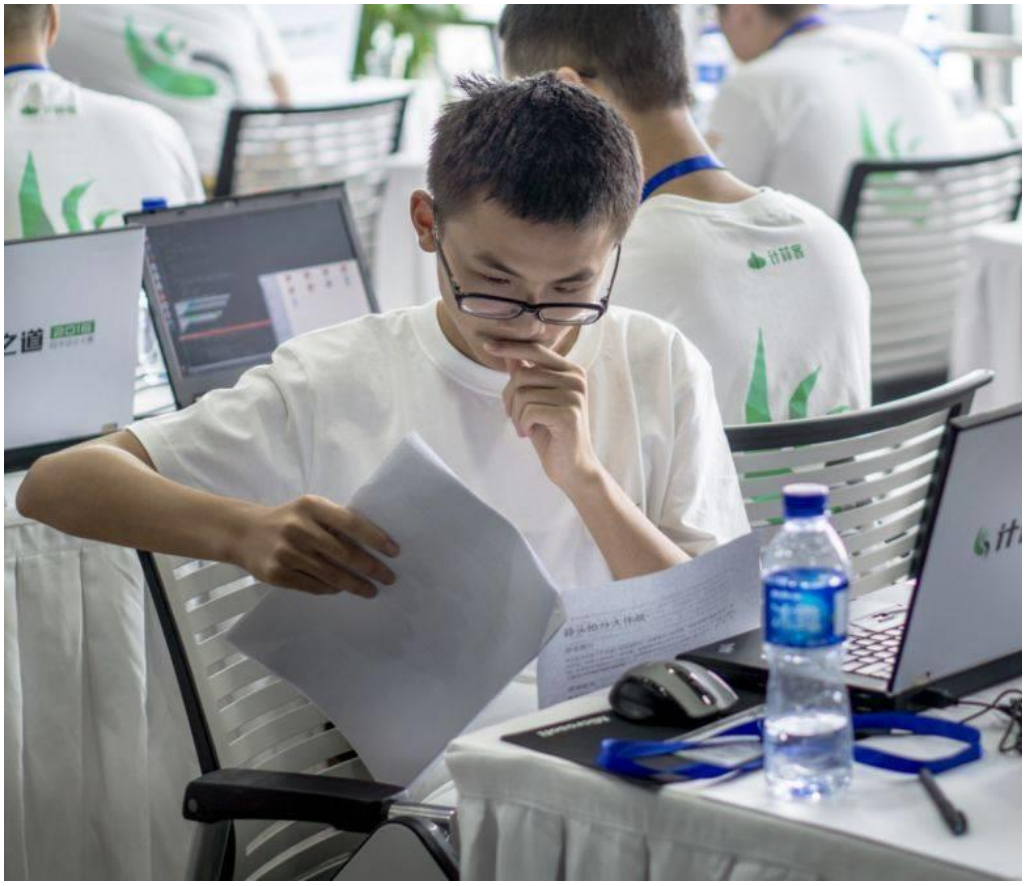
实验七 CPU 设计-指令集扩展实验报告

姓名：蒋仕彪 学号：3170102587 专业：求是科学班（计算机）1701

课程名称：计算机组成实验 同组学生姓名：

实验时间：2019-4-16 实验地点：紫金港东 4-509 指导老师：马德

个人形象照：



一、实验目的和要求

1. 运用寄存器传输控制技术
2. 掌握 CPU 的核心：指令执行过程与控制流关系
3. 设计数据通路和控制器
4. 设计测试程序

二、实验任务和原理

2.1 实验任务

1. 扩展实验六 CPU 指令集
    - 重新设计数据通路和控制器
        - 兼容 Exp05 的数据通路和控制器
        - 替换 Exp05 的数据通路控制器核
    - 扩展不少于下列指令

    R-Type：add, sub, and, or, xor, nor, slt, srl*, jr, jalr, eret；

    I-Type： addi, andi, ori, xori, lui, lw, sw, beq, bne, slti

    J-Type： J, Jal*；
    - 此实验在 Exp06 的基础上完成
2. 设计指令集测试方案
3. 设计指令集测试程序

2.2 实验原理

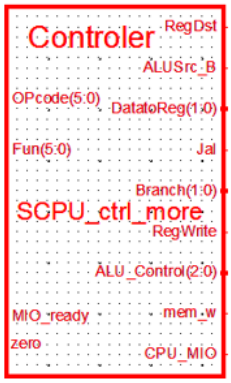
2.2.1 控制信号定义

兼容Exp06需要增加那些通路与操作控制

信号	源数目	功能定义	赋值0时动作	赋值1时动作
ALUSrc_B	2	ALU端口B输入选择	选择寄存器B数据	选择32位立即数（符号扩展后）
RegDst	2	寄存器写地址选择	选择指令rt域	选择指令rs域
MemtoReg	2	寄存器写入数据选择	选择存储器数据	选择ALU输出
Branch	2	Beq指令目标地址选择	选择PC+4地址	选择转移地址（Zero=1）
Jump	2	J指令目标地址选择	选择J目标地址	由Branch决定输出
RegWrite	-	寄存器写控制	禁止寄存器写	使能寄存器写
MemWrite	-	存储器写控制	禁止存储器写	使能存储器写
MemRead	-	存储器读控制	禁止存储器读	使能存储器读
ALU_Control	000-111	3位ALU操作控制	参考表 Exp04	Exp04

2.2.2 重新设计数据通路与控制器接口

- 重新设计接口
  - 扩展后增加了控制信号
  - 数据通路参考接口如右图
    - 模块符号文档：Dataa_path_more.sym
  - 控制器参考接口信号如下图
    - 模块符号文档：SCPU_ctrl_more.sym




```
module    SCPU_ctrl_more( input[5:0]OPcode,          //OPcode
                          input[5:0]Fun,             //Function
                          input MIO_ready,           //CPU Wait
                    ……
                    output reg mem_w,
                    output reg [2:0]ALU_Control,
                    output reg CPU_MIO
                    );
endmodule
```

```
module    Data_path_more( input clk,                 //寄存器时钟
                          input rst,                 //寄存器复位
                          input[25:0]inst_field,     //指令数据域
                          ……
                          output[31:0]ALU_out,
                          output[31:0]Data_out,
                          output[31:0]PC_out
                          );
endmodule
```

三、主要仪器设备

1. 计算机（Intel Core i5 以上，4GB 内存以上）系统
2. Spartan-3 Starter Kit Board/Sword 开发板
3. Xilinx ISE14.4 及以上开发工具

