《计算机组成原理实验》

实验报告

学 院 名 称 ：	计算机学院
专业（班级）：	19 级计算机科学与技术（人工智能与大数据方向）
学 生 姓 名 ：	陈恩婷
学　　号 ：	19335015
时　　间 ：	2020　年　12　月　11　日

成 绩 ：

实 验 ： 单周期CPU设计与实现

一. 实验目的

1. 理解MIPS常用的指令系统并掌握单周期CPU的工作原理与逻辑功能实现。
2. 通过对单周期CPU的运行状况进行观察和分析，进一步加深理解。

二. 实验内容

1. 利用 HDL 语言，基于 Xilinx FPGA basys3 实验平台，用 Verilog HDL 语言或 VHDL 语言来编写,实现单周期CPU 的设计,这个单周期 CPU 能够完成 16 条MIPS 指令，至少包含以下指令:

   支持基本的内存操作如lw，sw指令

   支持基本的算术逻辑运算如add，sub，and，ori，slt，addi指令

   支持基本的程序控制如beq，j指令
2. 掌握各个指令的相关功能并输出仿真结果进行验证，并最后在 FGPA 上实现，将其中的 alu 运算结果在开发板数码管上显示出来。
3. 可拓展添加其他指令。

三. 实验原理

1. 单时钟周期 CPU

单周期 CPU 的特点是每条指令的执行只需要一个时钟周期，一条指令执行完再执行下一条指令。再这一个周期中，完成更新地址，取指，解码，执行，内存操作以及寄存器操作。由于每个时钟上升沿时更新地址，因此要在上升沿到来之前完成所有运算，而这所有的运算除可以利用一个下降沿外，只能通过组合逻辑解决。这给寄存器和存储器 RAM 的制作带来了些许难度。且因为每个时钟周期的时间长短必须统一，因此在确定时钟周期的时间长度时，要依照最长延迟的指令时间来定，这也限制了它的执行效率。

