

# 重庆大学课程设计报告

课程设计题目： MIPS SOC 设计

学 院： 计算机学院

专业班级： 信息安全 1 班、2 班

年 级： 2021 级

姓 名： 罗哲、 刘倬宇、 庞博、 杨佳俊

学 号： 20215509、20215292、20215238、20215223

完成时间： 2024 年 1 月 12 日

成 绩：

指导教师： 吴长泽

重庆大学教务处制

综合设计指导教师评定成绩表

项目	分值	优秀 (100>x≥90)	良好 (90>x≥80)	中等 (80>x≥70)	及格 (70>x≥60)	不及格 (x<60)	评分
		参考标准	参考标准	参考标准	参考标准	参考标准	
学习态度	15	学习态度认真，科学作风严谨，严格保证设计时间并按任务书中规定的进度开展各项工作	学习态度比较认真，科学作风良好，能按期圆满完成任务书规定的任务	学习态度尚好，遵守组织纪律，基本保证设计时间，按期完成各项工作	学习态度尚可，能遵守组织纪律，能按期完成任务	学习马虎，纪律涣散，工作作风不严谨，不能保证设计时间和进度	
技术水平与实际能力	25	设计合理、理论分析与计算正确，实验数据准确，有很强的实际动手能力、经济分析能力和计算机应用能力，文献查阅能力强、引用合理、调查调研非常合理、可信	设计合理、理论分析与计算正确，实验数据比较准确，有较强的实际动手能力、经济分析能力和计算机应用能力，文献引用、调查调研比较合理、可信	设计合理，理论分析与计算基本正确，实验数据比较准确，有一定的实际动手能力，主要文献引用、调查调研比较可信	设计基本合理，理论分析与计算无大错，实验数据无大错	设计不合理，理论分析与计算有原则错误，实验数据不可靠，实际动手能力差，文献引用、调查调研有较大的问题	
创新	10	有重大改进或独特见解，有一定实用价值	有较大改进或新颖的见解，实用性尚可	有一定改进或新的见解	有一定见解	观念陈旧	
论文(计算书、图纸)撰写质量	50	结构严谨，逻辑性强，层次清晰，语言准确，文字流畅，完全符合规范化要求，书写工整或用计算机打印成文；图纸非常工整、清晰	结构合理，符合逻辑，文章层次分明，语言准确，文字流畅，符合规范化要求，书写工整或用计算机打印成文；图纸工整、清晰	结构合理，层次较为分明，文理通顺，基本达到规范化要求，书写比较工整；图纸比较工整、清晰	结构基本合理，逻辑基本清楚，文字尚通顺，勉强达到规范化要求；图纸比较工整	内容空泛，结构混乱，文字表达不清，错别字较多，达不到规范化要求；图纸不工整或不清晰	

指导教师评定成绩：

指导教师签名：

年 月 日

# MIPS SOC 设计报告

罗哲、刘倬宇、庞博、杨佳俊

## 一、设计简介

本次硬件综合实验课程，我们实现了具有 57 条指令、连接了 SRAM 接口、AXI 接口、实现了基础 cache 的五级流水线的 MIPS 处理器。此次的设计，我们成功通过了 89 个功能测试点、性能测试的 10 个程序的仿真以及一系列需要上板的操作，并最终成功得到了相应的性能分数。我们的设计以计算机组成原理的 lab4 实验的标准代码为基础，通过添加指令、模块和数据通路，成功完成了本次课程的相关要求。

### （一）小组分工说明

杨佳俊：添加逻辑运算指令、移位指令、性能测试与一系列的上板操作

庞 博：添加访存指令、数据移动指令、特权指令

刘倬宇：添加算术运算指令、添加异常处理模块与 cpu0

罗 哲：添加分支跳转指令、封装 SRAM 与 AXI 接口、添加 cache

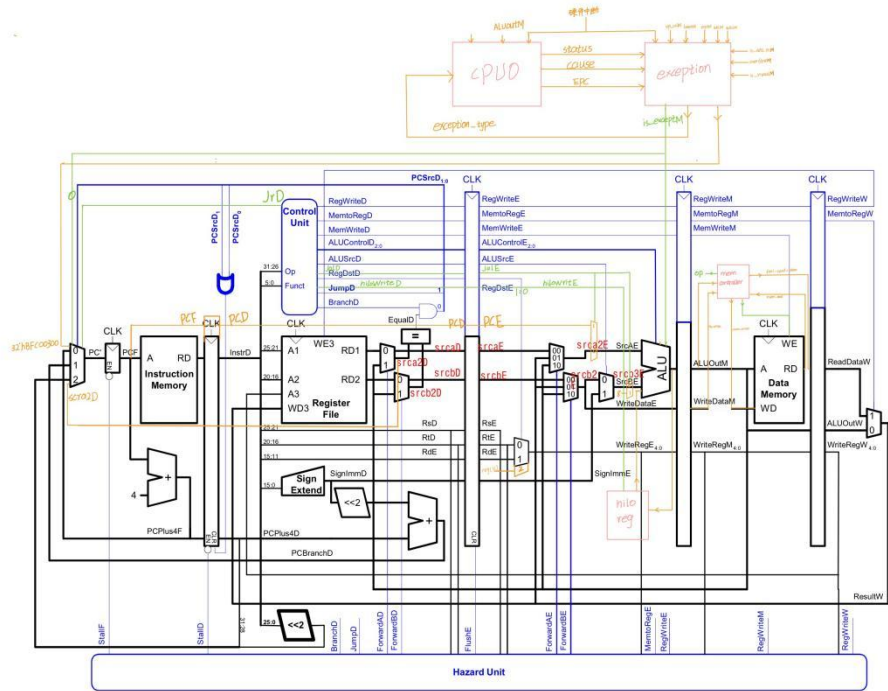
全体组员：对通路进行调试，对功能测试调试、跑通测试点

## 二、设计方案（30%）

### （一）总体设计思路

总体设计思路是先添加除特权指令的其他 52 条指令，然后封装 SRAM 接口跑通 89 个功能测试点，然后添加异常与特权指令，最后封装 AXI 接口与添加 cache。其中 5 级流水线 MIPS CPU 共有取指、译码、执行、访存、回写五个阶段，处理器的工作通过从指令存储器中取值，通过译码产生指令的控制信号，用于控制指令按规定的的数据通路执行，执行阶段进行运算、访存阶段访问数据存储器、回写阶段写入寄存器堆。

数据通路图：



为实现 57 条指令，需要完成的设计有控制信号的设计、数据通路的设计、处理数据冒险等。我们采用对指令分类，分模块逐步实现的方式，具体的设计详细描述如下：

## 1. 逻辑运算指令

### 1.1 指令机器码

	31:26	25:21	20:16	15:11	10:6	5:0	
逻辑运算指令	000000	rs	rt	rd	00000	100100	and rd,rs,rt
	000000	rs	rt	rd	00000	100101	or rd,rs,rt
	000000	rs	rt	rd	00000	100110	xor rd,rs,rt
	000000	rs	rt	rd	00000	100111	nor rd,rs,rt
	001100	rs	rt	immediate			andi rt,rs,immediate
	001110	rs	rt	immediate			xori rt,rs,immediate
	001111	00000	rt	immediate			lui rt,immediate
	001101	rs	rt	immediate			ori rs,rt,immediate

### 1.2 and, or, xor, nor 指令

#### 1.2.1 功能描述

寄存器 rs 中的值与寄存器 rt 中的值按位逻辑与，结果写入寄存器 rd 中。

#### 1.2.2 数据通路调整

数据通路与 lab4 相同，无需调整，只需在 alu 中增加对应运算即可。

1.2.3 控制信号调整

扩展 alucontrol 信号种类，识别在 alu 中进行何种运算。

1.3 andi, ori, xori, lui 指令

1.3.1 功能描述

ANDI ORI XORI：寄存器 rs 中的值与 0 扩展至 32 位的立即数 imm 按位逻辑运算，结果写入寄存器 rt 中。

LUI：将 16 位立即数 imm 写入寄存器 rt 的高 16 位，寄存器 rt 的低 16 位置 0。

1.3.2 数据通路调整

ANDI ORI XORI 三条指令是零扩展，其他需要扩展的指令都是有符号扩展，而 LUI 指令如何扩展都不影响结果，因为 LUI 指令只需要低十六位。只需根据这三条指令的 op 码中间两位判断是否零扩展即可。将 op 码的中间两位传入 signext 模块，并修改 signext 模块如下：

```
module signext(  
    input wire[15:0] a,  
    input wire[1:0] zero_s,  
    output wire[31:0] y  
);  
  
    assign y = (zero_s == 2'b11)? {{16{1'b0}}},a : {{16{a[15]}}},a;  
endmodule
```

1.3.3 控制信号调整

扩展 alucontrol 信号的种类，用于识别并控制这些指令在 ALU 中执行何种操作。

2. 移位运算指令

2.1 指令机器码

移位指令	000000	00000	rt	rd	sa	000000	sll rd,rt,sa
	000000	00000	rt	rd	sa	000010	srl rd,rt,sa
	000000	00000	rt	rd	sa	000011	sra rd,rt,sa
	000000	rs	rt	rd	00000	000100	sllv rd,rt,rs
	000000	rs	rt	rd	00000	000110	srlv rd,rt,rs
	000000	rs	rt	rd	00000	000111	srav rd,rt,rs