四、操作方法与实验步骤

4.1 设计工程

◎ **扩展不少于下列指令**

R-Type：add, sub, and, or, xor, nor, slt, srl*, jr, jalr, eret；

I-Type： addi, andi, ori, xori, lui, lw, sw, beq, bne, slti

J-Type： J, Jal*；

◎ **集成替换验证通过的新 CPU**

- 替换实验六(Exp06)中的 SCPU 模块
- 替换实验六(Exp06)中的 SCPU_ctrl 模块
- 替换实验六(Exp06)中的 Data_Path 模块
- 顶层模块延用 Exp05
  - 模块名：Top_OExp06_ExtSCPU.sch
  - 需要修改 CPU 逻辑符号

◎ **测试扩展后的 CPU 功能**

- 设计测试程序(MIPS 汇编)测试

4.2 设计要点

◎ **设计指令扩展后 DataPath 结构**

- 需要根据新的接口信号重新设计逻辑符号
- 在实验五的原理图上扩展

◎ **根据新 DataPath 结构设计控制器**

- 需要根据新的接口信号重新设计逻辑符号
- 建议用 HDL 结构化描述

◎ **设计 CPU 调用模块**

- 根据新的控制器和数据通路接口信号设计 CPU 模块
- 重新设计 CPU 逻辑符号

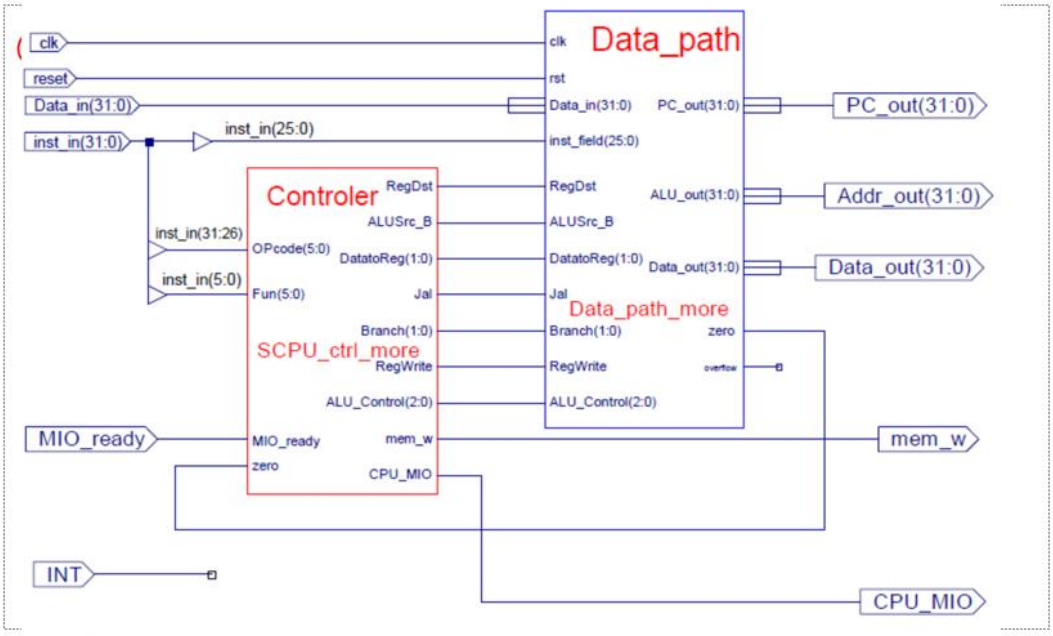
◎ **仿真新设计的模块**

- 独立仿真 DataPath 和控制器

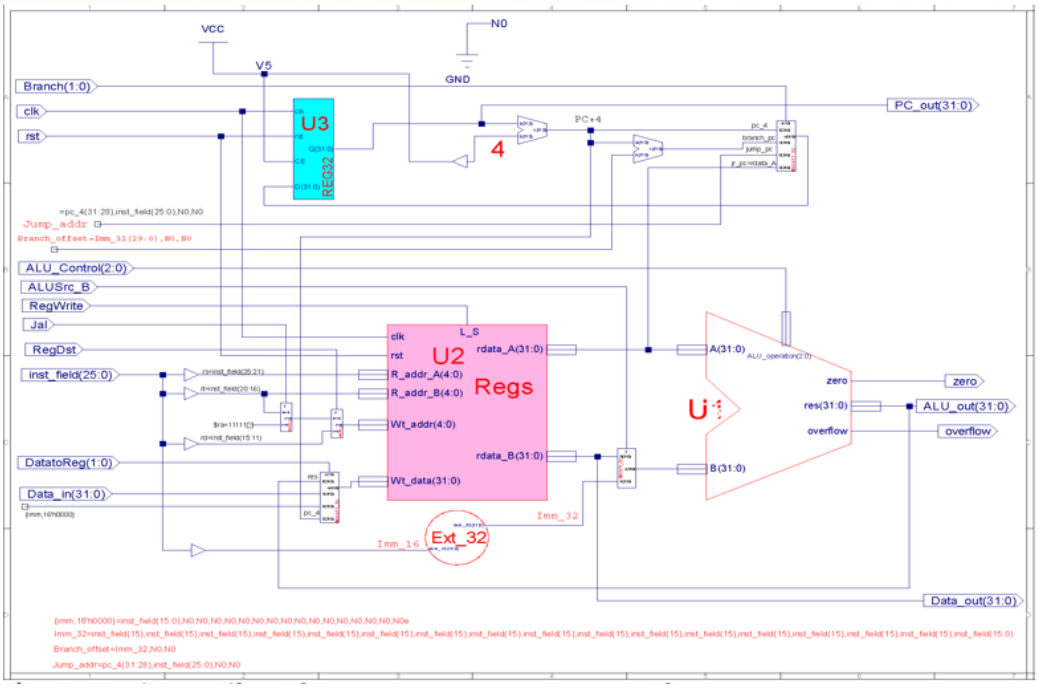
◎ **集成替换 CPU 及子模块**

- 仿真正确后
  - 集成替换 CPU、数据通路和控制器模块

4.2 扩展指令后的 CPU 参考模块



4.3 Datapath 参考设计



4.4 控制器描述参考结构

```
`define CPU_ctrl_signals
{RegDst,ALUSrc_B,MemtoReg,RegWrite,MemRead,MemWrite,Branch,Jump, ALU_Control，…
        ……}
        assign mem_w = MemWrite&&(~MemRead);
        always @* begin
                case(OPcode)
                6'b000000:                                              //ALU
                  case(Fun)
                    6'b100000: begin CPU_ctrl_signals = ?;  end         //add
                    6'b100010: begin CPU_ctrl_signals = ?;  end         //sub
                                        ……
                    default:    begin CPU_ctrl_signals = ?;  end;
                  endcase
                6'b100011: begin CPU_ctrl_signals = ?;  end             //load
                6'b101011: begin CPU_ctrl_signals = ?;  end             //store
                                                        ……
                default:    begin  CPU_ctrl_signals = ?; end
                endcase
        end
```

