浙江大学

本科实验报告

课程名称：计算机组成

姓　　名：应周骏

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3200103894

指导教师：马德

2022 年 5 月 21 日

浙江大学实验报告

课程名称：计算机组成 实验类型：综合

实验项目名称：CPU 设计之中断

学生姓名：应周骏 专业：计算机科学与技术 学号：3200103894

同组学生姓名：无 指导老师：马德

实验地点：东 4-509 实验日期：2022 年 4 月 25 日

一、实验目的和要求

1. 深入理解 CPU 结构

3. 学习如何提高 CPU 使用效率

3. 学习 CPU 中断工作原理

4. 设计中断测试程序

二、实验内容和原理

目标：

熟悉 RISC-V 中断的原理，了解引起 CPU 中断产生的原因及其处理方法，扩展包含中断的 CPU。

内容：

扩展实验 CPU 中断功能并设计测试。

原理：

1. 中断概念

中断是指程序执行过程中，当发生某个事件时，中止 CPU 上现行程序的运行，引出处理该事件的程序执行的过程，此过程都需要打断处理器正常的工作，为此，才提出了“中断”的概念。

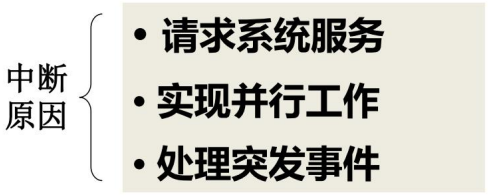


图 2.1 中断原因

在中断过程中，有如下几个概念：

• 中断源：引起中断的事件称为中断源；

• 中断请求：中断源向 CPU 提出处理的请求；

• 断点：发生中断时被打断程序的暂停点；

• 中断响应：CPU 暂停现行程序而转为响应中断请求的过程；

• 中断处理程序：处理中断源的程序；

• 中断处理：CPU 执行有关的中断处理程序；

• 中断返回：返回断点的过程；

2. 中断的分类与作用

按照中断信号的来源，可把中断分为外中断和内中断两类：

• 外中断（又称中断）：指来自处理器和主存之外的中断；

• 内中断（又称异常）：指来自处理器和主存内部的中断；

中断处理程序：保护 CPU 现场、处理发生的中断事件、恢复正常操作。

3. 中断（异常）处理过程

当 CPU 收到中断或者异常的信号时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个信号的相关处理程序中，在完成对这个信号的处理后再跳回到刚才被打断的程序或任务中。

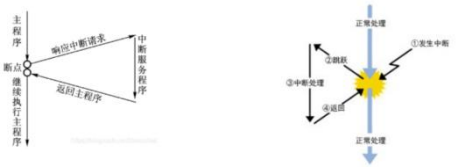


图 2.2 中断处理图解

4. 简化中断设计

①外部中断（Int）触发中断或非法指令（illegial）触发异常或 ecall 系统调用；

②响应 mtvec 寄存器定义的 PC 值分别针对 Int 为 0x0c；ecall 为 0x08；illegical 为 0x04；

③mepc 寄存器值更新为下一条指令的 PC 值；

④执行异常服务程序；

⑤执行 mret 指令，返回 mepc 保存的 PC 处继续程序流；

向量地址	ARM异常名称	ARM系统工作模式	本实验定义
0x0000000	复位	超级用户Svc	复位
0x0000004	未定义指令终止	未定义指令终止Und	非法指令异常
0x0000008	软中断（SWI）	超级用户Svc	ECALL
0x000000c	Prefetch abort	指令预取终止Abt	Int外部中断（硬件）
0x0000010	Data abort	数据访问终止Abt	Reserved自定义
0x0000014	Reserved	Reserved	Reserved自定义
0x0000018	IRQ	外部中断模式IRQ	Reserved自定义
0x000001C	FIQ	快速中断模式FIQ	Reserved自定义