单周期 CPU 在每个 CLK 上升沿时更新 PC，并读取新的指令。此指令无论执行时间长短，都必须在下一个上升沿到来之前完成。其时序示意如图Ⅰ。

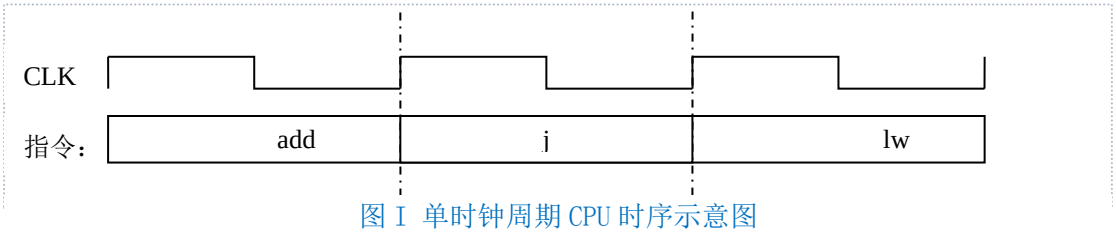


图 I 单时钟周期 CPU 时序示意图

下图是一个单周期 CPU 的顶层结构实现。主要器件有程序计数器 PC、程序存储器、寄存器堆、ALU、数据存储器和控制部件等。所有的控制信号简单地说明如下：

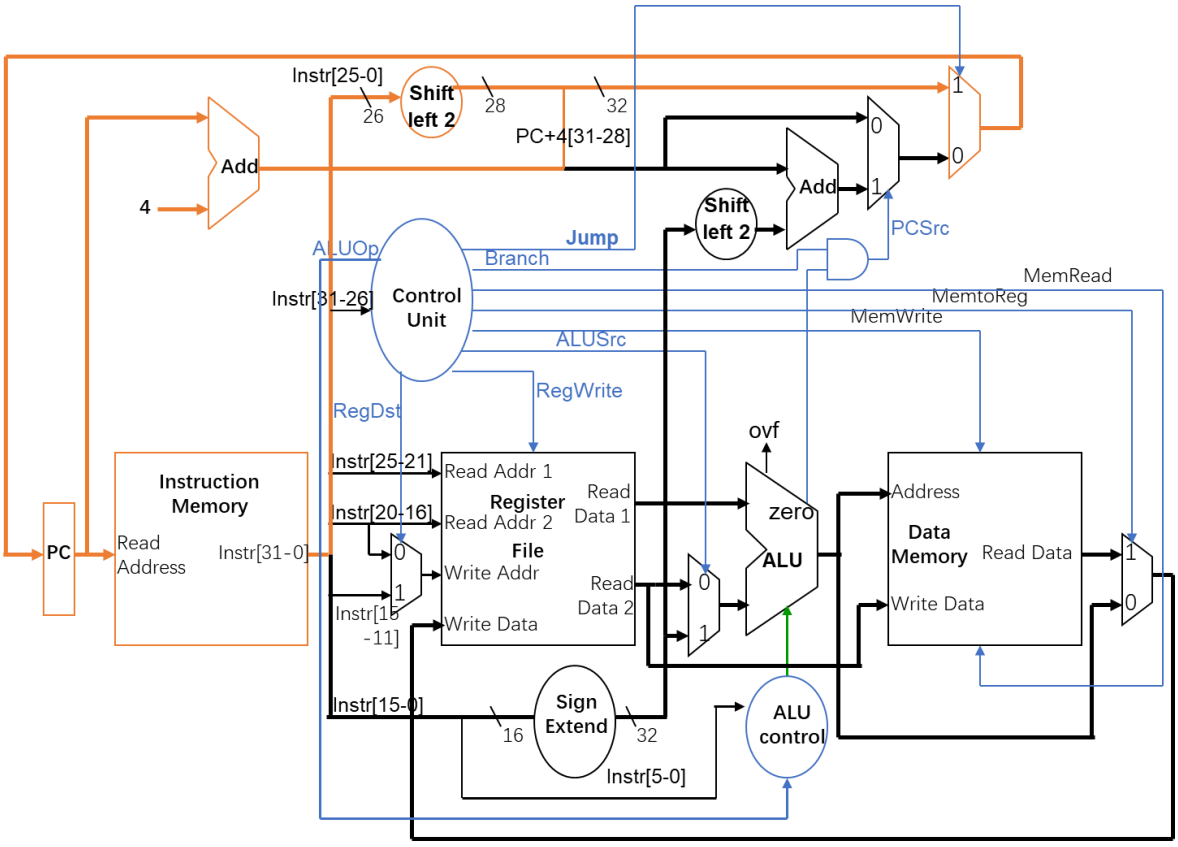


图 II 单时钟周期 CPU 详细逻辑设计图参考

其中，控制单元(Ctrl Unit)定义如下：

（1）JUMP：为 1 时，选择跳转目标地址；为 0 时，选择由 Branch 选出的地址；

（2）MemToReg：为 1 时，选择存储器数据；为 0 时，选择 ALU 输出的数据；

（3）Branch：为 1 时，选择转移目标地址；为 0 时，选择 PC +4（图中的 NextPC）；

（4）MemWrite：为 1 时写入存储器。存储器地址由 ALU 的输出决定，写入数据为寄存器 rt 的内容；

（5）ALUOP：ALU 控制码；

（6）ALUSrc：ALU 操作数 B 的选择，为 1 时，选择扩展的立即数；为 0 时，选择寄存器数据；

（7）RegWrite：为 1 时写入寄存器堆，目的寄存器号是由 RegDst 选出的 rt 或 rd，写入数据是由 MemToReg 选出的存储器数据或 ALU 的输出结果；

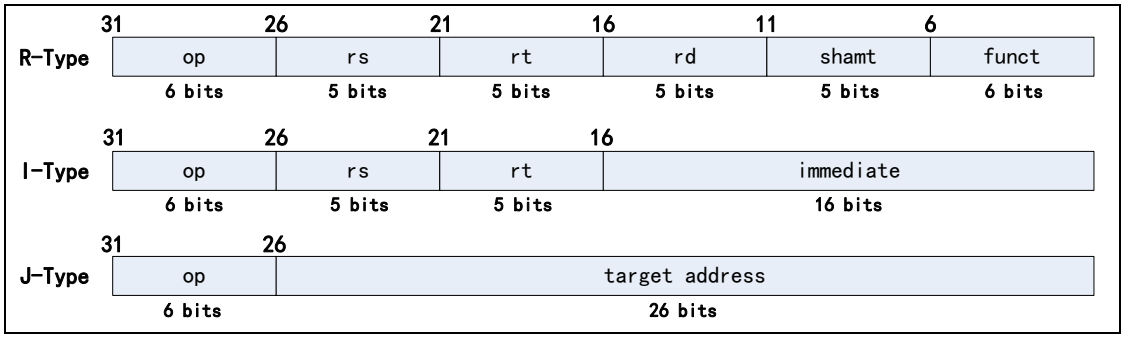
（8）ExtOp：符号扩展。为 1 时，符号扩展；为 0 时，0 扩展；

（9）RegDst：目的地址，为 1 时，选择 rd；为 0 时，选择 rt。

2. MIPS 指令集

本次实验共涉及三种类型的 MIPS 指令，分别为 R 型、I 型和 J 型，三种类型的 MIPS 指令格式定义如下：

- R（register）类型的指令从寄存器堆中读取两个源操作数，计算结果写回寄存器堆；
- I（immediate）类型的指令使用一个 16 位的立即数作为一个源操作数；
- J（jump）类型的指令使用一个 26 位立即数作为跳转的目标地址（target address）；



图Ⅲ MIPS 指令集

一条指令的执行过程一般有下面的五个阶段，指令的执行过程就是这五个状态的重复过程：

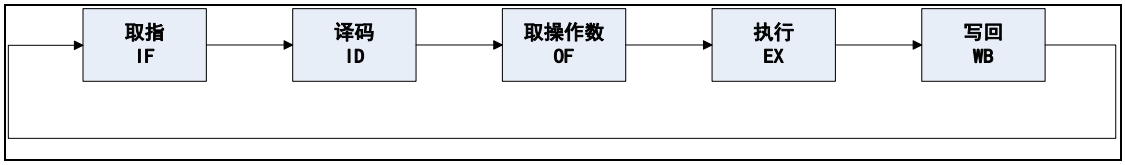


图 IV MIPS 指令集

在本次实验中，至少要完成 16 条指令的功能。

MIPS 的 31 种指令

助记符	指令格式						示例	示例含义	操作及解释
BIT #	31..26	25..21	20..16	15..11	10..6	5..0			
R-类型	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1,无符

									号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2&S3	(rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2|S3	(rd)←(rs) | (rt); rs=$2,rt=$3,rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2^S3	(rd)←(rs)^(rt); rs=$2,rt=$3,rd=$1
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1= ~($2 | S3)	(rd)←~((rs) | (rt)); rs=$2,rt=$3,rd=$1
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1，无符号数
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	(rd)←(rt)<<shamt,rt=$2,rd=$1,shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (逻辑右移)
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (算术右移，注意符号位保留)
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	(rd)←(rt)<<(rs), rs=$3,rt=$2,rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (逻辑右移)
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (算术右移，注意符号位保留)
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	(PC)←(rs)
I-类型	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2&10	(rt)←(rs)&(zero-extend)immediate,rt=$1,rs=$2
ori	001101	rs	rt	immediate			ori $1,$2,10	$1=$2|10	(rt)←(rs)|(zero-extend)immediate,rt=$1,rs=$2
xori	001110	rs	rt	immediate			xori $1,$2,10	$1=$2^10	(rt)←(rs)^(zero-extend)immediate,rt=$1,rs=$2
lui	001111	00000	rt	immediate			lui $1,10	$1=10*65536	(rt)←immediate<<16 & 0FFFF0000H，将 16 位立即数放到目的寄存器高 16 位，目的寄存器的低 16 位填 0
lw	100011	rs	rt	offset			lw $1,10($2)	$1=Memory[$2+10]	(rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2
sw	101011	rs	rt	offset			sw $1,10($2)	Memory[$2+10] =$1	Memory[(rs)+(sign_extend)offset]←(rt), rt=$1,rs=$2
beq	000100	rs	rt	offset			beq $1,$2,40	if($1=$2) goto PC+4+40	if ((rt)=(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2), rs=$1, rt=$2
bne	000101	rs	rt	offset			bne $1,$2,40	if($1≠$2) goto PC+4+40	if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2) , rs=$1, rt=$2

slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
J-类型	op	address					
j	000010	address			j 10000	goto 10000	(PC)←((Zero-Extend) address<<2), address=10000/4
jal	000011	address			jal 10000	$31=PC+4 goto 10000	($31)←(PC)+4; (PC)←((Zero-Extend) address<<2), address=10000/4

图 V 指令详细结构

以上即是 MIPS 指令说明。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3 板一块。

五. 实验过程与结果

1. CPU 设计的思想、方法:

该 CPU 主要由 9 个模块组成，各个子模块分别设计其特定的功能，最终利用一个总的模块进行子模块间连接，使得整个 CPU 能连贯执行指令，在仿真结果中观察设计结果，最终进行硬件下载，验证设计。其中各个模块简单功能如下:

（1）指令存储器模块：具备基本的读写功能，用于存放指令。

（2）寄存器堆模块：由 32 个 32 位的寄存器组成，提供较大的存储空间，用于存放暂存数据和指令。

（3）算术逻辑运算器模块：执行加减法等算术运算，与非或等逻辑运算，以及比较移位传送等操作的功能部件，是该 CPU 的设计核心部分，存在不同的运算处理功能，是体现实验设计结果正确性的模块。

（4）立即数扩展模块：执行 I 型指令时需要立即数扩展，该模块用于 MIPS 符号扩展，将 16 位数据扩展为 32 位数据。

（5）主控制模块：用于控制各个模块之间的分工运行，产生不同数据通路的控制信号，保证指令顺序执行不发生紊乱。

（6）ALU 控制模块：用于生成 ALU 执行各种功能的控制信号，使 ALU 内部运行不发生紊乱。

（7）数据存储器模块，用于 LW/SW 指令数据存取。

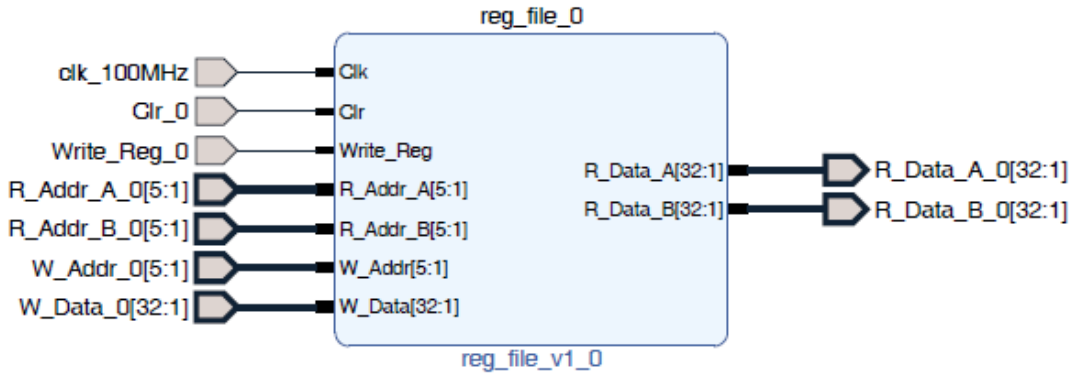
（8）取指模块：进行指令的取出及译码，同时包括程序计数器 PC 运行设计。

（9）显示模块 DISPLAY 数码管显示，显示 16 位运算结果。

各模块设计

1 寄存器模块：

寄存器组是指令操作的主要对象，MIPS 处理器里一共有 32 个 32 位的寄存器，故可以声明一个包含 32 个 32 位的寄存器数组。读寄存器时需要 Rs，Rd 的地址，得到其数据。写寄存器 Rd 时需要所写地址，所写数据，同时需要写使能。以上所有操作需要在时钟和复位信号控制下操作。故寄存器组设计如下：

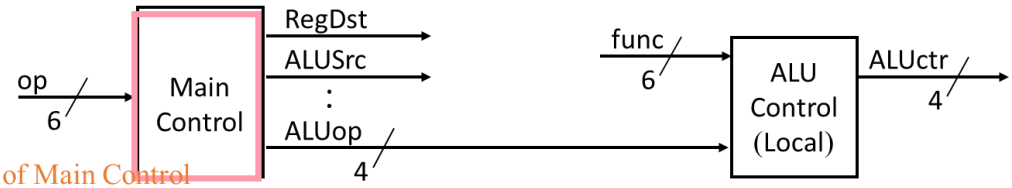



```
`timescale 1ns / 1ps//寄存器堆模块
module RegFile(Clk,
               Clr,
               Write_Reg,
               R_Addr_A,
               R_Addr_B,
               W_Addr,
               W_Data,
               R_Data_A,
               R_Data_B);

    parameter ADDR = 5;//寄存器编码地址位宽
    parameter NUMB = 1<<ADDR;//寄存器个数
    parameter SIZE = 32;//寄存器数据位数

    input Clk;//写入时钟信号
    input Clr;//清零信号
    input Write_Reg;//写控制信号
    input [ADDR:1]R_Addr_A;//A 端口读寄存器地址
    input [ADDR:1]R_Addr_B;//B 端口读寄存器地址
    input [ADDR:1]W_Addr;//写寄存器地址
    input [SIZE:1]W_Data;//写入数据

    output [SIZE:1]R_Data_A;//A 端口读出数据
    output [SIZE:1]R_Data_B;//B 端口读出数据
    reg [SIZE:1]REG_Files[0:NUMB-1];//寄存器堆本体
```

```
    integer i;//用于遍历 NUMB 个寄存器
    initial//初始化 NUMB 个寄存器，全为 0
        for(i = 0;i<NUMB;i = i+1)
            REG_Files[i] <= 0;
        always@(negedge Clk or posedge Clr)//时钟信号或清零信号上升沿
        begin
            if (Clr)//清零
                for(i = 0;i<NUMB;i = i+1)
                    REG_Files[i] <= 0;
            else//不清零,测写控制，高电平则写入寄存器
            if (Write_Reg)
                REG_Files[W_Addr] <= W_Data;
        end         //读操作没有使能或时钟信号控制，使用数据流(组合逻辑电路,要
时钟控制)
    assign R_Data_A = REG_Files[R_Addr_A];
    assign R_Data_B = REG_Files[R_Addr_B];
endmodule
```