4.5 CPU 调试与测试

- **调试**
  - SCPU_ctrl_more 模块仿真
    - 设计测试激励代码仿真测试*
  - Data_path_more 模块仿真
    - 设计测试激励代码仿真测试*
- **集成替换**
  - 仿真正确后逐个替换 Exp06 的相应模块
  - 使用 DEMO 程序目测控制器正常运行
    - DEMO 程序与前面实验不一样
    - 也可自行设计

```
memory_initialization_radix=16;
memory_initialization_vector=
08000008, 00000020, 00000020, 00000020, 00000020, 00000020, 00000020, 3c03f000,
3c04e000, 3c088000, 2014003f, 3c06f800, 00000827, 0001102a, 202affff, ac660004, 8c650000,
00a52820, 00a52820, ac650000, 21290001, ac890000, 8c0d0014, 8c650000, 00a52820, 00a52820,
ac650000, 8c650000, 00a85824, 21ad0001, 15a00001, 0c000037, 8c650000, 20120008, 0252b020,
02569020, 00b25824, 11600005, 11720009, 20120008, 1172000a, ac890000, 08000018, 15410002,
00005027, 014a5020, ac8a0000, 08000018, 8e290060, ac890000, 08000018, 8e290020, ac890000,
08000018, 8c0d0014, 014a5020, 354a0001, 22310004, 02348824, 01224820, 15210001, 21290005,
8c650000, 00a55820, 016b5820, ac6b0000, ac660004, 03e00008;
```

五、实验结果与分析

5.1 SCPU_Ctrl_more 代码展示

添加了 PPT 后表格里除 Eret 和 jal 的所有指令。

真值表如下：

	RegDst	ALU_Control	ALUSrc_B	DatatoReg	Jal	Branch	RegWrite	Mem_w	CPU_MIO
ADD	1	010	0	00	0	00	1	0	0
SUB	1	110	0	00	0	00	1	0	0
AND	1	000	0	00	0	00	1	0	0
OR	1	001	0	00	0	00	1	0	0
XOR	1	011	0	00	0	00	1	0	0
NOR	1	100	0	00	0	00	1	0	0
SLT	1	111	0	00	0	00	1	0	0
SRL	1	101	0	00	0	00	1	0	0
ERET	1	010	0	00	0	11	1	0	0
JALR	1	010	0	11	1	11	1	0	0
ADDI	0	010	1	00	0	00	1	0	0
ANDI	0	000	1	00	0	00	1	0	0
ORI	0	001	1	00	0	00	1	0	0
XORI	0	011	1	00	0	00	1	0	0
SLTI	0	111	1	00	0	00	1	0	0
LW	0	010	1	01	0	00	1	0	0
SW	1	010	1	00	0	00	0	1	0
BEQ(EQ)	0	110	0	00	0	01	0	0	0
~EQ	0	110	0	00	0	00	0	0	0
BNE(~EQ)	0	110	0	00	0	01	0	0	0
EQ	0	110	0	00	0	00	0	0	0
JR	1	000	0	00	1	11	0	0	0
J	0	000	0	00	0	10	0	0	0
JAL	0	010	0	11	1	10	1	0	0
LUI	0	010	0	10	0	00	1	0	0

经检验扩展指令 CPU 控制模块译码正确，上传开发板验证新 demo 程序有效

```
odule SCPU_ctrl_more(
                      input[5:0]OPcode,                    //OPcode
                      input[5:0]Fun,                       //Function
                       input MIO_ready,                    //CPU Wait
                       input zero,
                       output reg RegDst,
                       output reg ALUSrc_B,
                       output reg [1:0] DatatoReg,
                       output reg Jal,
                       output reg [1:0] Branch,
                       output reg RegWrite,
                       output reg [2:0]ALU_Control,
                       output reg mem_w,
                       output reg CPU_MIO
                     );    always @*
    begin
        CPU_MIO = 0;
        `define CPU_ctrl_signals
{RegDst,ALU_Control,ALUSrc_B,DatatoReg,Jal,Branch,RegWrite,mem_w}
        case(OPcode)
            6'b000000: begin
                           `CPU_ctrl_signals = 12'b100000000010;
```

```
                        case(Fun)
                          6'b100000: ALU_Control = 3'b010;    //add
                          6'b100010: ALU_Control = 3'b110;    //sub
                          6'b100100: ALU_Control = 3'b000;    //and
                          6'b100101: ALU_Control = 3'b001;    //or
                          6'b010110: ALU_Control = 3'b011;    //xor
                          6'b100111: ALU_Control = 3'b100;    //nor
                          6'b101010: ALU_Control = 3'b111;    //slt
                          6'b000010: ALU_Control = 3'b101;    //srl
                          6'b001000: `CPU_ctrl_signals =
12'b100000011100;  //jr
                          default:   ALU_Control = 3'bxxx;
                        endcase
                    end
            6'b001000: `CPU_ctrl_signals = 12'b001010000010;  //addi
            6'b001100: `CPU_ctrl_signals = 12'b000010000010;  //andi
            6'b001101: `CPU_ctrl_signals = 12'b000110000010; //ori
            6'b001110: `CPU_ctrl_signals = 12'b001110000010; //xori
            6'b001010: `CPU_ctrl_signals = 12'b011110000010; //slti
            6'b100011: `CPU_ctrl_signals = 12'b001010100010; //lw
            6'b101011: `CPU_ctrl_signals = 12'b101010000001; //sw
            6'b000100: `CPU_ctrl_signals = {9'b011000000, zero,2'b00};
            6'b000101: `CPU_ctrl_signals = {9'b011000000,!zero,2'b00};
            6'b000010: `CPU_ctrl_signals = 12'b000000001000;  //j
            6'b000011: `CPU_ctrl_signals = 12'b001001111010;  //jal
            6'b001111: `CPU_ctrl_signals = 12'b001001000010;  //lui
              default: `CPU_ctrl_signals = 12'bxxxxxxxxxxxx;
        endcase
    end
endmodule
```