2.2 立即数移位运算 SLL SRL SRA

2.2.1 功能描述

SLL SRL：由立即数 sa 指定移位量，对寄存器 rt 的值进行逻辑移位，结果写入寄存器

rd 中。SRA: 由立即数 sa 指定移位量, 对寄存器 rt 的值进行算术右移(即左边补上符号位), 结果写入寄存器 rd 中。

### 2.2.2 数据通路调整

新增 sa 信号, 从指令的[10:6]位直接连入 ALU 中。需在 ALU 中对应处理运算。并在数据通路中将 sa 的值有 ID 阶段传递至 EXE 阶段:

```
assign saD = instrD[10:6];  
  
flopencrc #(5) r7E(clk,rst,~stallE,flushE,saD,saE);
```

### 2.2.3 控制信号调整

扩展 alucontrol 信号的种类, 用于识别并控制这些指令在 ALU 中执行何种操作。

## 2.3 变量移位运算 SLLV SRLV SRAV

### 2.3.1 功能描述

由寄存器 rs 中的值指定移位量, 对寄存器 rt 的值进行逻辑移位, 结果写入寄存器 rd 中。

### 2.3.2 数据通路调整

数据通路与计组 Lab4 相同, 无需调整。

### 2.3.3 控制信号调整

扩展 alucontrol 信号的种类, 用于识别并控制这些指令在 ALU 中执行何种操作。

移位指令在 alu 中的运算如下:

```
//移位指令  
'SLL_CONTROL : result = b << sa; //sll  
'SRL_CONTROL : result = b >> sa; //srl  
'SRA_CONTROL : result = $signed(b) >>> sa; //sra  
'SLLV_CONTROL : result = b << a[4:0]; //sllv  
'SRLV_CONTROL : result = b >> a[4:0]; //srlv  
'SRAV_CONTROL : result = $signed(b) >>> a[4:0]; //srav
```

3. 算术运算指令

3.1 机器码

算术运算指令	000000	rs	rt	rd	00000	100000	add rd,rs,rt
	000000	rs	rt	rd	00000	100001	addu rd,rs,rt
	000000	rs	rt	rd	00000	100010	sub rd,rs,rt
	000000	rs	rt	rd	00000	100011	subu rd,rs,rt
	000000	rs	rt	rd	00000	101010	slt rd,rs,rt
	000000	rs	rt	rd	00000	101011	sltu rd,rs,rt
	000000	rs	rt	00000	00000	011000	mult rs,rt
	000000	rs	rt	00000	00000	011001	multu rs,rt
	000000	rs	rt	00000	00000	011010	div rs,rt
	000000	rs	rt	00000	00000	011011	divu rs,rt
	001000	rs	rt	immediate			addi rt,rs,immediate
	001001	rs	rt	immediate			addiu rt,rs,immediate
	001010	rs	rt	immediate			slti rt,rs,immediate
	001011	rs	rt	immediate			sltiu rt,rs,immediate

3.2 指令实现过程

3.2.1 指令功能

ADD: 将寄存器 rs 的值与寄存器 rt 的值相加，结果写入寄存器 rd 中。如果产生溢出，则触发整型溢出例外；

ADDU: 将寄存器 rs 的值与寄存器 rt 的值相加，结果写入 rd 寄存器中；

SUB: 将寄存器 rs 的值与寄存器 rt 的值相减，结果写入 rd 寄存器中。如果产生溢出，则触发整型溢出例外；

SUBU: 将寄存器 rs 的值与寄存器 rt 的值相减，结果写入 rd 寄存器中；

SLT:将寄存器 rs 的值与寄存器 rt 中的值进行有符号数比较，如果寄存器 rs 中的值小，则寄存器 rd 置 1，否则寄存器 rd 置 0；

SLTU:将寄存器 rs 的值与寄存器 rt 中的值进行无符号数比较，如果寄存器 rs 中的值小，则寄存器 rd 置 1； 否则寄存器 rd 置 0；

MULT: 有符号乘法，寄存器 rs 的值乘以寄存器 rt 的值，乘积的低半部分和高半部分分别写入 HILO 寄存器的低 32 位和高 32 位；

MULTU:无符号乘法，寄存器 rs 的值乘以寄存器 rt 的值，乘积的低半部分和高半部分分别写入 HILO 寄存器的低 32 位和高 32 位；

DIV: 有符号除法，寄存器 rs 的值除以寄存器 rt 的值，商写入 HILO 寄存器的低 32 位中，余数写入 HILO 寄存器的高 32 位中；

DIVU: 无符号除法，寄存器 rs 的值除以寄存器 rt 的值，商写入 HILO 寄存器的低 32 位中，余数写入 HILO 寄存器的高 32 位中；

ADDI: 将寄存器 rs 的值与有符号扩展至 32 位的立即数 imm 相加, 结果写入 rt 寄存器中。  
如果产生溢出, 则触发整型溢出例外;

ADDIU: 将寄存器 rs 的值与有符号扩展至 32 位的立即数 imm 相加, 结果写入 rt 寄存器中;

SLTI: 将寄存器 rs 的值与有符号扩展至 32 位的立即数 imm 进行有符号数比较, 如果寄存器 rs 中的值小, 则寄存器 rt 置 1; 否则寄存器 rt 置 0;

SLTIU: 将寄存器 rs 的值与有符号扩展至 32 位的立即数 imm 进行无符号数比较, 如果寄存器 rs 中的值小, 则寄存器 rt 置 1; 否则寄存器 rt 置 0。

### 3.2.2 实现过程

可以将这 14 条指令两两分组, ADD 和 ADDU、ADDI 和 ADDIU、SUB 和 SUBU、SLT 和 SLTU、SLTI 和 SLTIU、DIV 和 DIVU、MULT 和 MULTU。

每一组中的两条指令的功能是完全一样的, 唯一的区别在于操作数是看成有符号还是无符号。这意味着在数据通路设计方面, 可以复用一些已有的通路。

如果将 ALU 看成一个黑盒, 那么不同组指令执行的操作是完全一样的, 都是从寄存器或立即数中取得源操作数, 送入 ALU 中进行运算, 然后将结果写回 rd 或 rt 寄存器中, 所以区别仅在于 ALU 内部执行的操作。

①ADD、ADDI 和 ADDU、ADDUI:

```
`ADD_CONTROL    : {double_sign,result} = {a[31],a} + {b[31],b}; //ADD、ADDI
`ADDU_CONTROL    : result = a + b;                               //ADDU、ADDIU
```

其中我们在对有符号的运算时使用了一个符号位 double\_sign 来作为最高位相加运算的结果, 当操作为有符号的加或减时对 double\_sign 的值与运算结果的最高位进行异或运算, 若结果为一, 则产生溢出; 若为零, 则不产生溢出。

```
assign overflow = (alucontrolE==`ADD_CONTROL || alucontrolE==`SUB_CONTROL)
& (double_sign ^ result[31]);
```

②SUB、SUBI 和 SUBU、SUBUI:

```
`SUB_CONTROL     : {double_sign,result} = {a[31],a} - {b[31],b}; //SUB
`SUBU_CONTROL     : result = a - b;                               //SUBU
```

其中 double\_sign 与加法的用法一样。

③SLT 和 SLTU:

```
`SLT_CONTROL: result = $signed(a) < $signed(b) ? 32'b1 : 32'b0; //SLT
`SLTU_CONTROL:result = a < b ? 32'b1 : 32'b0;                      //SLTU
```

有符号的小于则置位使用了 \$sign()。\$signed(c) 是一个 function, 将无符号数 c 转化为有符号数返回, 不改变 c 的类型和内容。

④MUL 和 MULTU:

```
`MULT_CONTROL: hilo_out = $signed(a) * $signed(b);                //MULT
`MULTU_CONTROL:hilo_out = {32'b0, a} * {32'b0, b};                //MULTU
```

同样使用了 \$sign() 函数来进行有符号乘法的运算。将运算的 64 位结果存入 hilo\_out 中, 将这个信号传入 hilo 寄存器。

⑤: DIV 和 DIVU:



我们在 ALU 中接入了 ref\_code 中给出的 div 模块，这个模块的输入输出信号如下：

```
input wire  clk, //时钟信号
input wire  rst, //复位信号
input wire  signed_div_i, //是否为有符号除法（1为div, 0为divu）
input wire[31:0] opdata1_i, //第一个操作数（被除数）
input wire[31:0] opdata2_i, //第二个操作数（除数）
input wire  start_i, //除法是否开始的信号
input wire  annul_i, //结束除法的信号
output reg[63:0] result_o, //除法的运算结果
output reg  ready_o //除法是否完成
```

通过在 ALU 中对 div 模块传入输入信号，来对除法进行运算，ALU 中相关信号的结果如下：

```
`DIV_CONTROL : begin //指令DIV, 除法器控制状态机逻辑
    if(~div_ready & ~div_start) begin //~div_start : 为了保证除法进行过程中,
        除法源操作数不因ALU输入改变而重新被赋值
        div_start <= 1'b1;
        div_signed <= 1'b1;
        div_stall <= 1'b1;
        a_save <= a; //除法时保存两个操作数
        b_save <= b;
    end
    else if(div_ready) begin
        div_start <= 1'b0;
        div_signed <= 1'b1;
        div_stall <= 1'b0;
        hilo_out <= div_result;
    end
end

`DIVU_CONTROL : begin //指令DIVU, 除法器控制状态机逻辑
    if(~div_ready & ~div_start) begin
        div_start <= 1'b1;
        div_signed <= 1'b0;
        div_stall <= 1'b1;
        a_save <= a; //除法时保存两个操作数
        b_save <= b;
    end
    else if(div_ready) begin
        div_start <= 1'b0;
        div_signed <= 1'b0;
        div_stall <= 1'b0;
        hilo_out <= div_result;
    end
end
```