2　控制器模块:

根据指令中的指令码（op）和功能码（funct）的不同组合输出相应的控制信号。



Output of Main Control

Input of main control

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtr	xxx
ALUop	0000	0010	0011	0011	0101	x

```
`timescale 1ns / 1ps
module ctr(input [5:0] opCode,
           output reg regDst,
           output reg aluSrc,
           output reg memToReg,
           output reg regWrite,
           output reg memRead,
```

```
            output reg memWrite,
            output reg branch,
            output reg ExtOp, //符号扩展方式 1：sign-extend 0: zero-extend
            output reg[3:0] aluop, // 经过 ALU 控制译码决定 ALU 功能
            output reg jmp);

    always@(opCode) begin
    // 操作码改变时改变控制信号
    case(opCode) 6'b000010: begin
    regDst   = 0;  aluSrc   = 0; memToReg   = 0;
    regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0111;
jmp = 1; ExtOp = 1;
    end // J 型指令操作码: 000010，无 ALU

    6'b000000: begin
    regDst   = 1;  aluSrc   = 0; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0000;
jmp = 0; ExtOp = 1;
    end // R 型指令操作码: 000000

    //'I'型指令操作码
    6'b000100: begin
    regDst   = 0;  aluSrc   = 0; memToReg   = 0;
    regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0101;
jmp = 0; ExtOp = 1;
    end // 'beq' 指令操作码: 000100

    6'b000101: begin
    regDst   = 0;  aluSrc   = 0; memToReg   = 0;
    regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b0110;
jmp = 0; ExtOp = 1;
    end // 'bne' 指令操作码: 000101

    6'b000001: begin
    regDst = 0; aluSrc = 0; memToReg = 0;
    regWrite = 0; memRead = 0; memWrite = 0; branch = 1; aluop = 4'b1010;
jmp = 0; ExtOp = 1;
    end // 'bltz' 指令操作码：000001

    6'b001000: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0011;
jmp = 0; ExtOp = 1;
    end // 'addi' 指令操作码: 001000

    6'b001001: begin
    regDst   = 0; aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1001;
```

```
jmp = 0; ExtOp = 1;
    end // 'addiu' 指令操作码：001001

    6'b001100: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0001;
jmp = 0; ExtOp = 0;
    end // 'andi' 指令操作码: 001100

    6'b001101: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0010;
jmp = 0; ExtOp = 0;
    end // 'ori' 指令操作码: 001101

    6'b001110: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1000;
jmp = 0; ExtOp = 0;
    end // 'xori' 指令操作码: 001110

    6'b001010: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0100;
jmp = 0; ExtOp = 1;
    end // 'slti' 指令操作码: 001010

    6'b001011: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b1011;
jmp = 0; ExtOp = 1;
    end // 'sltiu' 指令操作码：001011

    6'b001111: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 1; memRead = 0; memWrite = 0; branch = 0; aluop = 4'b0111;
jmp = 0; ExtOp = 0;
    end // 'lui' 指令操作码: 001111

    6'b100011: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 1;
    regWrite = 1; memRead = 1; memWrite = 0; branch = 0; aluop = 4'b0011;
jmp = 0; ExtOp = 1;
    end // 'lw' 指令操作码: 100011

    6'b101011: begin
    regDst   = 0;  aluSrc   = 1; memToReg   = 0;
    regWrite = 0; memRead = 0; memWrite = 1; branch = 0; aluop = 4'b0011;
```

```
jmp = 0; ExtOp = 1;
    end // 'sw' 指令操作码: 101011

    default: begin
    regDst   = 0;  aluSrc   = 0; memToReg   = 0;
    regWrite = 0; memRead = 0; memWrite = 0; branch = 0; aluop = 3'b0xxx;
jmp = 0; ExtOp = 0;
    end // 默认设置
    endcase end

endmodule
```