5.2 data_path_more 代码展示

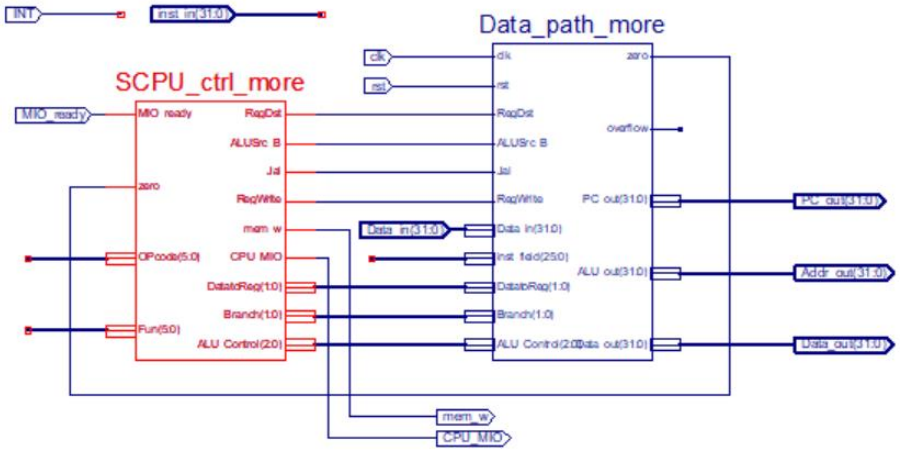
在上一次的 data_path 的代码基础上改。对照给出的参考图，增设了一些节点，并修改了一些 input 和 output 的节点。

```
module   Data_path_more(
           input clk,
           input rst,
           input [31:0]Data_in,
           input [25:0]inst_field,
           input RegDst,
           input ALUSrc_B,
           input [1:0]DatatoReg,
```

```
            input Jal,
            input [1:0]Branch,
            input RegWrite,
            input [2:0]ALU_Control,
            output[31:0]PC_out,
            output[31:0]ALU_out,
            output[31:0]Data_out,
            output zero,
            output overflow
            );
        wire V5,N0;
        assign V5 = 1'b1;
        assign N0 = 1'b0;
        wire [31:0] inD;
        wire [31:0] Jump_addr;
        wire [31:0] Branch_offset;
        wire [31:0] PC_4;
        wire [31:0] RdataA;
        wire [31:0] RdataB;
        wire [4:0]  Prenode;
        wire [31:0] imm32;
        assign PC_4 = PC_out + 32'b100;
        assign Jump_addr = {PC_4[31:28], inst_field[25:0], N0, N0};
        assign Branch_offset = {imm32[29:0], N0, N0};
        assign inD = Branch[1] ? (Branch[0] ? RdataA : Jump_addr) :
(Branch[0] ? PC_4 + Branch_offset : PC_4);
        //(Jump?Jump_addr:((Branch&&zerO)?PC_4+Branch_offset:PC_4));
        REG32 v_PC(.clk(clk), .rst(rst), .CE(V5), .D(inD), .Q(PC_out));
        assign Prenode = Jal ? 5'b11111 : inst_field[20:16];
        regs U22(.clk(clk), .rst(rst), .R_addr_A(inst_field[25:21]),
.R_addr_B(inst_field[20:16]), .Wt_addr(RegDst ? inst_field[15:11] :
Prenode), .Wt_data(DatatoReg[1] ? (DatatoReg[0] ? PC_4 :
{inst_field[15:0], 16'h0000}) : (DatatoReg[0] ? Data_in :
ALU_out)), .L_S(RegWrite), .rdata_A(RdataA), .rdata_B(RdataB));
        assign Data_out = RdataB;
        assign imm32={inst_field[15], inst_field[15], inst_field[15],
inst_field[15], inst_field[15], inst_field[15], inst_field[15],
inst_field[15], inst_field[15], inst_field[15], inst_field[15],
inst_field[15], inst_field[15], inst_field[15], inst_field[15],
inst_field[15], inst_field[15], inst_field[15], inst_field[15],
inst_field[15], inst_field[15:0]};
        ALU U11(.A(RdataA), .B(ALUSrc_B ? imm32 :
RdataB), .ALU_operation(ALU_Control), .zero(zero), .res(ALU_out), .over
flow(overflow));
endmodule
```

5.3 SCPU_more sch 展示

外层的 SCPU 结构简单、清晰，因此继续采用画图的方式。
这样也可以减少错误。



5.4 调试和验证新指令。

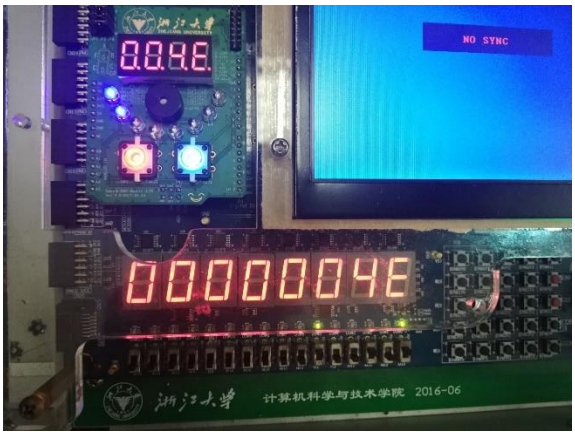
根据 PPT 里的 coe，重新修改 ROM_D 的数据。
Coe 用到了新指令，但是结果应该和原来一样。

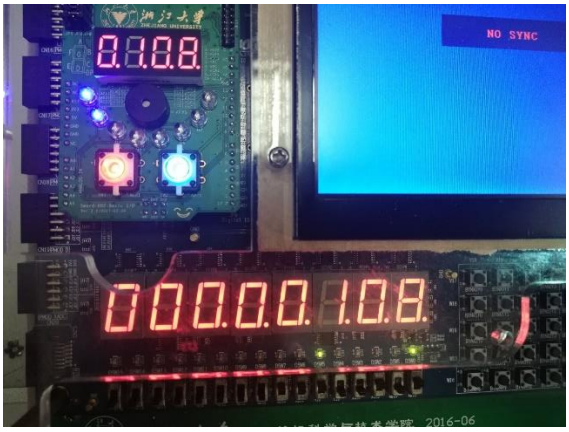
如果跑出来结果不一样，可能是某些指令写错了，这时候我们就要一种一种去验证。我设计了一些很 simple 的指令，先将某个寄存器不断 addi 上 1，然后 j 到某个地方，然后再将调用 ori，xori……J 指令是以前写过的，所以如果 Data_out 的值随着你的预想不断变化的话，说明 addi，ori，xori 等等可以视为是对的。