图 2.3 ARM 中断向量表

三、实验过程和数据记录

1. 工程文件建立

新建工程文件，命名为“OxExp04_Interrupt”，“OExp02-IP2SOC_inter”。

2. SCPU 模块更新

在实验四扩展指令的“SCPU. v”文件基础上，修改部分代码。

```
module SCPU(
    input clk,
    input rst,
    input MIO_ready,
    input [31:0]inst_in,
    input [31:0]Data_in,
    input INT0,

    output CPU_MIO,
    output MemRW,
    output [31:0]PC_out,
    output [31:0]Data_out,
    output [31:0]Addr_out
    );

    wire [3:0]ALU_Control;
    wire [2:0]ImmSel;
    wire [1:0]MemtoReg;
    wire [1:0]Jump;
    wire Branch, BranchN, RegWrite, ALUSrc_B;
    wire ecall, mret, ill_instr;
```

图 3.2.1 SCPU 接口

```
SCPU_ctrl_int U1(
.OPcode(inst_in[6:2]),
.Fun3(inst_in[14:12]),
.Fun7(inst_in[30]),
.MIO_ready(MIO_ready),
.Fun_ecall(inst_in[22:20]),
.Fun_mret(inst_in[29:28]),
.ImmSel(ImmSel),
.ALUSrc_B(ALUSrc_B),
.MemtoReg(MemtoReg),
.Jump(Jump),
.Branch(Branch),
.BranchN(BranchN),
.RegWrite(RegWrite),
.MemRW(MemRW),
.ALU_Control(ALU_Control),
.CPU_MIO(CPU_MIO),
.ecall(ecall),
.ill_instr(ill_instr),
.mret(mret)
);
```

```
DataPath_int U0(
    .ALUSrc_B(ALUSrc_B),
    .ALU_operation(ALU_Control),
    .Branch(Branch),
    .BranchN(BranchN),
    .Data_in(Data_in),
    .ImmSel(ImmSel),
    .Jump(Jump),
    .MemtoReg(MemtoReg),
    .RegWrite(RegWrite),
    .clk(clk),
    .inst_field(inst_in),
    .rst(rst),
    .INT0(INT0),
    .ecall(ecall),
    .ill_instr(ill_instr),
    .mret(mret),
    .ALU_out(Addr_out),
    .Data_out(Data_out),
    .PC_out(PC_out)
);
```

图 3.2.2 连接数据通路和控制通路

3. 数据通路模块更新

在实验 4 的“Datapath.v”基础上，添加中断信号及中断模块。

```
module DataPath_int(
    input wire clk,
    input wire rst,
    input wire[31:0] inst_field,
    input wire[31:0] Data_in,
    input wire[3:0] ALU_operation,
    input wire[2:0] ImmSel,
    input wire[1:0] MemtoReg,
    input wire ALUSrc_B,
    input wire[1:0]Jump,
    input wire Branch,
    input wire BranchN,
    input wire RegWrite,
    input wire INT0,
    input wire ecall,
    input wire mret,
    input wire ill_instr,

    output wire[31:0] PC_out,
    output wire[31:0] Data_out,
    output wire[31:0] ALU_out
    );
```

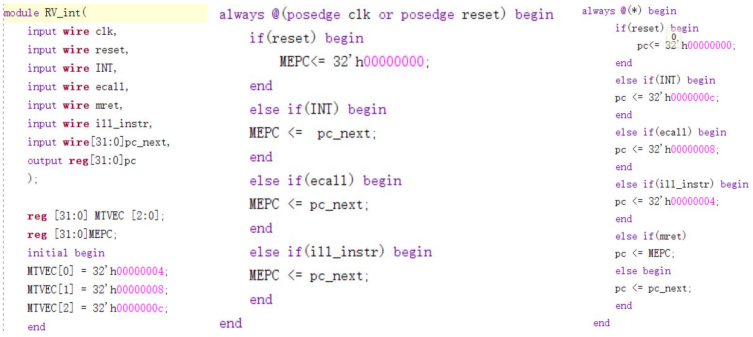
图 3.3.1 数据通路接口

```
RV_int RV_int_0(
    .clk(clk),
    .reset(rst),
    .INT(INT0),
    .ecall(ecall),
    .mret(mret),
    .ill_instr(ill_instr),
    .pc_next(PC_new_2),
    .pc(PC_new)
);
```