3 ALU 控制译码模块

ALU 主要执行5 种操作：与，或，加，减，小于设置。这五种操作可以使用四位的编码表示：0000，0001，0010，0110，0111。指令不同，则对应的ALU 运算不同，所以该模块需要根据指令来控制 ALU 进行正确的运算。

lw，sw，addi 指令均要求 ALU 执行加操作，则可分为一类，aluop编码 0011； beq 指令要求ALU 执行减操作，则分为一类，编码0101；

最后一类是 R 型指令，可以编码为 0000；但不同的R 型指令对应不同的 ALU 运算，故需要再通过指令的功能码进一步确定 ALU 的运算。

最终该模块即实现4 位操作码以及6 位功能码输出4 位ALU 控制信号码。

译码器的真值表：

ALU 功能真值表

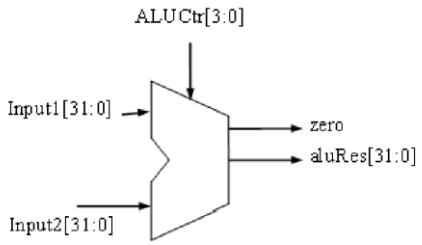
指令	ALUop	func	ALU_Operation	功能描述
R	0000	100000	0010	Add（加）
R	0000	100010	0110	Sub（减）
R	0000	100100	0000	And（与）
R	0000	100101	0001	Or（或）
R	0000	100111	0101	Nor（或非）
R	0000	101010	0111	Slt（小于设置）
lui	0111	op=001111	1101	Lui（设置高位）
R	0000	000000	1000	Sll（左移）
R	0000	000010	1001	Srl（右移）
R	0000	000011	1010	Sra（算术右移）
beq	0101	x	0110	Rs==rt,zero=1
bne	0110	x	0011	Rs==rt,zero=0

ori	0010	x	0001	
xori	1000	x	0100	异或
slti	0100	x	0111	（小于设置）
lw	0011	x	1010	Add unsigned
sw	0011	x	1010	Add unsigned
addiu	0011	x	1010	Add unsigned
addi	1001	x	0010	Add （加）

```
`timescale 1ns / 1ps
module aluctr(input [3:0] ALUOp,
              input [5:0] funct,
              output reg [3:0] ALUCtr);

    always @(ALUOp or funct) //  如果操作码或者功能码变化执行操作
        casex({ALUOp, funct}) // 拼接操作码和功能码便于下一步的判断
            10'b0011xxxxxx: ALUCtr = 4'b1010; // lw , sw , addiu
            10'b1001xxxxxx: ALUCtr = 4'b0010; // addi
            10'b0101xxxxxx: ALUCtr = 4'b0110; // beq
            10'b0110xxxxxx: ALUCtr = 4'b0011; // bne
            10'b0000100000: ALUCtr = 4'b0010; // add
            10'b0000100010: ALUCtr = 4'b0110; // sub
            10'b0000100100: ALUCtr = 4'b0000; // and
            10'b0000100101: ALUCtr = 4'b0001; // or
            10'b0000101010: ALUCtr = 4'b0111; // slt
            10'b1011xxxxxx: ALUCtr = 4'b1011; // sltiu
            10'b0000000000: ALUCtr = 4'b1000; // sll
            10'b0100xxxxxx: ALUCtr = 4'b0111; // slt
            10'b0111xxxxxx: ALUCtr = 4'b1101; // lui
            10'b0010xxxxxx: ALUCtr = 4'b0001; // ori
            10'b1010xxxxxx: ALUCtr = 4'b0111; // bltz
            10'b0001xxxxxx: ALUCtr = 4'b0000; // andi
            10'b1000xxxxxx: ALUCtr = 4'b0100; // xori
            10'b0000100111: ALUCtr = 4'b0101; // nor
            10'b0000000010: ALUCtr = 4'b1001; // srl
            default:ALUCtr         = 4'b1111;
            // still available: 1010, 1100, 1110
        endcase
endmodule
```

4 ALU 运算器模块

ALU 的逻辑框图如图所示。



- ✧ **在图中各信号的功能：**
- ✧ input1：操作数，32 位，输入；
- ✧ input2：操作数，32 位，输入；
- ✧ ALUCtr：4 位操作码，输入；
- ✧ aluRes：运算结果，32 位，输出；
- ✧ zero：零标志，1 位；当运算结果为 0 时，该位为 1，否则为 0；

```
`timescale 1ns / 1ps
module alu(input [4:0] shamt,
           input [31:0] input1,
           input [31:0] input2,
           input [3:0] aluCtr,
           output reg[31:0] aluRes,
           output reg ZF,
           output reg CF,OF,
           output reg Branch_ctr );
always@(*) // 运算数或控制码变化时操作
begin
case(aluCtr)

4'b0000: // and
begin
    aluRes = input1 & input2; if (aluRes == 0) ZF = 1; else ZF = 0;
    OF = 0; CF = 0; Branch_ctr = 0;
end

4'b0001: // or
begin
    aluRes = input1 | input2; if (aluRes == 0) ZF = 1; else ZF = 0;
    OF = 0; CF = 0; Branch_ctr = 0;
end

4'b0010: // 加 addi add
begin
    {CF,aluRes} = $signed(input1) + $signed(input2); if (aluRes == 0)
    ZF = 1; else ZF = 0;
    OF          = input1[31]^input2[31]^aluRes[31]^CF;//溢出标志公式
    Branch_ctr = 0;
```

```
end

4'b1010: // 加 addiu lw sw
begin
   {CF,aluRes} = input1 + input2; if (aluRes == 0) ZF = 1; else ZF = 0;
   OF = 0;//溢出标志公式
   Branch_ctr = 0;
end

4'b0110: // 减 sub
begin
    {CF,aluRes} = input1 - input2; if (aluRes == 0)
    ZF = 1; else ZF = 0;
    OF = input1[31]^input2[31]^aluRes[31]^CF;//溢出标志公式
    if ( aluRes == 0 ) Branch_ctr = 1; else Branch_ctr = 0;
end

4'b0101: // 或非 nor
begin
    aluRes  = ~(input1 | input2); if (aluRes == 0) ZF  = 1;
    else ZF = 0;
    OF = 0; CF = 0;
    Branch_ctr = 0;
end
4'b0111: // 小于设置 slt
begin
    if ($signed(input1)<$signed(input2)) aluRes = 1;
    //($signed(a) < $signed(b));
    else aluRes = 0; if (aluRes == 0) ZF = 1;  else ZF = 0;
    if ( aluRes == 0 ) Branch_ctr = 0; else Branch_ctr = 1;
    OF = 0; CF = 0;
end

4'b1011: // 小于设置 sltiu
begin
    if ($signed(input1)<input2) aluRes = 1;
    //($signed(a) < $signed(b));
    else aluRes = 0; if (aluRes == 0) ZF = 1;  else ZF = 0;
    if ( aluRes == 0 ) Branch_ctr = 0; else Branch_ctr = 1;
    OF = 0; CF = 0;
end

4'b1000://sll
begin
    aluRes = input2<<shamt; if (aluRes == 0) ZF = 1; else ZF = 0;
    OF = 0; CF = 0; Branch_ctr = 0;
end
```

```
4'b1001://srl
begin
    aluRes = input2>>shamt; if (aluRes == 0) ZF = 1; else ZF = 0;
    OF = 0; CF = 0; Branch_ctr = 0;
end

4'b0011://bne
begin
    aluRes = input1-input2;
    if (aluRes == 0) ZF = 0;//这里的 zero 是指不为 0，不相等
    else ZF = 1;
    if ( aluRes == 0 ) Branch_ctr = 0; else Branch_ctr = 1;
    OF = 0; CF = 0;
end

4'b0100://xor
begin
  aluRes = (~input1&input2)|(input1&~input2); if (aluRes == 0) ZF  = 1;
  else ZF = 0;
  OF = 0; CF = 0; Branch_ctr = 0;
end

4'b1101://lui
begin
    aluRes = {input2[15:0],16'b0000_0000_0000_0000};
    OF = 0; CF = 0; ZF = 0; Branch_ctr = 0;
end
default:
begin
    aluRes = 0; ZF = 0;OF = 0; ZF = 0; Branch_ctr = 0;
end
endcase
end

endmodule
```