Beq 和 bnq 比较麻烦一点，感觉用 Data_out 查看也不太方便。如果 beq/bne 条件成立了，我就让 PC 跳到很远的地方去。这样，我每次拨到 111，查询 PC_out。如果 PC_out 在某个地方突然增加好多，可以认为是对的。

5.5 实验现象一

SW[0]=1, SW[2]=1，SW[7:5]=111。
输出 CPU 指令字节地址 PC_out。



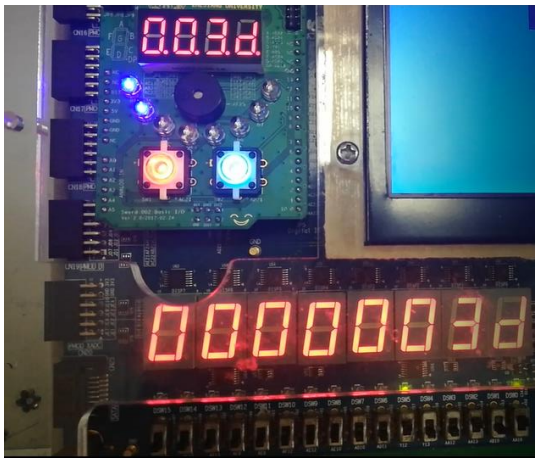


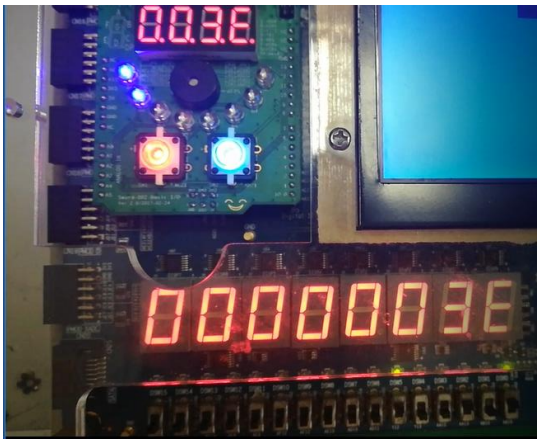
□ 实验现象 1

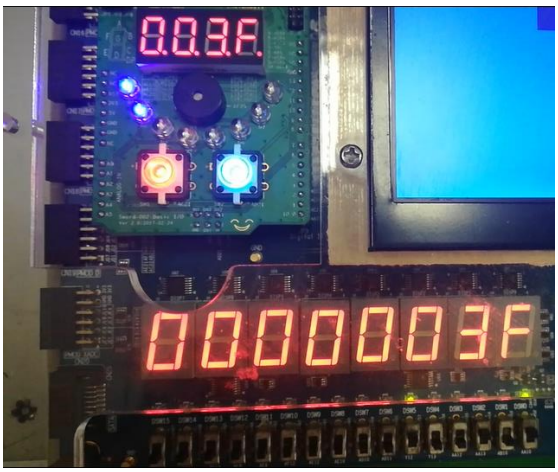
5.6 实验现象二

SW[0]=1, SW[2]=1，SW[7:5]=001。

输出 CPU 指令字地址 PC_out[31:2]。





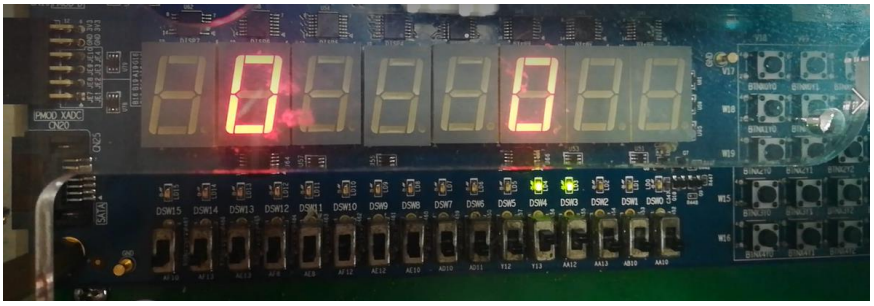


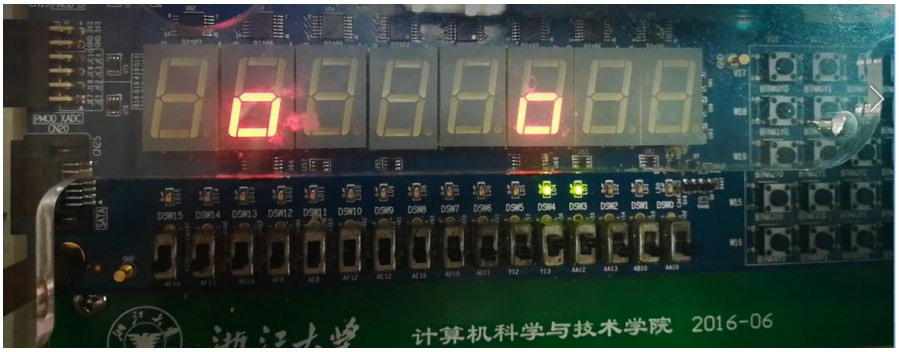
□ 实验现象 2

5.7 实验现象三

SW[0]=0, SW[3]=1，SW[4]=1。

矩形框跳舞。



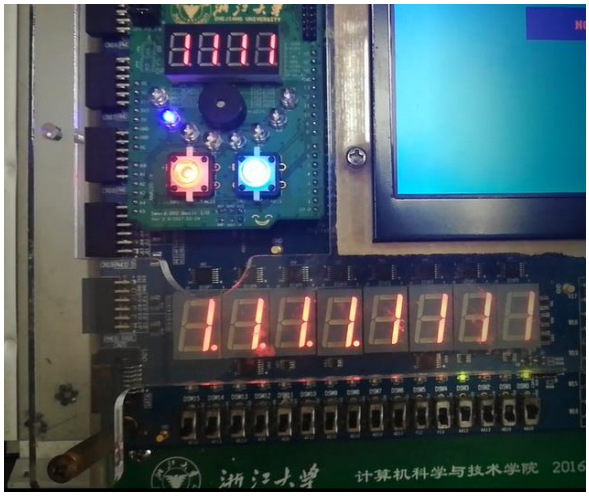


□ 实验现象 3

5.8 实验现象四

SW[0]=1, SW[3]=1。

全速时钟。



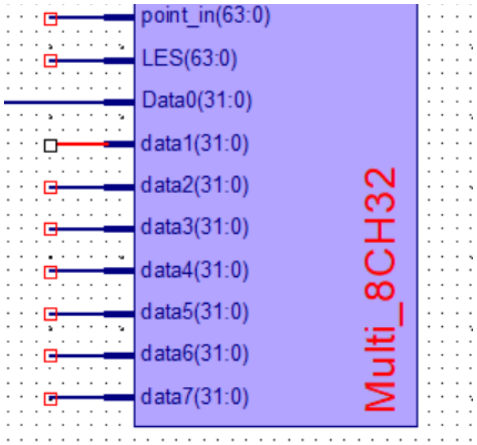
□ 实验现象 4

六、讨论、心得

这次的实验完全拓宽了 SCPU 的可操作性指令书。看上去只是加了几条指令，但是接口什么都有所变化，还要做到能兼容之前的指令，细节还是有点繁琐的。

在这一节里，我真正搞明白了 SW 的调试系统。以前我一直靠死记，SW[7:5]=111 是用来输出 PC_out 的，但一直没弄明白到底是为什么。

后来我发现了 sch 右侧的这个八选一期间。



我点开 data0~data7，突然明白了什么：这其实是对应 SW[7:5]的。也就是说，我拨动 SW 开关，是可以控制七段数码管的输出的。更进一步来说，我可以把这里传进去的值，改成任何我想要的值，从而实现调试的功能。

实验八-- CPU 设计-中断实验报告

姓名：蒋仕彪 学号：3170102587 专业：求是科学班（计算机）1701

课程名称：计算机组成实验 同组学生姓名：

实验时间：2019-5-13 实验地点：紫金港东 4-509 指导老师：马德

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、实验任务和原理

2.1 实验任务

1. 扩展实验七 CPU 中断功能

- 修改设计数据通路和控制器
  - 兼容 Exp07 数据通路增加中断通路
  - 增加中断控制
    - 修改或替换 Exp07 的数据通路及控制器
- 扩展 CPU 中断功能
  - 非法指令中断；
  - 算术溢出中断；
  - 外部中断；
- 此实验在 Lab7 的基础上完成

2. 设计 CPU 中断测试方案

3. 设计 CPU 中断测试程序

2.2 实验原理

2.2.1 MIPS 中断结构

- 协处理器CP0
  - MIPS用来辅助的部件
    - 处理异常
    - 存储器管理
    - 系统配置
    - 其他片上功能
- 常用CP0寄存器

寄存器	编号	作用
BadVAddr	8	最近内存访问异常的地址
Count	9	高精度内部计时计数器
Compare	11	定时常数匹配比较寄存器
Status(SR)	12	状态寄存器、特权、中断屏蔽及使能等，可位控