当检测到是 DIV 或 DIVU 运算时，要用 a\_save 和 b\_save 来保存操作数，这是为了避免

alu 中操作数的改变对除法运算的结果造成影响。

```
wire annul; //终止除法信号
assign annul = ((alucontrolE == `DIV_CONTROL)|(alucontrolE == `DIVU_CONTROL)) & is_except;
```

终止除法信号 annul 取决于 is\_except 的值, 这个值是通过 exception 模块传入, 是触发异常, 会导致除法的刷新。

因为除法要 36 个时钟周期, 所以要进行流水线暂停。新增 stallE 控制信号, 控制译码 --> 执行阶段的流水线暂停, 在除法进行时, 拉高 stallE。

stallE 控制信号经由 hazard 模块生成, 要接入 ID-->EX 阶段的流水线寄存器中 (stallE 取反后接到使能端)。

```
//execute stage
flopenrc #(32) r1E(clk,rst,~stallE,flushE,srcaD,srcaE);
flopenrc #(32) r2E(clk,rst,~stallE,flushE,srcbD,srcbE);
flopenrc #(32) r3E(clk,rst,~stallE,flushE,signimmD,signimmE);
flopenrc #(5) r4E(clk,rst,~stallE,flushE,rsD,rsE);
flopenrc #(5) r5E(clk,rst,~stallE,flushE,rtD,rtE);
flopenrc #(5) r6E(clk,rst,~stallE,flushE,rdD,rdE);
flopenrc #(5) r7E(clk,rst,~stallE,flushE,saD,saE);
flopenrc #(6) r8E(clk,rst,~stallE,flushE,opD,opE);
flopenrc #(4) r9E(clk,rst,~stallE,flushE,
    {is_AdEL_pcD,is_syscallD,is_breakD,is_eretD},
    {is_AdEL_pcE,is_syscallE,is_breakE,is_eretE});
flopenrc #(1) r10E(clk,rst,~stallE,flushE,is_in_delayslotD,is_in_delayslotE);
flopenrc #(32) r11E(clk,rst,~stallE,flushE,pcD,pcE);
flopenrc #(5) r12E(clk,rst,~stallE,flushE,cp0_waddrD,cp0_waddrE);
flopenrc #(5) r13E(clk,rst,~stallE,flushE,cp0_raddrD,cp0_raddrE);
```

4. 数据移动指令

4.1 机器码

数据移动指令	000000	00000	00000	rd	00000	010000	mfhi rd
	000000	00000	00000	rd	00000	010010	mflo rd
	000000	rs	00000	00000	00000	010001	mthi rs
	000000	rs	00000	00000	00000	010011	mtlo rs

4.2 指令功能实现过程

4.2.1 指令功能

指令分别可以将 hilo 寄存器中的值移动到普通寄存器中, 或是将普通寄存器中的值移动到 hilo 寄存器中。

4.2.2 实现过程

为了能够让指令可以移动数据的位置, 首先需要实现一个 hilo 寄存器, 学校的实验代码中提供了 hilo 寄存器, 稍微修改之后使用;

```

module hilo_reg(
    input wire clk,rst,
    input wire we, //写使能
    input wire[63:0] hilo_in, //写入值
    output wire[63:0] hilo_out //读出值
);
    reg [63:0] hilo_reg; //HILO寄存器
    always @(negedge clk) begin
        if(rst)begin
            hilo_reg <= 0;
        end
        else if(we) begin
            hilo_reg <= hilo_in;
        end
    end
    assign hilo_out = hilo_reg;
endmodule

```

实现了 hilo 寄存器之后，在 maindec 中添加新的控制信号 hilo\_write，将其传递至执行阶段来控制 hilo 寄存器写入。并在 aludec 中添加新值控制 alu。

实现 MTHI、MTLO 时，执行阶段写 hilo 寄存器，复用 ALU，ALU 的输出新增 64 位的 hilo\_out 信号，接到 hilo 寄存器的输入端口，此时 hilo 寄存器的写控制信号是拉高的。在时钟下降沿，写入 hilo 寄存器。

实现 MFHI、MFLO 时，执行阶段读 hilo 寄存器，复用 ALU，ALU 新增 64 位输入值 hilo\_in 信号，运算后将其对应位置作为结果输出，传递到写回阶段后写入寄存器。

### 4.3 提示

由于 hilo 寄存器无论是读还是写都是在执行阶段进行的，因为多数情况下四条指令都是紧跟在乘除法后执行的，所以同一阶段写入读取不存在数据冒险。

## 5. 访存指令

### 5.1 指令机器码

访存指令	100000	base	rt	offset	lb rt,offset(base)
	100100	base	rt	offset	lbu rt,offset(base)
	100001	base	rt	offset	lh rt,offset(base)
	100101	base	rt	offset	lhu rt,offset(base)
	100011	base	rt	offset	lw rt,offset(base)
	101000	base	rt	offset	sb rt,offset(base)
	101001	base	rt	offset	sh rt,offset(base)
	101011	base	rt	offset	sw rt,offset(base)

## 5.2 指令功能实现过程

### 5.2.1 指令功能

LW: 根据地址，取一个字存入对应寄存器中；

LH: 根据地址，取一个半字有符号扩展后存入对应寄存器中；

LHU: 根据地址，取一个半字无符号扩展后存入对应寄存器中；

LB: 根据地址，取一个字节有符号扩展后存入对应寄存器中；

LBU: 根据地址，取一个字节无符号扩展后存入对应寄存器中；

SW: 根据地址，将对应寄存器中的值存入内存中；

SH: 根据地址，将对应寄存器中的低十六位值存入内存中；

SB: 根据地址，将对应寄存器中的低八位值存入内存中；

### 5.2.2 实现过程

为了实现对寄存器和数据存储器中的值进行处理，添加了新的模块 mem\_ctrl 来根据指令 op 以及地址对数据进行处理；

```
module mem_ctrl(  
    input wire[5:0] op_code,  
    input wire[31:0] addr,  
    //Load  
    input wire[31:0] mem_read_data,  
    output reg[31:0] final_read_data,  
    //Store  
    input wire[31:0] pre_write_data,  
    output reg[31:0] mem_write_data,  
    output reg [3:0] mem_wen ,  
    //except  
    output reg is_AdELM, //地址错例外（读数据或读取指令）  
    output reg is_AdESM //地址错例外（写数据）  
);
```

#### ①Load 指令：

load 指令可以复用实验四的 lw 指令的数据通路及控制信号，但需要对从数据存储器中取出的值进行进一步处理；

load 类指令每次从数据存储器取出一个字取出值发送到 mem\_ctrl 模块中，然后再根据指令类型处理出最后要回写的值：对于 LB、LBU 指令，需要根据偏移量选择 1 个字节；LH、LHU 指令需要选择低 2 字节或高 2 字节。

上面选择出来的内容可能是 8 位或 16 位，需要扩展到 32 位才能写回到寄存器中。LB、LH

进行有符号扩展，LBU、LHU 进行无符号扩展。

②Store 指令：

store 指令可以复用实验四的 SW 指令的数据通路，但需要对从寄存器中取出的值进行进一步处理；

store 类指令每次从寄存器中取出值发送到 mem\_ctrl 模块中，然后再根据指令类型处理出最后要写入内存的值：对于 SW 指令，直接将所有值写入； 对于 SH 指令，将低 2 字节扩展后写入；对于 SB 指令，将低 1 字节扩展后写入。

由于 store 指令写入数据存储器的值的长度是不同的，所以需要新增一个 4 位 mem\_wen 信号来控制数据存储器写入位置，当信号对应位为 1 时才会写入对应位置，为 0 时会丢弃，即 mem\_wen 信号为 0 处的数据值不重要。mem\_wen 的值由地址偏移决定。

比如 SH 指令地址计算后低两位为 00，则 mem\_wen 为 0011，写入数据存储器时只有数据低两字节会被真正写入。

6.3 提示

Load 指令是在写回阶段将数据写回的，若下一条指令需要使用写回寄存器的数据，则会产生数据冒险，我们的处理方法是暂停流水线，将数据写回后再继续执行。

6. 分支跳转指令

6.1 机器码

分支跳转指令	000000	rs	00000	00000	00000	001000	jr rs
	000000	rs	00000	rd	00000	001001	jalr rs/jalr rd,rs
	000010	instr_index					j target
	000011	instr_index					jal target
	000100	rs	rt	offset			beq rs,rt,offset
	000111	rs	00000	offset			bgtz rs,offset
	000110	rs	00000	offset			blez rs,offset
	000101	rs	rt	offset			bne rs,rt,offset
	000001	rs	00000	offset			bltz rs,offset
	000001	rs	10000	offset			bltzal rs,offset
	000001	rs	00001	offset			bgez rs,offset
	000001	rs	10001	offset			bgezal rs,offset