图 3.3.2 添加中断控制模块

4. RV_int 模块设计

新建“RV_int.v”模块，输入对应代码，以实现对中断的 PC 控制。



```
module RV_int(
    input wire clk,
    input wire reset,
    input wire INT,
    input wire ecall,
    input wire mret,
    input wire ill_instr,
    input wire[31:0]pc_next,
    output reg[31:0]pc
    );

    reg [31:0] MTVEC [2:0];
    reg [31:0]MEPC;
    initial begin
    MTVEC[0] = 32'h00000004;
    MTVEC[1] = 32'h00000008;
    MTVEC[2] = 32'h0000000c;
    end
```

```
always @(posedge clk or posedge reset) begin
    if(reset) begin
        MEPC<= 32'h00000000;
    end
    else if(INT) begin
    MEPC <=  pc_next;
    end
    else if(ecall) begin
    MEPC <= pc_next;
    end
    else if(ill_instr) begin
    MEPC <= pc_next;
    end
end
```

```
always @(*) begin
    if(reset) begin
        pc<= 32'h00000000;
    end
    else if(INT) begin
    pc <= 32'h0000000c;
    end
    else if(ecall) begin
    pc <= 32'h00000008;
    end
    else if(ill_instr) begin
    pc <= 32'h00000004;
    end
    else if(mret)
    pc <= MEPC;
    else begin
    pc <= pc_next;
    end
end
```

图 3.4.1 中断模块代码

对该文件设计对应仿真代码，代码如下。

```
always  #10 clk = ~clk;
initial begin
clk = 0;
reset = 1;
INT = 0;
ecall = 0;
mret = 0;
ill_instr = 0;
pc_next = 32'h10;
#100;
reset = 0;
INT = 1;
#100;
INT = 0;
pc_next = 32'h20;
#100;
pc_next = 32'h24;
#100;
mret = 1;
end
```

图 3.4.2 中断模块仿真代码

仿真结果如下。

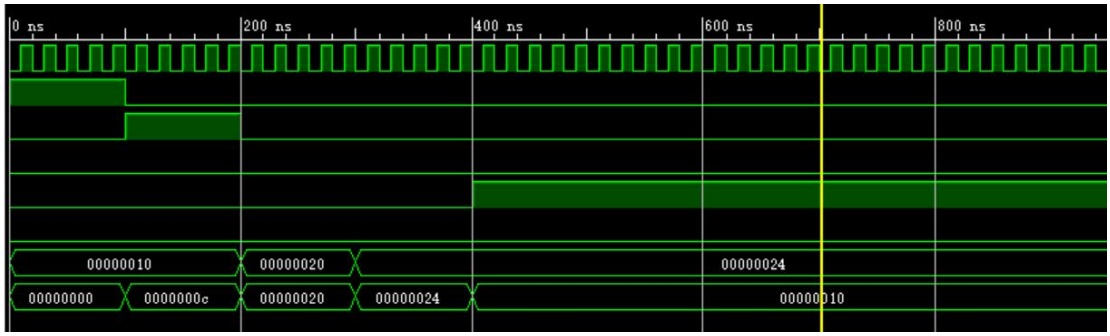


图 3.4.3 中断模块仿真结果

5. 控制通路模块设计

在实验 4 的“SCPU_ctrl.v”基础上，添加中断信号。

```
module SCPU_ctrl_int(
input[4:0]OPcode,   //Opcode------inst[6:2]
input[2:0]Fun3,     //Function-----inst[14:12]
input Fun7,         //Function-----inst[30]
input MIO_ready,    //CPU Wait
input wire[2:0]Fun_ecall,
input wire[1:0]Fun_mret,
output reg [2:0]ImmSel,
output reg ALUSrc_B,
output reg [1:0]MemtoReg,
output reg [1:0]Jump,//jal
output reg Branch,//beq
output reg BranchN,
output reg RegWrite,
output reg MemRW,
output reg [3:0]ALU_Control, //alu
output reg CPU_MIO,//not use
output reg ecall,
output reg mret,
output reg ill_instr
);

reg [1:0] ALUop;
wire [3:0] Fun;
```