5 符号扩展模块

```
`timescale 1ns / 1ps
module signext(input [15:0] inst,   // 输入 16 位
               input ExtOp,
               output [31:0] data); // 输出 32 位
    // 根据符号位补充符号
    //  如果符号位为 1，则补充 16 个 1，即 16'h ffff
    // 如果符号位为 0，则补充 16 个 0
    assign data = inst[15:15]&ExtOp?{16'hffff,inst}:{16'h0000,inst};
endmodule
```

6 指令存储器模块。

该模块有 4 位地址输入和 32 位数据输出，首先将 16 种类型的 16 条指令写入存储单元中，然后根据 4 位地址输入选择相应的单元指令内容，将数据写入到输出变量中。地址输入数据、存储单元编号与指令的对应规则如下表所示：

由于我们的主要目的是对所设计的控制器进行测试，故可以不使用 IP 核设计指令存储器，直接使用通用代码实现。

```
reg [31:0] regs [0:255]; // 寄存器组
```

根据输入，读取相应单元的的值输出，对相应存储单元进行写操作。

```
`timescale 1ns / 1ps
module IM_unit(input clk,
               input [7:0] Addr,
               output reg [31:0] instruction);
 //寄存器地址都是 4 位二进制数，因为寄存器只有 16 个，4 位就能表示所有寄存器
   reg [31:0] regs [0:255]; // 寄存器组
   initial begin   //此处为绝对地址，注意斜杠方向
   $readmemh("D:/Vivado/CPU/CPU.srcs/sources_1/single_cpu.dat", regs);
   end
   always @(posedge clk) // 时钟上升沿操作
   instruction = regs[Addr] ; //  取指令
endmodule
```

7 数据存储器（LW,SW）

```
`timescale 1ns / 1ps
module DM_unit(input clk,
               Wr,
               input reset,
               input [31:0] DMAdr,
               input [31:0] wd,
               output [31:0] rd);
    reg [31:0] RAM[15:0];
    //read
    assign rd = RAM[DMAdr>>2];
    //write
    integer i;
    always @ (posedge clk,posedge reset)
    begin
        if (reset)begin
```

```
            for(i = 0; i < 16; i = i + 1)
                RAM[i] = 0;
        end
        else if (Wr) begin
            RAM[DMAdr>>2] = wd;
        end
    end
endmodule
```

8 数码管显示模块

七段译码显示的内容是 16 位的，而 ALU 的运算结果是 32 位的，将运算结果分为高十六位和低十六位，分别传进七段译码模块；在顶层模块可用一个开关，来选择是显示高 16 位还是低 16 位；

```
assign disp_num = (SW[2] == 1)? data[31:16]:display_content[15:0];
```

sm_wei 选择哪一个数码管亮，sm_duan 选择数码管的哪一段亮，sm_wei 变换的速度是 1 秒 1000 次，使人眼看起来数码管是同时显示数值的。

```
`timescale 1ns / 1ps
module display(clk,
               data,
               sm_wei,
               sm_duan);
    input clk;
    input [15:0] data;
    output [3:0] sm_wei;
    output [6:0] sm_duan;
    //分频
    integer clk_cnt;
    reg clk_400Hz;
    always @(posedge clk)
        if (clk_cnt == 32'd100000)

    begin clk_cnt    <= 1'b0; clk_400Hz    <= ~clk_400Hz;
    end else clk_cnt <= clk_cnt + 1'b1;
    //位控制

    reg [3:0]wei_ctrl = 4'b1110;
    always @(posedge clk_400Hz)

    wei_ctrl <= {wei_ctrl[2:0],wei_ctrl[3]}; //位控制
    reg [3:0]duan_ctrl;

    always @(wei_ctrl or data)
        case(wei_ctrl)
            4'b1110:duan_ctrl = data[3:0];
```

```
            4'b1101:duan_ctrl = data[7:4];
            4'b1011:duan_ctrl = data[11:8];
            4'b0111:duan_ctrl = data[15:12];
            default:duan_ctrl = 4'hf;
        endcase

    //解码模块
    reg [6:0]duan;
    always @(duan_ctrl)
        case(duan_ctrl)
            4'h0:duan      = 7'b100_0000;//0
            4'h1:duan      = 7'b111_1001;//1
            4'h2:duan      = 7'b010_0100;//2
            4'h3:duan      = 7'b011_0000;//3
            4'h4:duan      = 7'b001_1001;//4
            4'h5:duan      = 7'b001_0010;//5
            4'h6:duan      = 7'b000_0010;//6
            4'h7:duan      = 7'b111_1000;//7
            4'h8:duan      = 7'b000_0000;//8
            4'h9:duan      = 7'b001_0000;//9
            4'ha:duan      = 7'b000_1000;//a
            4'hb:duan      = 7'b000_0011;//b
            4'hc:duan      = 7'b100_0110;//c
            4'hd:duan      = 7'b010_0001;//d
            4'he:duan      = 7'b000_0110;//e
            4'hf:duan      = 7'b000_1110;//f
            // 4'hf:duan   = 7'b111_1111;//不显示
            default : duan = 7'b100_0000;//0
        endcase

    //-------------------------------------------------------------
    assign sm_wei  = wei_ctrl;
    assign sm_duan = duan;
endmodule
```