6.2 指令功能实现过程

6.2.1 指令功能

指令名称格式	指令功能简述
--------	--------



BEQ rs, rt, offset	相等转移
BNE rs, rt, offset	不等转移
BGEZ rs, offset	大于等于 0 转移
BGTZ rs, offset	大于 0 转移
BLEZ rs, offset	小于等于 0 转移
BLTZ rs, offset	小于 0 转移
BLTZAL rs, offset	小于 0 调用子程序并保存返回地址
BGEZAL rs, offset	大于等于 0 调用子程序并保存返回地址
J target	无条件直接跳转
JAL target	无条件直接跳转至子程序并保存返回地址
JR rs	无条件寄存器跳转
JALR rd, rs	无条件寄存器跳转至子程序并保存返回地址

## 6.2.2 实现过程

### ①BNE、BGEZ、BGTZ、BLEZ、BLTZ

这 5 条指令功能和 BEQ 类似，区别在于它们判断是否跳转的方式不一样。

数据通路方面，只需要对生成是否跳转信号的逻辑进行扩展即可，其余功能可直接复用 BEQ 的通路。

控制信号方面，要分别生成对应的控制信号，用于控制比较逻辑，可参照 BEQ 指令。

```
//分支跳转
`J:          controls <= 12'b0000_00_010000;
`JAL:        controls <= 12'b0000_10_110010;
`BEQ,`BNE,`BGTZ,`BLEZ: controls <= 12'b0010_00_000000;
`REGIMM_INST:
  case (rt)
    `BGEZ,`BLTZ:      controls <= 12'b0010_00_000000;
    `BGEZAL,`BLTZAL: controls <= 12'b0010_10_100010;
    default: begin
      controls <= 12'b0000_00_000000;
      is_invalid <= 1'b1;
    end
  endcase
```

### ②JR

JR 指令的功能与 J 指令完全相同，都是无条件跳转到目标地址，区别在于 JR 指令的跳转目标地址来自通用寄存器堆的第 rs 项。

数据通路只需新增寄存器堆的 rdata1 端口连接到生成下一个 PC 的多选器的输入。要添加新的控制信号，用于控制下一个 PC 的选择。

```
mux2 #(32) pcbrmux(pcplus4F,pcbranchD,pcsrcD,pcnexttbrFD);
mux2 #(32) pcmux(pcnexttbrFD,
  {pcplus4D[31:28],instrD[25:0],2'b00},
  jumpD,pcnextFD);
//jrD=1:地址为srca2D, 0:地址为pcnextFD
mux2 #(32) pc_jr_mux(pcnextFD,srca2D,jrD,pcnextjrD);
```

### ③JAL

根据指令规范文档可知，JAL 指令完成跳转目标地址的生成，并将 PC+8 写入到 31 号寄存器中。跳转这部分功能可以复用 J 指令的数据通路，而保存返回地址的操作在原有的基础

上无法完成，需要增加新的数据通路。

写寄存器涉及到写端口的地址和写数据，所以对于写地址，需要调整通用寄存器堆写端口的地址输入 waddr 的生成逻辑，增加一个固定数值 31 作为新的输入；而对于写数据，需要得到 PC+8，可以复用 ALU 的加法器，只不过第一个源操作数原来是仅来自寄存器堆的第一个输出端口，需要添加一个二选一部件，使其可以选择 PC 作为第一个源操作数，第二个需要将原有的二选一部件调整为三选一部件，8 作为第三个输入。

```
mux3 #(5) wrmux(rtE,rdE,5'b11111,regdstE,writeregE);
```

#### ④JALR

JALR 指令的功能类似于 JR 和 JAL 的结合。对于跳转操作，可以复用 JR 指令对应的数据通路。对于 Link 操作，其返回地址的计算还是 PC+8，这与 JAL 指令是一样的，不过计算的结果不再固定写到 31 号通用寄存器中，而是写入第 rd 项寄存器。因此 JALR 指令可以复用 JAL 指令计算 PC+8 并传递的数据通路，无需新增，只是写回级控制信号有所区别。

#### ⑤BLTZAL、BGEZAL

这两条指令可以看成是 BLTZ、BGEZ 指令与 JAL 指令的结合。对于跳转操作，下一个 PC 的生成逻辑等同于 BLTZ、BGEZ；对于 Link 操作，与 JAL 相同，将 PC+8 写入 31 号寄存器。因此，这两条指令不需要增加数据通路。

## 7. 特权指令

### 7.1 机器码

	31:26	25:21	20:16	15:11	10:3	2:0	
特权指令	010000	00100	rt	rd	00000000	sel	mtc0 rt,rd
	010000	00000	rt	rd	00000000	sel	mfc0 rt,rd
	31:26	25	24:6			5:0	
	010000	1	0000 0000 0000 0000 000			011000	eret

### 7.2 指令功能实现过程

#### 7.2.1 指令功能

特权指令可以在异常发生时对对应的异常进行处理：

BREAK：发生断点异常，无条件地将控制权转到异常处理程序；

SYSCALL：发生系统调用异常，无条件地将控制权转到异常处理程序；

ERET：在中断、异常或错误处理完成时返回中断指令。ERET 不执行下一条指令；

MTC0：将对应寄存器中的值存入 cp0 寄存器中；

MFC0：将 cp0 寄存器中的值存入对应寄存器中。

#### 7.2.2 实现过程

##### ①BREAK：

新增标记信号 `is_breakD`，并将此标记信号传递到访存阶段。在译码阶段通过判断指令的高 6 位以及最低 6 位来产生此标记信号。

#### ②SYSCALL:

新增标记信号 `is_syscallD`，并将此标记信号传递到访存阶段。在译码阶段通过判断指令的高 6 位以及最低 6 位来产生此标记信号。

#### ③ERET:

新增标记信号 `is_eretD`，并将此标记信号传递到访存阶段。这个指令取值是唯一的，直接用对应信号来判断。

#### ④MTCO:

新增写 CP0 寄存器的地址信号 `cp0_waddrD`，并将此地址信号传递到访存阶段。扩展 `alucontrol` 信号的种类，用于识别并控制指令在 ALU 中执行何种操作。新增 CP0 寄存器的写控制信号 `cp0_write`，并将此地址信号传递到访存阶段。在译码阶段通过判断指令的高 6 位以及最低 6 位来产生这些信号。

#### ⑤MFCO:

新增读 CP0 寄存器的地址信号 `cp0_raddrD`，并将此地址信号传递到访存阶段。扩展 `alucontrol` 信号的种类，用于识别并控制指令在 ALU 中执行何种操作。在译码阶段通过判断指令的高 6 位以及最低 6 位来产生这些信号。

### 7.3 提示

我们在执行阶段读取 `cp0`，在访存阶段写 `cp0`，所以可能产生数据冒险，解决方法是添加一个二选一多路选择器来进行数据前推。

## (二) Maindec 模块设计

### 1. 功能描述

该模块对指令进行译码，根据指令的 `op`、`funct`、`rs` 和 `rt` 识别所有指令，生成各个控制信号。



## 2. 接口定义

```
module maindec(  
    input wire[5:0] op,  
    input wire[5:0] funct,  
    input wire[4:0] rs,  
    input wire[4:0] rt,  
    output wire memtoreg,  
    output wire memwrite,  
    output wire branch,  
    output wire alusrc,  
    output wire[1:0] regdst,  
    output wire regwrite,  
    output wire jump,  
    output wire hilo_write,  
    output wire jr,  
    output wire jal,  
    output wire cp0_write,  
    output reg is_invalid  
);
```

信号名	位宽	输入/输出	功能
Op	6bit	input	指令的 opcode
funct	6bit	input	指令的 funct
rs	5bit	input	指令的 rs
rt	5bit	input	指令的 rt
memtoreg	1bit	output	回写的数据来自于 ALU 计算的结果还是 存储器读取的数据
memwrite	1bit	output	是否需要写数据存储器
branch	1bit	output	是否为 branch 指令
alusrc	1bit	output	送入 ALU B 端口的 值是立即数的 32 位 扩展还是寄存器堆读 取的值
regdst	2bit	output	写入寄存器堆的地址 是 rt 还是 rd 还是 31
regwrite	1bit	output	是否需要写寄存器堆
jump	1bit	output	是否为 jump 指令
hilo_write	1bit	output	是否需要写 HI、LO

			寄存器
jr	1bit	output	是否是 jr 指令
jal	1bit	output	是否是 Link 类型指令
cp0_write	1bit	output	是否需要写 CP0 寄存器
is_invalid	1bit	output	是否是无效指令