图 3.5.1 控制通路接口

```
always @* begin
ecall <= 1'b0;
mret <= 1'b0;
ill_instr <= 1'b0;
case(OPcode)
5'b01100: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_00_1_0_0_0_00_10_000; end //ALU
5'b00000: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_01_1_0_0_0_00_00_001; end //load
5'b01000: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_00_0_1_0_0_00_00_010; end //store
5'b11000: begin
case(Fun3)
3'b000: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_00_0_0_1_0_00_01_011; end
3'b001: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_00_0_0_0_1_00_01_011; end
default: ill_instr <= 1'b1;
endcase //beq
end
5'b11011: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_10_1_0_0_0_01_00_101; end //jump
5'b00100: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_00_1_0_0_0_00_11_001; end //ALU(addi;;;;)
5'b11001: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_10_1_0_0_0_10_00_001; end //jarl
5'b01101: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_11_1_0_0_0_00_00_000; end //lui
5'b11100: begin
{ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_0_0_0_0_0_00_00_000;
if(Fun_mret == 2'b00 && Fun_ecall == 3'b000) ecall <= 1'b1;
else if(Fun_mret == 2'b11 & Fun_ecall == 3'b010) mret <= 1'b1; //may be the same??
else ill_instr <= 1'b1;
end
default: begin  ill_instr <= 1'b1; end
endcase
end
```

图 3.5.2 控制通路信号生成

```
assign Fun = {Fun3,Fun7};
always @* begin
case(ALUop)
2'b00: ALU_Control = 4'b0010 ; //add
2'b01: ALU_Control = 4'b0110 ; //sub
2'b10:
case(Fun)
4'b0000: ALU_Control = 4'b0010 ; //add
4'b0001: ALU_Control = 4'b0110 ; //sub
4'b1110: ALU_Control = 4'b0000 ; //and
4'b1100: ALU_Control = 4'b0001 ; //or
4'b0100: ALU_Control = 4'b0111 ; //slt
4'b1010: ALU_Control = 4'b0101 ; //srl
4'b1000: ALU_Control = 4'b0011 ; //xor
4'b0110: ALU_Control = 4'b1010 ; //sltu
4'b0010: ALU_Control = 4'b1001 ; //sll
4'b1011: ALU_Control = 4'b1000 ; //sra
default: ALU_Control = 4'bx;  //nor(no this kind)
endcase
2'b11:
case(Fun3)
3'b000: ALU_Control = 4'b0010 ; //addi
3'b010: ALU_Control = 4'b0111 ; //slti
3'b100: ALU_Control = 4'b0011 ; //xori
3'b110: ALU_Control = 4'b0001 ; //ori
3'b111: ALU_Control = 4'b0000 ; //andi
3'b101:
    if(Fun7)
        ALU_Control = 4'b1000 ; //srai
    else
        ALU_Control = 4'b0101 ; //srli
3'b011: ALU_Control = 4'b1010 ; //sltiu
3'b001: ALU_Control = 4'b1001 ; //slli
default: ALU_Control = 4'bx;  //nor(no this kind)
endcase
default: ALU_Control = 4'bx;
endcase
end
endmodule
```

图 3.5.3 控制通路 ALU 两级译码

6. 测试模块接口调整

在实验二测试框架基础上，在接口处加入按钮控制的外部中断。

```
SCPU U1
      (.Addr_out(Addr_out),
       .Data_in(Data_in),
       .Data_out(Data_out),
       .INT0(BTN_OK[1]),
       .MIO_ready(1'b0),
       .MemRW(MemRW),
       .PC_out(PC_out),
       .clk(Clk_CPU),
       .inst_in(Inst_in),
       .rst(rst));
```