9 PC 模块

```
`timescale 1ns / 1ps
module NPC(
    input reset,
    input Branch, Jump, Clk,
    input Branch_ctr,   //for blez
    input halt,
    input [31:0] extimm,
    input [25:0] target,
    inout [31:0] PC
);
```

```
    wire [31:0] tmp1,tmp2,tmp3,tmp4,nextPC;
    wire tmp;
    wire [31:0]NPC;
    assign tmp=Branch&Branch_ctr;
    assign tmp1=(halt)?PC:PC+4;
    assign tmp2=(extimm<<2)+PC+4;
    assign tmp3=(tmp)?tmp2:tmp1;
    assign tmp4={PC[31:28],target, {2{1'b0}}};
    assign nextPC=(Jump)?tmp4:tmp3;
    PCCount pcc( nextPC, Clk, NPC);
    assign PC = (reset)? 0:NPC;
endmodule

`timescale 1ns / 1ps
module PCCount(
    input[31:0] nextPC,
    input Clk,
    output[31:0] NPC
);
    reg[31:0] PC;
    always @(negedge Clk)
    begin
        PC<=nextPC;
    end
    assign NPC = PC;
endmodule
```

10 主模块

TOP 文件，组装各个模块

顶层模块需要将前面的多个模块实例化后，通过导线以及多路复用器将各部件链接起来。顶层模块需要输入时钟和复位信号，然后首先读取 IM_unit 的机器指令，然后通过各个模块执行。

完成顶层模块后就可以进行仿真来验证自己的基础指令功能或额外添加功能是否正确，所以在这里我输出了大量信号量和中间值来观察，以免在出现错误时可以及时观察对应值的结果是否符合预期，便于发现错误。

```
`timescale 1ns / 1ps
module top(
    input clkin,
    input clk_system,
    input reset,
    input [2:0] SW,        //开关
    output [6:0] sm_duan,//段码
```

```
output [3:0] sm_wei,//哪个数码管
output [31:0] aluRes,
output [31:0] instruction,
output [31:0] PC,
output branch, jmp,
output [31:0] expand,
output ZF,
output OF,
output Branch_ctr,
//output [31:0] tmp1,tmp2,tmp3,tmp4,nextPC,
output [4:0] shamt,
output [31:0] input1,
output [31:0] input2,
output [3:0] aluCtr
//output [31:0] memreaddata
//output [15:0] disp_num
);

wire halt = (instruction == 32'b11111100000000000000000000000000)? 1:0;

NPC NPC(
    .reset(reset),
    .Branch(branch),
    .Jump(jmp),
    .Clk(clkin),
    .Branch_ctr(Branch_ctr),
    .halt(halt),
    .extimm(expand),
    .target(instruction[25:0]),
    .PC(PC)
);

// 例化指令存储器
IM_unit IM(
    .clk(clkin),
    .Addr(PC[9:2]),
    .instruction(instruction)
);

// CPU 控制信号线
wire reg_dst, alu_src, memtoreg, regwrite;
wire memread, memwrite, ExtOp;
wire [3:0] aluop;
//wire branch, jmp;

// 实例化控制器模块
ctr mainctr(
    .opCode(instruction[31:26]),
```

```
        .regDst(reg_dst),
        .aluSrc(alu_src),
        .memToReg(memtoreg),
        .regWrite(regwrite),
        .memRead(memread),
        .memWrite(memwrite),
        .branch(branch),
        .ExtOp(ExtOp),
        .aluop(aluop),
        .jmp(jmp)
    );

    // 寄存器信号线
    wire[4:0] regWriteAddr;
    wire[31:0] regWriteData;
    wire[31:0] RsData, RtData;

    assign regWriteAddr = reg_dst ? instruction[15:11] : instruction[20:16];
    //写寄存器的目标寄存器来自rt或rd
    assign regWriteData = memtoreg ? memreaddata : aluRes;
    //写入寄存器的数据来自ALU或数据寄存器

    // ..... 实例化寄存器模块
    RegFile regfile(
        //Clk, Clr, Write_Reg, R_Addr_A, R_Addr_B, W_Addr, W_Data, R_Data_A,
R_Data_B
        .Clk(!clkin),
        .Clr(reset),
        .R_Addr_A(instruction[25:21]),
        .R_Addr_B(instruction[20:16]),
        .W_Addr(regWriteAddr),
        .W_Data(regWriteData),
        .Write_Reg(regwrite),
        .R_Data_A(RsData),
        .R_Data_B(RtData)
    );

    //  实例化 ALU 控制模块
    aluctr aluctr1(
        .ALUOp(aluop),
        .funct(instruction[5:0]),
        .ALUCtr(aluCtr)
    );
    //实例化符号扩展模块
    signext
signext(.inst(instruction[15:0]),.ExtOp(ExtOp), .data(expand));

    assign shamt=instruction[10:6];
```

```
    assign input2 = alu_src ? expand : RtData;
    assign input1 = RsData;
    //ALU的第二个操作数来自寄存器堆输出或指令低16位的符号扩展

    // ..................... 实例化ALU模块
    alu alu(
        .shamt(shamt),
        .input1(RsData), //写入alu的第一个操作数必是Rs
        .input2(input2),
        .aluCtr(aluCtr),
        .ZF(ZF),
        .OF(OF),
        .CF(CF),
        .aluRes(aluRes),
        .Branch_ctr(Branch_ctr)
    );

    //数据存储器
    wire[31:0] memreaddata;

    //实例化数据存储器
    DM_unit dm(
        .clk(clkin),
        .Wr(memwrite),
        .reset(reset),
        .DMAdr(aluRes),
        .wd(RtData),
        .rd(memreaddata)
    );

    //控制数码管显示
    wire [31:0] display_content;
    wire [15:0] disp_num;
    //显示内容选择
    assign display_content = (SW[1:0] == 2'b00)? instruction:
                             (SW[1:0] == 2'b01)?  PC:
                             (SW[1:0] == 2'b10)? aluRes: memreaddata;
    //高低16位选择
    assign disp_num = (SW[2] == 1)? display_content[31:16]:
    display_content[15:0];
    //数码管输出显示
    display
Smg(.clk(clk_system),  .sm_wei(sm_wei),.data(disp_num  ),.sm_duan(sm_duan
));

endmodule
```