### 3. 逻辑控制

根据 op 区分指令是 R-Type 还是 I-Type 还是 J-Type 还是特权指令。R-Type 指令根据 funct 再作区分;I-Type 指令中的 REGIMM 指令根据 rt 再作区分;特权指令根据 rs 再作区分。然后再为不同指令生成各个控制信号。

```
assign {memtoreg,memwrite,branch,alusrc, regdst ,regwrite,jump,hilo_write,jr,jal,cp0_write} = controls;
always @(*) begin
    is_invalid <= 1'b0;
    case (op)
        `R_TYPE:
            case(funct)
                //逻辑指令
                `AND,`NOR,`OR,`XOR: controls<= 12'b0000_01_100000;
                //移位指令
                `SLLV,`SLL,`SRAV,`SRA,`SRLV,`SRL: controls<= 12'b0000_01_100000;
                //算术指令
                `ADD,`ADDU,`SUB,`SUBU,`SLT,`SLTU: controls<= 12'b0000_01_100000;
                `DIV,`DIVU,`MULT,`MULTU: controls<= 12'b0000_00_001000;
                //数据移动指令
                `MFHI,`MFLO: controls<= 12'b0000_01_100000;
                `MTHI,`MTLO: controls<= 12'b0000_00_001000;
                //跳转
                `JR: controls<= 12'b0000_00_000100;
                `JALR: controls<= 12'b0000_01_100110;
                //自陷指令
                `BREAK,`SYSCALL: controls<= 12'b0000_00_000000;
                default: begin
                    controls <= 12'b0000_00_000000;
                    is_invalid <= 1'b1;
                end
            endcase
        //I-type
        //逻辑指令
        `ANDI,`LUI,`ORI,`XORI: controls <= 12'b0001_00_100000;
        //算术指令
        `ADDI,`ADDIU,`SLTI,`SLTIU: controls <= 12'b0001_00_100000;
        //访存指令
        `LB,`LBU,`LH,`LHU,`LW: controls <= 12'b1001_00_100000;
        `SB,`SH,`SW: controls <= 12'b0101_00_000000;
        //分支跳转
        `J: controls <= 12'b0000_00_010000;
        `JAL: controls <= 12'b0000_10_110010;
        `BEQ,`BNE,`BGTZ,`BLEZ: controls <= 12'b0010_00_000000;
        `REGIMM_INST:
            case (rt)
                `BGEZ,`BLTZ: controls <= 12'b0010_00_000000;
                `BGEZAL,`BLTZAL: controls <= 12'b0010_10_100010;
                default: begin
                    controls <= 12'b0000_00_000000;
                    is_invalid <= 1'b1;
                end
            endcase
    endcase
end
```

```

//特权指令
`SPECIAL3_INST:
    case(rs)
        `MTC0: controls <= 12'b0000_00_000001;
        `MFC0: controls <= 12'b0000_00_100000;
        `ERET: controls <= 12'b0000_00_000000;
        default: begin
            controls <= 12'b0000_00_000000;
            is_invalid <= 1'b1;
        end
    endcase
default: begin
    controls <= 12'b0000_00_000000;
    is_invalid <= 1'b1;
end
endcase
end
endmodule

```

### (三) Aludec 模块设计

#### 1. 功能描述

该模块对指令进行译码，根据指令的 op、funct、rs 和 rt 识别所有指令，生成 ALU 的控制信号 alucontrol。

#### 2. 接口定义

```

module aludec(
    input wire[5:0] funct,
    input wire[5:0] op,
    input wire[4:0] rs,
    input wire[4:0] rt,
    output reg[4:0] alucontrol
);

```

信号名	位宽	输入/输出	功能
Op	6bit	input	指令的 opcode
funct	6bit	input	指令的 funct
rs	5bit	input	指令的 rs
rt	5bit	input	指令的 rt
alucontrol	5bit	output	ALU 控制信号，代表不同的运算类型

#### 3. 逻辑控制

根据 op 区分指令是 R-Type 还是 I-Type 还是 J-Type 还是特权指令。R-Type 指令根据 funct 再作区分；I-Type 指令中的 REGIMM 指令根据 rt 再作区分；特权指令根据 rs 再作区分。然后再为不同指令生成 ALU 控制信号 alucontrol。

```

always @(*) begin
    case (op)
        `R_TYPE:
            case (funct)
                //逻辑运算
                `AND:    alucontrol = `AND_CONTROL;
                `NOR:    alucontrol = `NOR_CONTROL;
                `OR:     alucontrol = `OR_CONTROL;
                `XOR:    alucontrol = `XOR_CONTROL;
                //移位
                `SLLV:   alucontrol = `SLLV_CONTROL;
                `SLL:    alucontrol = `SLL_CONTROL;
                `SRAV:   alucontrol = `SRAV_CONTROL;
                `SRA:    alucontrol = `SRA_CONTROL;
                `SRLV:   alucontrol = `SRLV_CONTROL;
                `SRL:    alucontrol = `SRL_CONTROL;
                //数据移动
                `MFHI:   alucontrol = `MFHI_CONTROL;
                `MTHI:   alucontrol = `MTHI_CONTROL;
                `MFLO:   alucontrol = `MFLO_CONTROL;
                `MTLO:   alucontrol = `MTLO_CONTROL;
                //算术
                `ADD:    alucontrol <= `ADD_CONTROL;
                `ADDU:   alucontrol <= `ADDU_CONTROL;
                `SUB:    alucontrol <= `SUB_CONTROL;
                `SUBU:   alucontrol <= `SUBU_CONTROL;
                `SLT:    alucontrol <= `SLT_CONTROL;
                `SLTU:   alucontrol <= `SLTU_CONTROL;
                `MULT:   alucontrol <= `MULT_CONTROL;
                `MULTU:  alucontrol <= `MULTU_CONTROL;
                `DIV:    alucontrol <= `DIV_CONTROL;
                `DIVU:   alucontrol <= `DIVU_CONTROL ;
                //JALR
                `JALR:   alucontrol = `ADDU_CONTROL; //做加法
            default:    alucontrol = 5'b00000;
            endcase
    endcase

```

```

//I-type
//逻辑运算
`ANDI:    alucontrol = `AND_CONTROL;
`LUI:     alucontrol = `LUI_CONTROL;
`ORI:     alucontrol = `OR_CONTROL;
`XORI:    alucontrol = `XOR_CONTROL;
//算术运算
`ADDI:    alucontrol <= `ADD_CONTROL;
`ADDIU:   alucontrol <= `ADDU_CONTROL;
`SLTI:    alucontrol <= `SLT_CONTROL;
`SLTIU:   alucontrol <= `SLTU_CONTROL;
`LB, `LBU, `LH, `LHU, `LW, `SB, `SH, `SW:    alucontrol = `ADDU_CONTROL;

6'b111111: alucontrol<=5'b10110;1
//分支与跳转
`REGIMM_INST:
    case(rt)
        `BGEZAL, `BLTZAL:    alucontrol = `ADDU_CONTROL; //做加法
        default:    alucontrol = 5'b00000;
    endcase
`JAL : alucontrol = `ADDU_CONTROL; //做加法pc+8
//例外处理
`SPECIAL3_INST:
    case(rs)
        `MTC0:    alucontrol = `MTC0_CONTROL;
        `MFC0:    alucontrol = `MFC0_CONTROL;
        default:    alucontrol = 5'b00000;
    endcase
default:    alucontrol = 5'b00000;
endcase
end
endmodule

```

### 三、实验过程（40%）

#### （一）设计工作日志

2023.12.26

全体组员配置实验需要的环境，了解任务与要求，观看视频。

2023.12.27——2023.12.28

回顾计算机组成原理实验四、阅读文档与资料。

2023.12.29

讨论并分配成员任务，确认总体设计通路

罗哲开始编写前 8 条分支跳转指令，刘倬宇开始编写算术运算指令，庞博开始编写数据转移指令，杨佳俊开始编写逻辑指令。

2023.12.30——2023.12.31

罗哲完成所有分支跳转指令，刘倬宇完成算术指令的编写，庞博完成数据转移与访存指令的编写，杨佳俊完成逻辑指令与移位指令的编写。

2024.1.1

对所有成员编写的指令进行整合，完善并统一数据通路与变量的设定，并成功调通逻辑与移位指令的仿真测试结果。

2024.1.2

对算术指令，数据移动指令，跳转指令和访存指令进行调试，成功得到正确的仿真结果。

2024.1.3

罗哲编写 SRAM 接口，进行功能测试，通过前面 8 个测试点。其余人进行调试。

2024.1.4

所有人继续调试功能测试，更改数据通路，最后成功跑通前面 64 个功能测试点。

2024.1.5

刘倬宇编写例外、异常处理模块，庞博编写 5 条特权指令，罗哲继续完善数据通路和补充数据通路图。杨佳俊辅助前面两人的编写并汇合到一起。

2024.1.6

所有人将全部的 57 条指令进行功能测试，进行调试与调整，最后成功通过 89 个测试点。

2024.1.7

进行上板测试，发现数码管显示 0x44，不是 89 个测试点都能通过上板测试。所有人继

续对通路进行调试与改进，最后上板数码管显示 0x59,成功通过上板测试。

2024.1.8

所有人查看文档与视频，罗哲编写 AXI 接口并完成，进行功能测试，无法通过 89 个测试点。所有人开始进行调试与 debug。

2024.1.9

罗哲查看文档与视频开始编写 cache，其余人继续进行调试并最终成功通过 axi 接口的 89 个功能测试点。

2024.1.10

罗哲完成对 cache 的添加，刘倬宇与庞博对整个完整的处理器设计进行调试发现没有问题后，由杨佳俊开始性能测试仿真。

2024.1.11

杨佳俊通过更改频率等调试，成功通过了 10 个性能测试仿真。最后所有人一起进行性能的上板测试，最终等到相应的性能分。

## （二）主要的错误记录

### 1、错误 1

#### （1）错误现象

在进行 inst\_ram 导入时，无法正确识别指令。

#### （2）分析定位过程

观察 pc 的值看出 pc 没有正确自增，所以指令存储器读取错误，通过观察 datapath 模块中的指令选择器，发现是由于 pcsrcD 的值不正确导致，pcsrcD 的值是由跳转指令部分决定的。