Cause	13	中断异常类型及中断持起位
EPC	14	中断返回地址
Config	16	配置寄存器，依赖于具体系统

2.2.2 CPO 传输指令

- 控制寄存器访问指令
  - 读CP0指令mfco
    - mfc0 rt,rd： GPR[rt] <= CP0[rd]

Op=6bit	rs=00000	rt=5bit	Rd=5bit	=11个0
0x10	0	rt	rd	00000 000000

    - 写CP0mtco指令
      - mtc0 rd， rt： GPR[rd] <= CP0[rt]

Op=6bit	rs=00000	rt=5bit	Rd=5bit	=11位0
0x10	4	rt	rd	00000 000000

2.2.3 中断相关指令

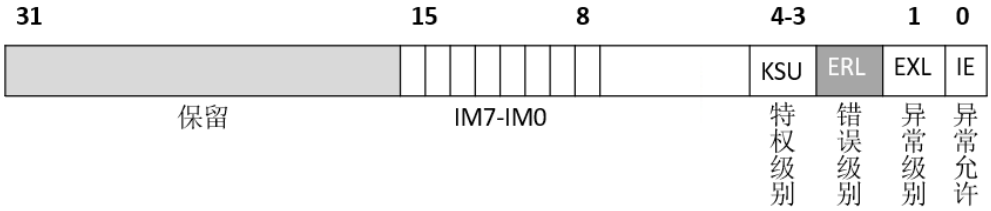
- 异常返回
  - eret
    - PC <= EPC；(CP0的Cause和Status寄存器有变化)

Op=6bit	1	19bit	FUN
0x0	1	000 0000 0000 0000 0000	011000

- 系统调用
  - syscall
    - EPC = PC+4; PC <= 异常处理地址; Cause和Status寄存器有变化
  - 参数：
    - $v0=系统调用号：Fig B-9-1
    - $a0~$a3、$f12，返回在$v0

Op=6bit	20bit	FUN
0x0	0000 0000 0000 0000 0000	0011000

2.2.4 状态寄存器：Status(SR)



- **IE**： IE=1全局中断使能
- **EXL**：异常设置，强制CPU进入内核模式并关闭中断，优先级高于IE
- **ERL**：数据错误设置，进入内核模式并关闭中断，优先级高于IE
- **IM7-0**：8个中断屏蔽位，6个外中断，2个软中断。没有优先级
- **KSU**：00=内核态、01=监管模式、10=用户态。优先级低于EXL和ERL

2.2.5 MIPS 中断响应

- 初始化
  - 设置 SR：关中断 IE=0， KSU ERL EXL=00 0 0
  - 系统初始化
  - 设置 SR：开中断 IE=1，设置 IM7-0
- 中断响应
  - 硬件保存断点：EPC←PC+4
  - 硬件修改 PC：PC←向量 、硬件关中断
  - 中断服务：保护寄存器、开中断*、服务、恢复寄存器
  - 开中断*
  - 返回：eret （硬件修改 Cause 和 Status 寄存器）

2.2.6 典型处理器中断结构

- **Intel x86中断结构**
  - 中断向量：000～3FF，占内存最底1KB空间
    - 每个向量由二个16位生成20位中断地址
    - 共256个中断向量，向量编号n=0～255
    - 分硬中断和软中断，响应过程类同，触发方式不同
    - 硬中断响应由控制芯片8259产生中断号n(接口原理课深入学习)
- **ARM中断结构**
  - 固定向量方式(嵌入式课程深入学习)

异常类型	偏移地址(低)	偏移地址(高)	
复 位	00000000	FFFF0000	
未定义指令	00000004	FFFF0004	
软中断	00000008	FFFF0008	
预取指令终	0000000C	FFFF000C	
数据终止	00000010	FFFF0010	
保留	00000014	FFFF0014	
中断请求(IRQ)	00000018	FFFF0018	
快速中断请求(FIQ)	0000001C	FFFF001C	

