、中断、协处理器

1 实验目的

1. 理解 MIPS 架构中 CP0（协处理器 0）的作用和功能
2. 掌握 CP0 寄存器的设计和实现方法
3. 学习异常处理机制的设计与实现
4. 理解中断处理流程和定时器中断的实现
5. 掌握流水线中异常信号的传递和仲裁机制

2 实验要求

1. 实现 CP0 模块，包含 STATUS、CAUSE、EPC、BADVADDR、COUNT、COMPARE 等 6 个寄存器
2. 实现 SYSCALL、BREAK、地址错误（AdEL/AdES）、定时器中断等 4 种异常类型
3. 实现异常仲裁机制，确定异常优先级
4. 实现 MTC0/MFC0 指令，支持 CP0 寄存器访问
5. 实现 ERET 指令，支持从异常处理返回

3 实验过程

3.1 CP0 模块总体架构

3.1.1 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
);
```

3.1.2 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。

3.2 CP0 寄存器实现

3.2.1 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

3.2.2 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
```

3.2.3 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)

3.2.4 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。

3.2.5 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。

3.2.6 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。

3.2.7 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。

3.3 异常处理流程

3.3.1 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。

3.3.2 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

3.3.3 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
```

3.3.4 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。

3.4 中断处理

3.4.1 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
```

3.4.2 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）

3.5 总线排布

3.5.1 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
```

```
2
3  // 扩展MEM->WB总线，新增：mem_ex_adel, mem_ex_ades, badvaddr(dm_addr)
4  assign MEM_WB_bus = {rf_wen,rf_wdest,           // WB需要使用的信号
5                       mem_result,                // 最终要写回寄存器的数据
6                       lo_result,                 // 乘法低32位结果
7                       hi_write,lo_write,         // HI/LO写使能
8                       mfhi,mflo,                 // WB需要使用的信号
9                       mtc0,mfc0,cp0r_addr,syscall,brk,eret, // WB需要使用的信号
10                      mem_ex_adel, mem_ex_ades,  // 地址异常标志（新增）
11                      dm_addr,                   // BADVADDR（新增）
12                      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

3.5.2 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

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

3.6 各异常类型的实现

3.6.1 SYSCALL 异常

检测位置：ID 级（指令译码）

Listing 16: SYSCALL 指令检测

```
1  // decode.v
2  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。

3.6.2 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。

3.6.3 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。

3.6.4 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。

3.7 CP0 寄存器访问

3.7.1 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。

3.7.2 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。

3.8 流水线控制信号

3.8.1 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。

3.8.2 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）。

3.9 软件异常处理程序

注意：本章节预留，用于描述异常处理程序的编写方法、测试程序的设计思路等。具体内容待补充。

3.9.1 异常处理程序框架

（待补充）

3.9.2 测试程序设计

（待补充）

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。