#### （3）错误原因

跳转指令还未编写完成，单独测试无法提供 pcsrcD 的值，导致无法正确读取指令。

#### （4）修正效果

将 pcsrcD 的值暂时先设置为 1，之后可正确获取指令。

### 2、错误 2

#### （1）错误现象

编写完分支跳转指令后出现无法仿真的错误

#### （2）分析定位过程

观察仿真图 and 对比检查代码，查看错误文件

#### （3）错误原因

发现在 eqcmp 模块中在非寄存器类型的变量 (wire) y 上进行了过程赋值

#### (4) 修正效果

将 y 的类型改为 reg. 成功仿真

### 3、错误 3

#### (1) 错误现象

测试算术指令的除法运算时，发现只有第一次除法运算的结果

#### (2) 分析定位过程

观察仿真图，通过观察不同周期，寄存器对中相应寄存器的值和相应模块的输出

#### (3) 错误原因

除法运行的周期不够，导致除法运算不能正常完成

#### (4) 修正效果

在控制模块中对执行阶段的流水线加入 stall 暂停，当进行除法运算时就暂停直至完成除法。

### 4、错误 4

#### (1) 错误现象

进行功能测试的时候，第 9 个测试点错误

#### (2) 分析定位过程

查看报错发现是 pc 值为 0xbfc0f918 的地址发生错误，查看功能测试文件中该地址执行的指令，为 jalr 型指令。再次更换指令测试的跳转指令的 test1 的 coe 文件，观察仿真图，发现指令执行一段时间后重复执行。

#### (3) 错误原因

jalr 指令没有正确保存 pc 的值，查看代码发现是 aludec 模块没有添加 jalr 指令应该执行的操作

#### (4) 修正效果

在 aludec 模块中加入执行 jalr 指令时，进行加法的操作。成功通过改测试点

### 5、错误 5

#### (1) 错误现象

测试算术指令时，在观察仿真图时发现符号减法和加法的结果有问题。

#### (2) 分析定位过程

通过对其他算数指令的运算的结果仿真图的观察，发现其余的算术指令运算的结果没有问题，所以我们考虑应该是运算指令有问题。

#### (3) 错误原因

在进行有符号加减法运算时，我们通过最高位来判断操作数是正数还是负数，如果是负数的话通过取反加一来作为新的操作数。后来我们改变了思路，考虑到有符号加减法和无符号加减法的区别在于是否有溢出，我们直接对操作数进行运算，但是增加了一个 double\_sign 位来和运算结果的最高位一起来判断是否有溢出。

原来的代码：



```

wire [31:0] nb, addresult,subresult;
assign nb = ~b;
assign addresult = a + b;
assign subresult = a + nb;
assign signed_a = (a[31]==1'b1)?(~a+1):a; //根据操作数的最高位来判断是正数还是负数
assign signed_b = (b[31]==1'b1)?(~b+1):b;

```

```

`ADD_CONTROL: result <= signed_a+signed_b;
`ADDU_CONTROL:result <= a+b;
`SUB_CONTROL: result <= signed_a+(~signed_b);
`SUBU_CONTROL:result <= a-b;

```

修改后的代码：

```

reg double_sign; //凑运算结果的双符号位，处理整型溢出
assign overflow = (alucontrolE==`ADD_CONTROL || alucontrolE==`SUB_CONTROL) & (double_sign ^ result[31]);
//溢出标志
`ADD_CONTROL : {double_sign,result} = {a[31],a} + {b[31],b}; //指令ADD、ADDI
`ADDU_CONTROL : result = a + b; //指令ADDU、ADDIU
`SUB_CONTROL : {double_sign,result} = {a[31],a} - {b[31],b}; //指令SUB
`SUBU_CONTROL : result = a - b; //指令SUBU

```

#### (4) 修正效果

在修改了代码之后，仿真图上的结果正确。

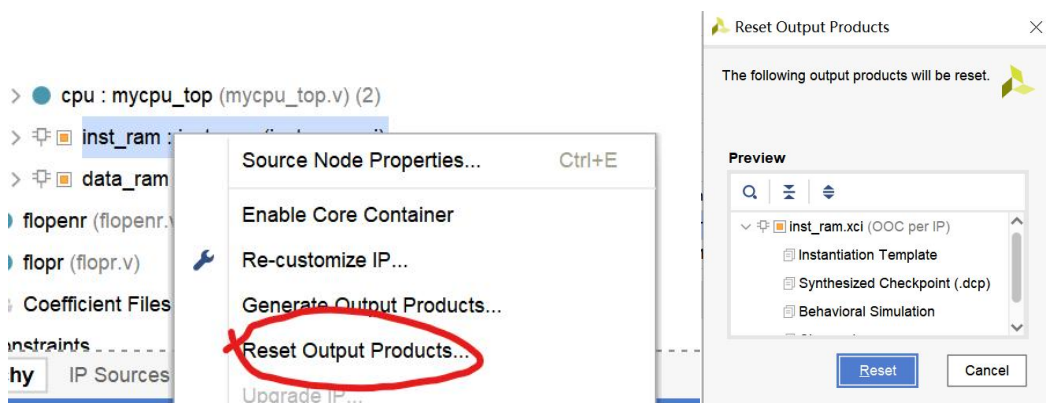
## 6、错误 6

### (1) 错误现象

在测试时无法根据新的 coe 文件测试，提示：Generate of output products did not run again as all output products were previously generated and up-to-date.

### (2) 解决办法

通过在网上查阅资料，我们找到了解决办法：右键单击列表中的 IP，点击 Reset Output Products. 在弹出的小窗中点击 Reset，即可重新修改并生成 IP。



## 7、错误 7

### (1) 错误现象

在测试跳转指令时，发现应存入\$31 的 PC+8 的值始终为 X。



## (2) 分析定位过程

经过仿真调试，我们加入了 `aluout` 的值以及一些信号观察，发现 `aluout` 的值其实是写入了 \$31 的，但只是当 `regdst` 为 11（即写入到 \$31）时的 `aluout` 始终为 XXXXX，所以才会出现这样的情况，而别的时候 `aluout` 的值是正确的。由此我们考虑到可能是有关 `pc` 的某个信号出现了问题。

## (3) 错误原因

我们发现是 `pcE` 未进行初始化，于是添加 `wire [31:0] pcE`，然后再进行仿真，发现 \$31 有正确的 `PC+8` 的值了。

## (4) 修正效果

通过仿真，应存入 \$31 的 `PC+8` 的值显示正确。

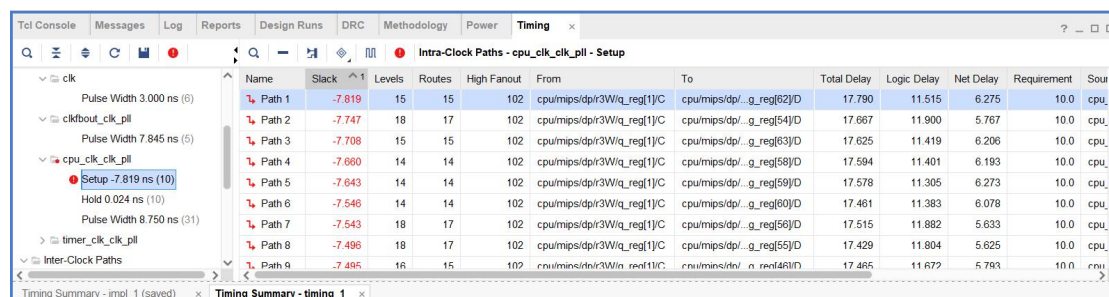
# 8、错误 8

## (1) 错误现象

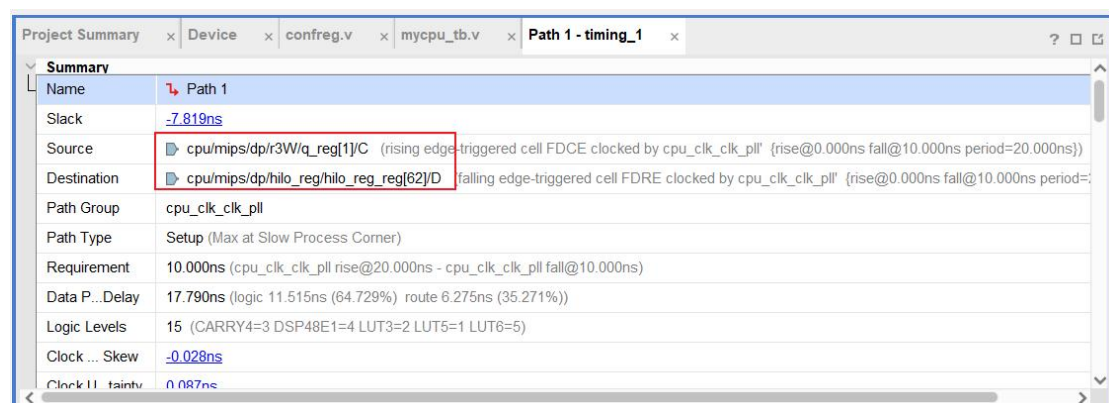
在 `sram` 上板测试时，板子上显示 2b 而不是 59（通过 89 条指令）

## (2) 分析定位过程

通过对文档《A10\_FPGA 在线调试说明\_v1.00》的阅读，我们对上板异常的原因进行了排查，在检查到时序报告时通过对 `intra clock path` 中出现报红的 `path` 进行查看，发现所有 10 个 `path` 的 `Destination` 都是 `hilo_reg` 寄存器的某位的值，于是我们猜想或许是 `alu` 运算时有关 `hilo_reg` 的赋值的时序出现问题。