三、主要仪器设备

1. 计算机（Intel Core i5 以上，4GB 内存以上）系统　　1 套
2. 计算机软硬件课程贯通教学实验系统　　1 套
3. Xilinx ISE14.4 及以上开发工具　　1 套

四、操作方法与实验步骤

4.1 简化中断设计：ARM 模式

- **ARM中断向量表**

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	内核模式
0x0000004	未定义指令终止	未定义指令终止Und	RI内核模式
0x0000008	软中断（SWI）	超级用户Svc	Sys系统调用
0x000000c	Prefetch abort	指令预取终止Abt	Reserved自定义
0x0000010	Data abort	数据访问终止Abt	Ov
0x0000014	Reserved	Reserved	Reserved自定义
0x0000018	IRQ	外部中断模式IRQ	Int外中断（硬件）
0x000001C	FIQ	快速中断模式FIQ	Reserved自定义

4.2 设计方案参考：DataPath

◎ **DataPath 修改**

- CPU 复位时 IE=0，EPC=PC=ox00000000
  - IE=中断使能**(重要)**
- 修改 PC 模块增加(ARM 模式)
  - EPC 寄存器，INT 触发 PC 转向中断地址
    - 相当于硬件触发 Jal，用 eret 返回
  - 增加控制信号 INT、RFE/eret
    - INT 宽度根据扩展的外中断数量设定
- 修改 DataPath 增加(MIPS 模式)*
  - RI、Ov、EI
  - Cause 和 Status
  - 增加 Cause 和 Status 通道
  - 增加 Wt_Data 通道
  - 增加控制信号 CP0_Write

注意：INT 是电平信号，不要重复响应

4.3 设计控制器

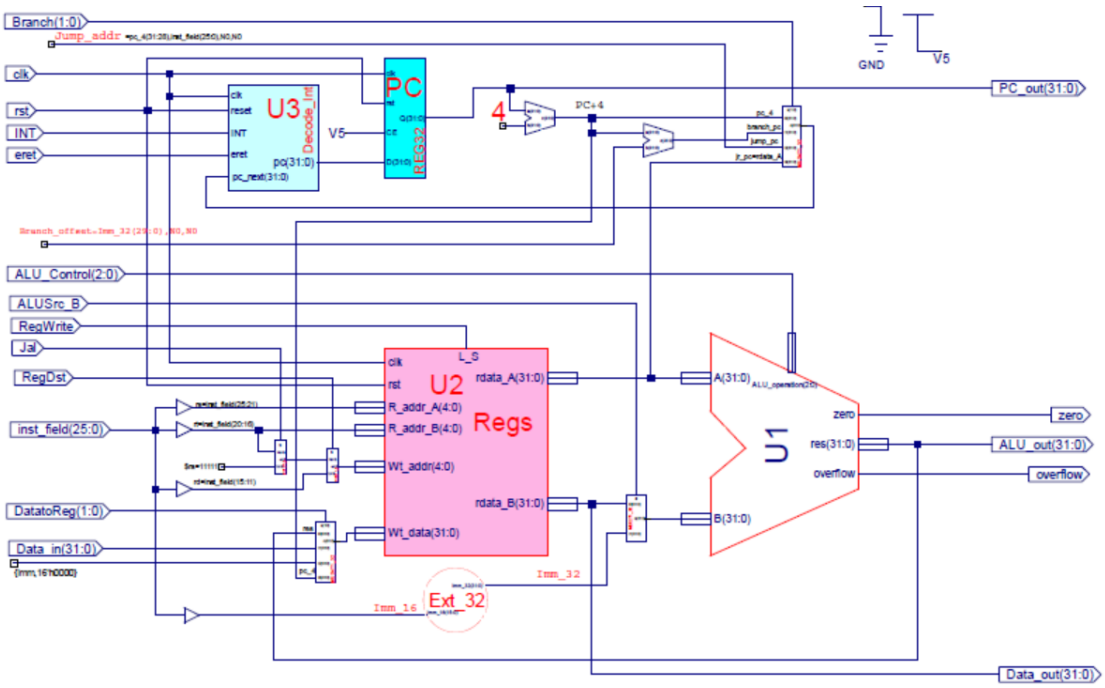
◎ **控制器修改**

- ARM 模式(简单)
  - 仅增加 eret 指令
  - 中断请求信号触发 PC 转向，在 Datapath 模块中修改
- MIPS 模式(可独立模块)*
  - 扩展 mfc0、mtc0 指令译码
  - 增加 Wt_Write 通道选择控制
  - 增加 CP0_Write
  - 增加控制信号 eret、RI、CP0_Write

◎ **中断调试**

- 首先时序仿真
- 物理验证
  - 用 BTN[0]触发调试：静态或低速
  - 用计数器 counter1_OUT 调试：动态或高速
    - 动态时注意死锁