图 3.6.1 外部中断接口

7. 物理验证

对上述扩展指令生成 bit 文件，在 SWORD 实验板上进行验证。

四、实验结果分析

1. RV_int 功能分析

该模块主要响应各类中断信号的 PC 变化，通过时序逻辑控制 MEPC 寄存器，并通过组合逻辑实现响应中断对 PC 的影响。

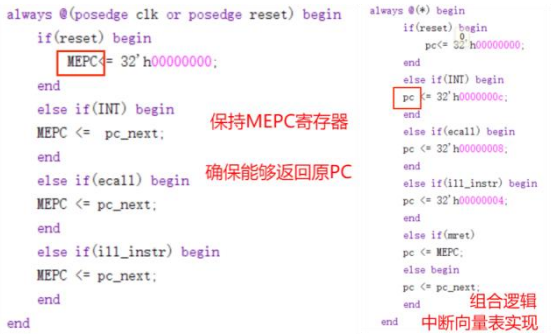


图 4.1 RV_int 功能

2. 中断信号生成

控制通路中，增加对中断信号的响应，包括无效指令和 ecall/外部中断。

```
always @* begin
ecall <= 1'b0;
mret <= 1'b0;
ill_instr <= 1'b0;
case(OPcode)
5'b01100: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_00_1_0_0_0_00_10_000; end //ALU
5'b00000: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_01_1_0_0_0_00_00_001; end //load
5'b01000: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_00_0_1_0_0_00_00_010; end //store
5'b11000: begin
case(Fun3)
3'b000: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_00_0_0_1_0_00_01_011; end
3'b001: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_00_0_0_0_1_00_01_011; end
default: ill_instr <= 1'b1;
endcase //beq
end
5'b11011: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_10_1_0_0_0_01_00_101; end //jump
5'b00100: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_00_1_0_0_0_00_11_001; end //ALU(addi;;;;)
5'b11001: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b1_10_1_0_0_0_10_00_001; end //jarl
5'b01101: begin {ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_11_1_0_0_0_00_00_000; end //lui
5'b11100: begin
{ALUSrc_B,MemtoReg,RegWrite,MemRW,Branch,BranchN,Jump,ALUop,ImmSel}  = 14'b0_0_0_0_0_0_00_00_000;
if(Fun_mret == 2'b00 && Fun_ecall == 3'b000) ecall <= 1'b1;
else if(Fun_mret == 2'b11 & Fun_ecall == 3'b010) mret <= 1'b1; //may be the same??
else ill_instr <= 1'b1;
end
default: begin  ill_instr <= 1'b1; end
endcase
end
```

非法指令

ecall/INT

图 4.2 控制通路产生中断信号

3. 物理验证结果

测试程序为三种中断情况，运行能够进入中断。

图 4.3.1 按钮导致的外部中断

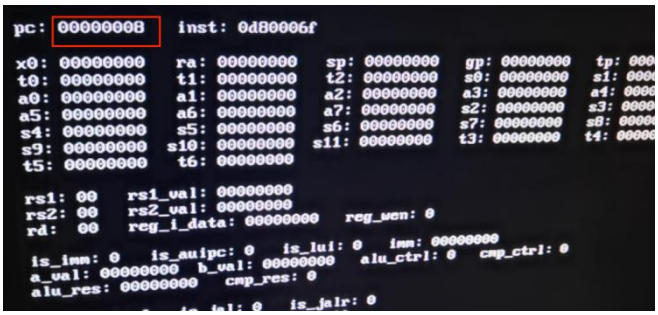


图 4.3.2 ecall 型中断

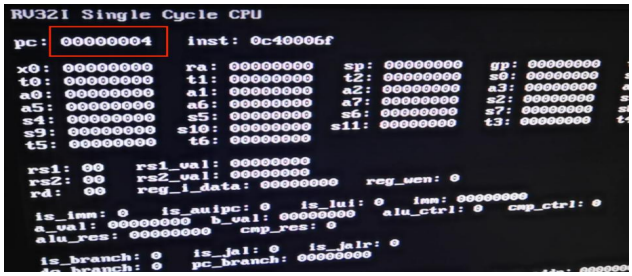


图 4.3.3 非法指令中断

五、讨论与心得

1. 通过本次实验，我掌握了中断的概念，基本了解了 CPU 执行中断的流程。

2. 本次实验中，在中断信号控制过程中，我起初没有厘清中断信号的格式，导致在指令译码时，产生了对非法指令等处理有误的情况，后来通过进一步校对指令格式以及各指令概念，最终实现了中断信号的准确生成。

3. 在设计 RV_int 的过程中，最初由于时序逻辑与组合逻辑的混淆，导致在实际测试时，出现了中断后 PC 停止、PC 时钟加快等一系列问题，这也提示我此后实验中，要注意 Verilog 语言设计时的逻辑选择。