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source
Path 1	-7.819	15	15	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[62]D	17.790	11.515	6.275	10.0	cpu
Path 2	-7.747	18	17	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[54]D	17.667	11.900	5.767	10.0	cpu
Path 3	-7.708	15	15	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[63]D	17.625	11.419	6.206	10.0	cpu
Path 4	-7.680	14	14	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[58]D	17.594	11.401	6.193	10.0	cpu
Path 5	-7.643	14	14	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[59]D	17.578	11.305	6.273	10.0	cpu
Path 6	-7.546	14	14	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[60]D	17.461	11.383	6.078	10.0	cpu
Path 7	-7.543	18	17	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[56]D	17.515	11.882	5.633	10.0	cpu
Path 8	-7.496	18	17	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[55]D	17.429	11.804	5.625	10.0	cpu
Path 9	-7.485	16	15	102	cpu/mips/dpr3W/q_reg[1]C	cpu/mips/dpr3W/q_reg[61]D	17.465	11.672	5.793	10.0	cpu



Name	Path 1
Slack	-7.819ns
Source	cpu/mips/dpr3W/q_reg[1]C (rising edge-triggered cell FDCE clocked by cpu_clk_clk_pll {rise@0.000ns fall@10.000ns period=20.000ns})
Destination	cpu/mips/dpr3W/hilo_reg/hilo_reg[62]D (falling edge-triggered cell FDRE clocked by cpu_clk_clk_pll {rise@0.000ns fall@10.000ns period=20.000ns})
Path Group	cpu_clk_clk_pll
Path Type	Setup (Max at Slow Process Corner)
Requirement	10.000ns (cpu_clk_clk_pll rise@20.000ns - cpu_clk_clk_pll fall@10.000ns)
Data Path Delay	17.790ns (logic 11.515ns (64.729%) route 6.275ns (35.271%))
Logic Levels	15 (CARRY4=3 DSP48E1=4 LUT3=2 LUT5=1 LUT6=5)
Clock Skew	-0.028ns
Clock Hysteresis	0.087ns

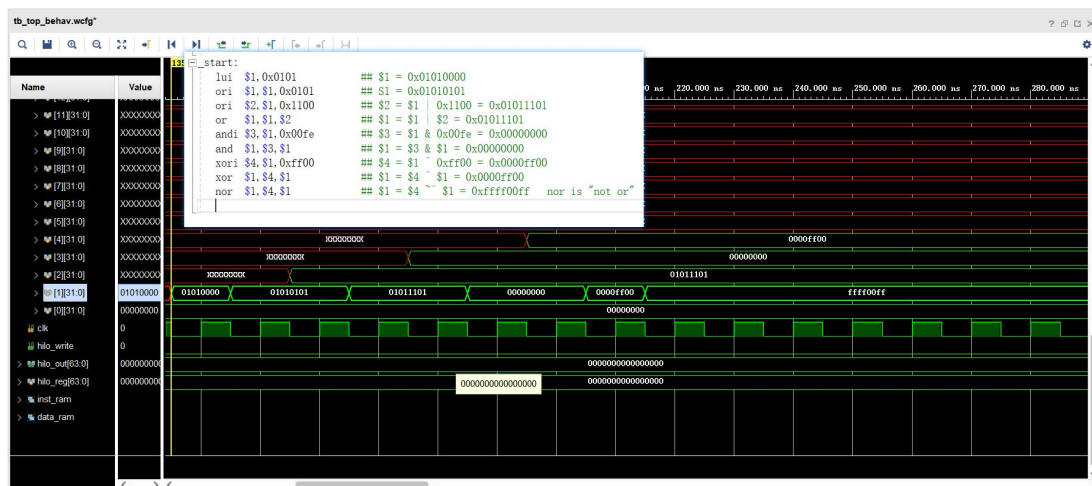
## (3) 错误原因

查看 `alu` 文件后发现乘除法运算时使用的是非阻塞赋值，于是我们将其改为阻塞赋值。

## (4) 修正效果

修改之后再上板测试，七段数码管成功显示为 59（即 89 个测试全部通过）。

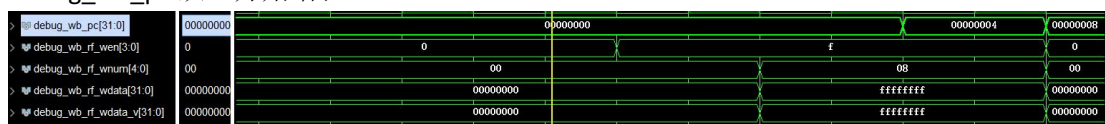
运算的正确结果都写入对应的寄存器当中。



## 10. 错误 10

### (1) 错误现象

Debug\_wb\_pc 从 0 开始增长



### (2) 分析定位过程

```
flop rwc #(32) rwcW(clk, rst, flushW, pcM, pcW);
```

```
assign debug_wb_pc = pcW;
```

```
assign debug_wb_rf_wen = {4{regwriteW}};
```

```
assign debug_wb_rf_wnum = writeregW;
```

```
assign debug_wb_rf_wdata = resultW;
```

根据流水线传递值的过程，Debug\_wb\_pc 的值受原 pc 的影响，从 0 开始应该是因为 PC 复位地址是 0。

### (3) 错误原因

PC 复位地址是 0。

### (4) 修正结果

修改 pc 模块中 pc 复位值，并添加 mmu 地址转换模块后，Debug\_wb\_pc 值从 bfc00000 开始增长。

## 四、设计结果

### （一）设计交付物说明

/submit/mycpu\_sram 文件夹下，是我们实现的 sram 接口的 MIPS 处理器。

/submit/mycpu\_axi 文件夹下，是我们实现的 axi 接口和添加了 cache 的 MIPS 处理器。

若要进行仿真，需要把整个 mycpu\_axi 或 mycpu\_sram 文件夹导入实验资料包相应的工程中。即可仿真、综合、上板。

### （二）设计演示结果

#### 1. sram 接口的功能测试

```
----[1335535 ns] Number 8'd80 Functional Test Point PASS!!!
      [1342000 ns] Test is running, debug_wb_pc = 0xbfc0069c
----[1350985 ns] Number 8'd81 Functional Test Point PASS!!!
      [1352000 ns] Test is running, debug_wb_pc = 0x00000000
      [1362000 ns] Test is running, debug_wb_pc = 0x00000000
----[1366435 ns] Number 8'd82 Functional Test Point PASS!!!
      [1372000 ns] Test is running, debug_wb_pc = 0xbfc0047c
----[1381885 ns] Number 8'd83 Functional Test Point PASS!!!
      [1382000 ns] Test is running, debug_wb_pc = 0xbfc00d60
      [1392000 ns] Test is running, debug_wb_pc = 0xbfc00680
----[1397345 ns] Number 8'd84 Functional Test Point PASS!!!
      [1402000 ns] Test is running, debug_wb_pc = 0x00000000
      [1412000 ns] Test is running, debug_wb_pc = 0xbfc5560c
----[1412825 ns] Number 8'd85 Functional Test Point PASS!!!
      [1422000 ns] Test is running, debug_wb_pc = 0x00000000
----[1428295 ns] Number 8'd86 Functional Test Point PASS!!!
      [1432000 ns] Test is running, debug_wb_pc = 0xbfc00490
      [1442000 ns] Test is running, debug_wb_pc = 0xbfc00694
----[1443755 ns] Number 8'd87 Functional Test Point PASS!!!
      [1452000 ns] Test is running, debug_wb_pc = 0xbfc004c0
----[1459225 ns] Number 8'd88 Functional Test Point PASS!!!
      [1462000 ns] Test is running, debug_wb_pc = 0xbfc15368
      [1472000 ns] Test is running, debug_wb_pc = 0xbfc00510
----[1474685 ns] Number 8'd89 Functional Test Point PASS!!!
=====
Test end!
----PASS!!!
```