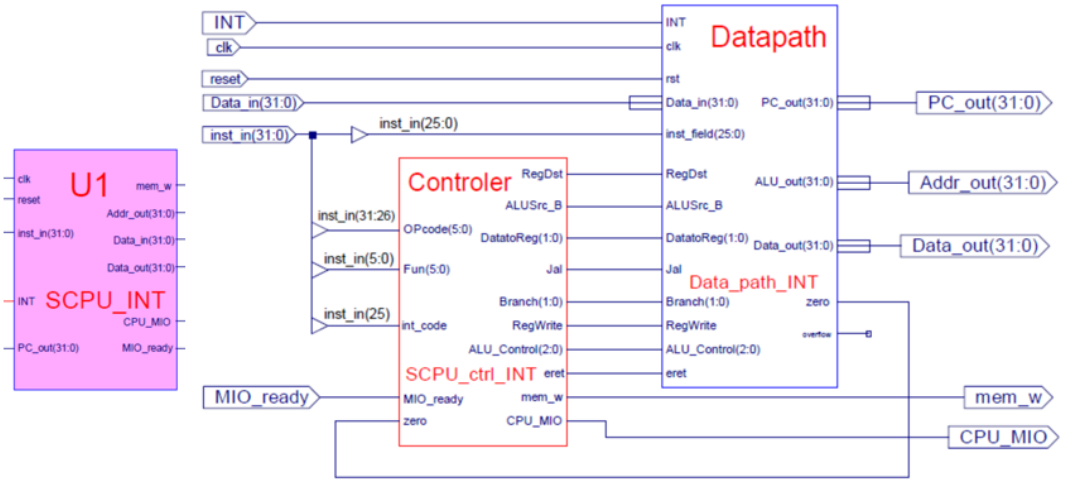
4.4 增加简单中断后的 DataPath



4.5 增加中断后的 CPU 模块

◎ 注意修改模块逻辑符号

℄ SCPU_INT.sym、SCPU_ctrl_INT.sym和Data_path_INT.sym



五、实验结果与分析

5.1 中断核心：Decoder_int

```
module Decoder_Int(
        input clk,
        input reset,
        input INT,
        input eret,
        input [31:0] pc_next,
        output reg [31:0] pc
);

reg doing = 0, cando = 1, newint = 0;
reg [31:0] EPC;

always @(posedge clk or posedge reset)
    begin
        if (reset)
            begin
                doing <= 0;
                cando <= 1;
                EPC <= 0;
            end
        else if (cando & newint)
            begin
                doing <= 1;
                cando <= 0;
                EPC <= pc_next;
            end
        else begin
            doing <= 0;
            if (eret) cando <= 1;
        end
    end

wire clr;
assign clr = reset | doing;
always @(posedge INT or posedge clr)
begin
    if (clr) newint <= 0; else newint <= 1;
end
```

```
always @*
    begin
        if (reset) pc <= 32'h00000000;
        else if (cando & newint) pc <= 32'h00000004;
        else if (eret) pc <= EPC;
        else pc <= pc_next;
    end
endmodule
```

我开了 doing，cando，newint 这些量。

Doing 的意义是，当我遇到中断并走进去处理时，遇到别的异常我也不响应。每当一个异常处理完，doing 就会置为 0，同时 cando 置为 1。Newint 相当于是对 int 进行了一个小处理。因为 int 是电平信号，可能会一直触发，所以只有当 int 处于上升沿的时候，才会触发 newint。

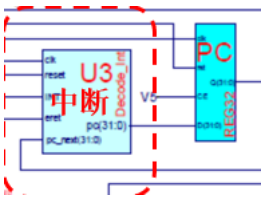
每次当进入一个异常的时候，就把新的 pc_next 赋值成 h’4，同时用 EPC 记录一下原来的 pc。处理完异常后，pc_next 重新指向原来的位置。

5.2 Data_path_INT.v

修改后的代码核心如下：

```
       assign PC_4 = PC_out + 32'b100;
        assign Jump_addr = {PC_4[31:28], inst_field[25:0], N0, N0};
        assign Branch_offset = {imm32[29:0], N0, N0};
        assign inD = Branch[1] ? (Branch[0] ? RdataA : Jump_addr) :
(Branch[0] ? PC_4 + Branch_offset : PC_4);

  //REG32 v_PC(.clk(clk), .rst(rst), .CE(V5), .D(inD), .Q(PC_out));
   Decoder_Int
t(.clk(clk),.reset(rst),.INT(INT),.eret(eret),.pc_next(inD),.pc(mid));
  REG32 v_PC(.clk(clk), .rst(rst), .CE(V5), .D(mid), .Q(PC_out));
```



其实就是根据这个图，把原来的 PC 拆成两部分。

5.3 SCPU_ctrl_int

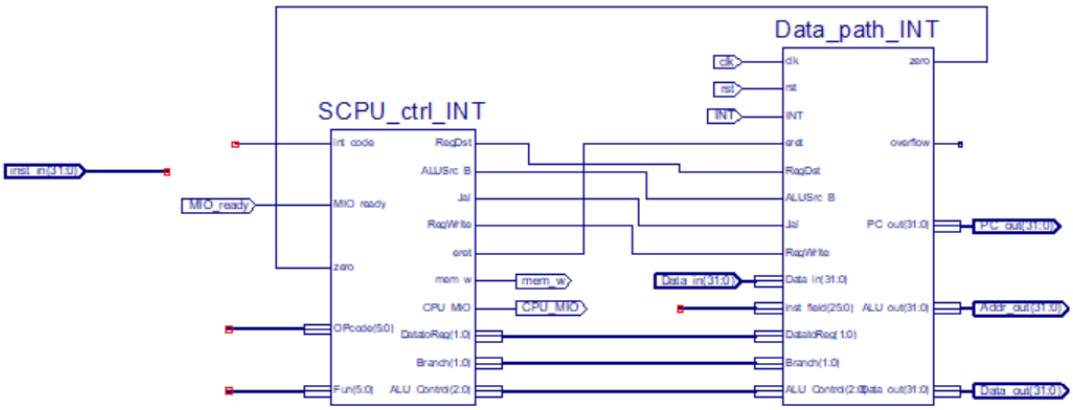
SCPU 里只要写上

```
    eret = int_code;
```

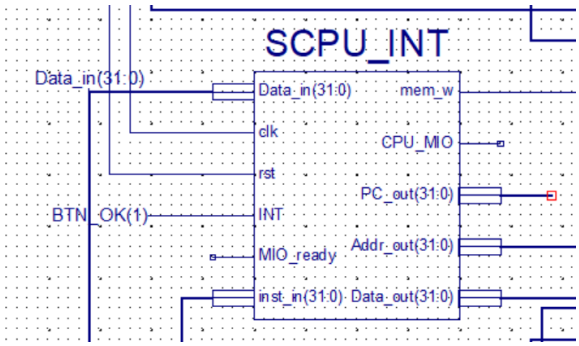
再增加一个 eret 指令即可（因为处理完异常后要用 eret 指回来）。

```
    6'b011000: `CPU_ctrl_signals = 12'b101000001110;  //eret
```

5.4 SCPU_int 顶层图



5.5 top 中的 SCPU_INT



六、讨论、心得

我们实现的异常和中断，其实只是比较简单的 arm 中的处理方式，而且只要求遇到中断指令后，立即跳到 h’4 的位置开始做，并在 eret 指令命令后回到原来的位置。

助教说，SCPU_INT 的 INT 要从 BTN_OK 中接受指令。最后的效果是，**如果强行按 BTN_OK(1)，会强制认为碰到了异常，并进入 h’4 地址。**我最后也做到了这一步，在 SW[7：5]=111 时，如果一按 BTN_ok(1)，PC_out 会立刻跳到 h’4。但是由于时间紧迫，也可能是给出的 int_coe 有点问题，我虽然 PC_out 能正确地跳会 h’4，但是显示出来的实验结果（跑马灯和矩形框）并不能做到和前几个实验一样。有点可惜>_<。