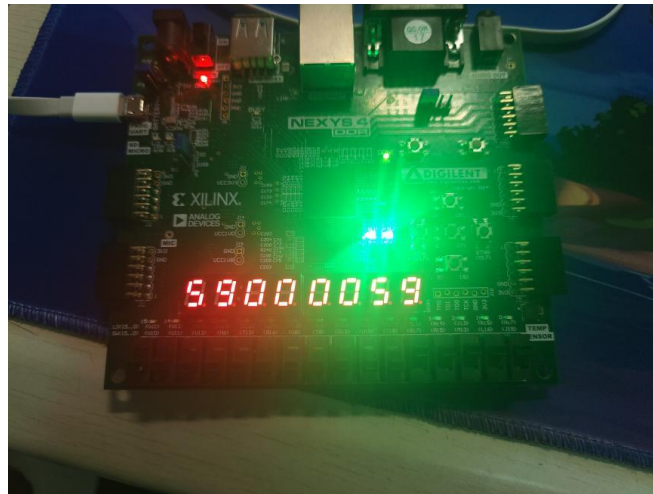
## 2. axi 接口的功能测试

```
[56702000 ns] Test is running, debug_wb_pc = 0x00000000
[56712000 ns] Test is running, debug_wb_pc = 0xbfc004d0
[56722000 ns] Test is running, debug_wb_pc = 0xbfc15490
[56732000 ns] Test is running, debug_wb_pc = 0x00000000
[56742000 ns] Test is running, debug_wb_pc = 0xbfc00390
[56752000 ns] Test is running, debug_wb_pc = 0x00000000
[56762000 ns] Test is running, debug_wb_pc = 0x00000000
[56772000 ns] Test is running, debug_wb_pc = 0xbfc00690
[56782000 ns] Test is running, debug_wb_pc = 0xbfc003a8
[56792000 ns] Test is running, debug_wb_pc = 0xbfc0067c
[56802000 ns] Test is running, debug_wb_pc = 0x00000000
[56812000 ns] Test is running, debug_wb_pc = 0xbfc0067c
[56822000 ns] Test is running, debug_wb_pc = 0xbfc003a0
[56832000 ns] Test is running, debug_wb_pc = 0xbfc00678
[56842000 ns] Test is running, debug_wb_pc = 0x00000000
[56852000 ns] Test is running, debug_wb_pc = 0xbfc00680
[56862000 ns] Test is running, debug_wb_pc = 0xbfc00398
[56872000 ns] Test is running, debug_wb_pc = 0xbfc005a4
[56882000 ns] Test is running, debug_wb_pc = 0xbfc15588
[56892000 ns] Test is running, debug_wb_pc = 0xbfc00588
[56902000 ns] Test is running, debug_wb_pc = 0xbfc15598
----[56903535 ns] Number 8'd89 Functional Test Point PASS!!!
[56912000 ns] Test is running, debug_wb_pc = 0xbfc00d90
[56922000 ns] Test is running, debug_wb_pc = 0x00000000
=====
Test end!
----PASS!!!
```

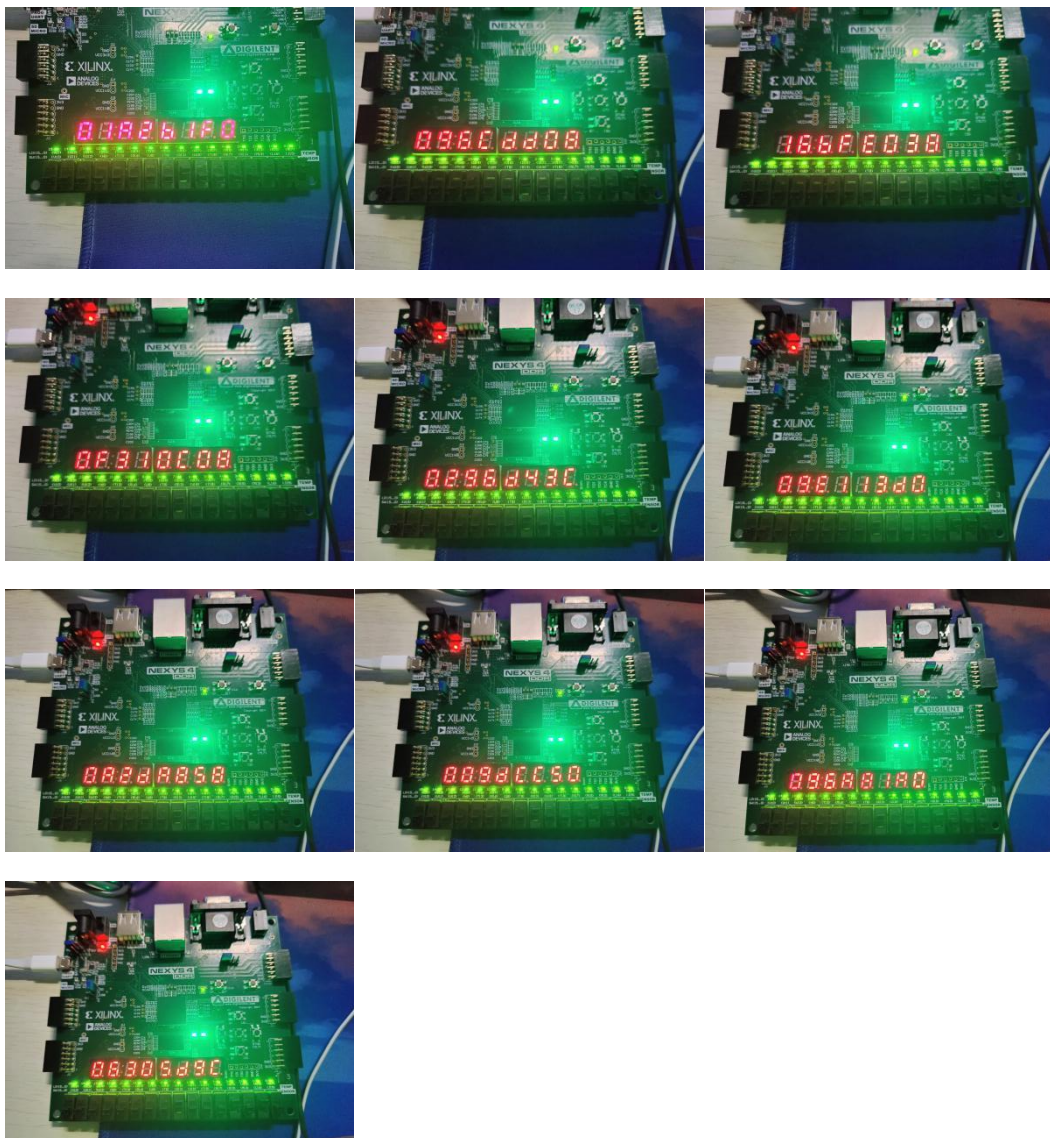
## 3. 性能测试仿真

```
dhrystone PASS!
string search PASS!
bitcount PASS!Bits: 811
crc32 PASS!
select sort PASS!
bubble sort PASS!
quick sort PASS!
coremark PASS!
```

#### 4. 功能测试上板



#### 5. 性能测试上板



## 6. 性能分

### 二、性能测试分数计算

序号	测试程序	myCPU	gs132	$T_{gs132}/T_{mycpu}$
		上板计时(16进制)	上板(16进制)	
		数码管显示	数码管显示	
cpu_clk: sys_clk		30MHz: 100MHz	50MHz: 100MHz	-
1	bitcount	1A2B1F0	13CF7FA	0.757039388
2	bubble_sort	96CDD08	7BDD47E	0.821360233
3	coremark	16BFE038	10CE6772	0.738752712
4	crc32	F310C08	AA1AA5C	0.699828382
5	dhystone	296D43C	1FC00D8	0.766416775
6	quick_sort	9E113D0	719615A	0.718594174
7	select_sort	A2DA858	6E0009A	0.675454194
8	sha	96A81A0	74B8B20	0.77475152
9	stream_copy	9DCC50	853B00	0.844309737
10	stringsearch	6305D9C	50A1BCC	0.814274516

性能分 0.759

## 五、参考设计说明

本次硬件综合模块引用如下：

- 1, 计算计组成原理 lab4 标准代码：实验资料包提供。
- 2, defines2.vh 宏定义：实验资料包提供。
- 3, div.v 除法模块：实验资料包提供。
- 4, cp0\_reg.v：实验资料包提供。
- 5, 基础 cache:实验资料包提供。
- 6, d\_sram\_to\_sram\_like.v、i\_sram\_to\_sram\_like.v 类 SRAM 转换：引用自重庆大学硬件综合设计实验文档中 2020 视频中。
- 7, 转接桥 bridge\_1x2.v, bridge\_2x1.v：引用自计算机系统结构实验二资料包。
- 8, mmu.v：实验资料包提供。
- 9, cpu\_axi\_interface.v：实验资料包提供。

## 六、总结

### （一）组员：罗皙

对我而言，这次硬件综合设计实验整体来说还是比较困难的，不过好在经过三周的努力，终于成功完成了它。通过这次项目，我们积累了硬件综合设计的经验，也拥有了一些团队协作能力和问题解决能力。过程虽然痛苦，但结果是好的。

### （二）组员：刘倬宇

在实验过程中我们遇到了很多错误,通过在仿真图中对信号的加入和观察和小组成员们的互帮互助,我们逐一解决了这些问题。这次硬件综合设计课程锻炼了我们的团队协作能力和沟通能力,小组成员们都积极讨论和热心互相帮助,解决问题。感谢每个小组成员的辛勤付出!

### **(三) 组员: 杨佳俊**

此次硬件综合设计难度较大,但通过与组员共同坚持,最终还算是圆满完成。非常感谢队友们的热心帮助,特别是刚开始做的时候不知道如何下手,是队友的耐心指导才让我有做下去的动力!最后,祝老师新年快乐,祝罗哲、刘倬宇和庞博新年快乐!

### **(四) 组员: 庞博**

经过这次硬件综合设计,使我更加了解了一个 cpu 运行的过程,也让我明白了要完成一个 cpu 的设计有多么困难,这让我更加敬佩那些设计 cpu 的工作人员们,希望未来我国的科研工作者能够在这方面取得更大的进步!

## **七、参考文献**

- [1]《计算机组成原理实验指导书》,重庆大学计算机学院编.
- [2]《A03\_“系统能力培养大赛”MIPS 指令系统规范\_v1.01》.
- [3]《功能测试说明》.
- [4]《性能测试说明》.
- [5]《自己动手写 CPU》,雷思磊著,电子工业出版社.
- [6]重庆大学硬件综合设计实验文档[EB/OL].[2023-01-12].<https://co.ccslab.cn/>
- [7]《指令及对应机器码\_2018》