11 CPU_Basys模块

top 模块的简化，用来烧到Basys3电路板上。

```
`timescale 1ns / 1ps

module CPU_Basys(
    input clkin,
    input clk_system,
    input reset,
    input [2:0] SW,        //开关
    output [6:0] sm_duan,//段码
    output [3:0] sm_wei//哪个数码管
    //output [15:0] disp_num,
    //output [31:0] PC
    );

    top top(
        .clkin(clkin),
        .clk_system(clk_system),
        .reset(reset),
        .SW(SW),
        .sm_duan(sm_duan),
        .sm_wei(sm_wei)
    );
endmodule
```

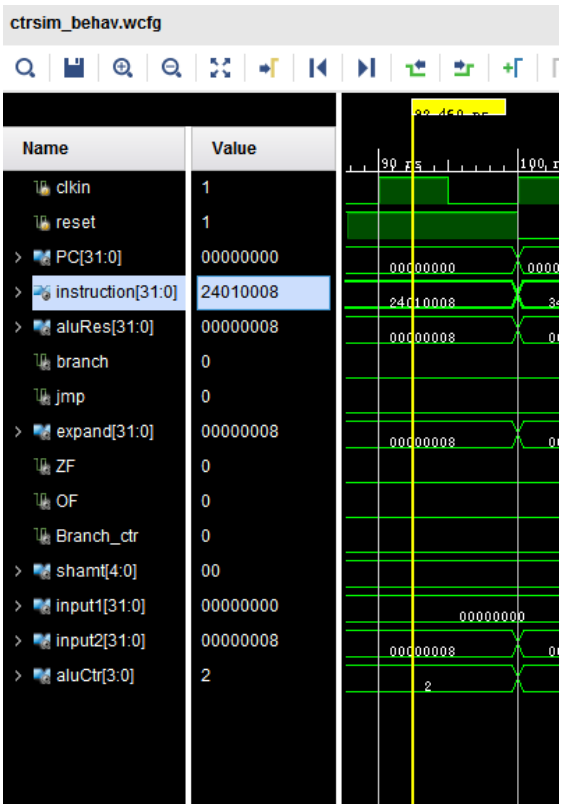
2 验证 CPU 正确性

利用 Vivado 软件的行为仿真功能，我验证了所设计的所有指令的功能及其正确性。以下为仿真的波形截图及分析。

1 Addiu 指令

如图所示，PC = 0x00000000，指令为 0x24010008，即 addiu $1,$0,8。

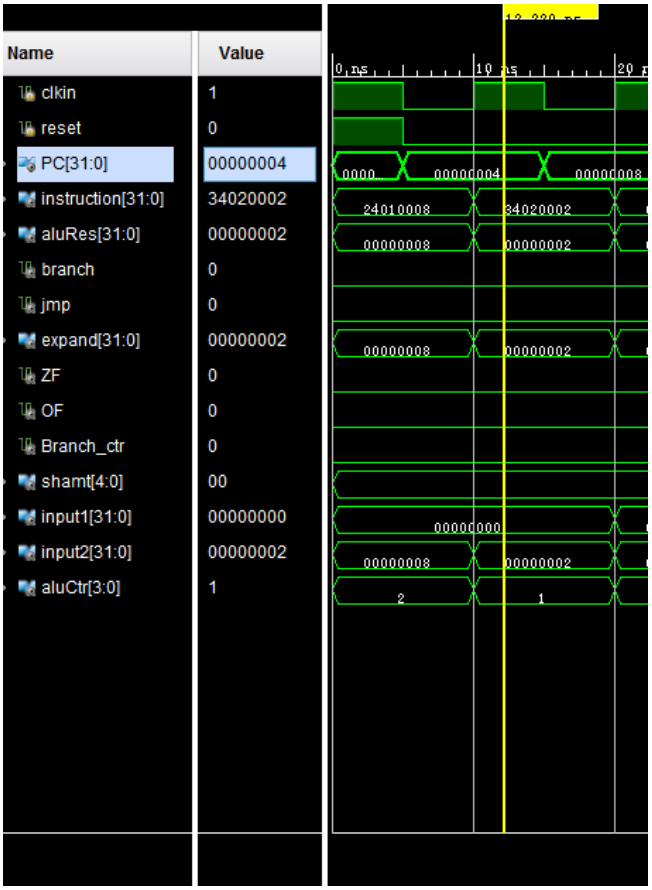
从图中可见 input1 = 0 （$0 已被初始化为 0），input2 为立即数 8，无符号数相加后 aluRes = 8，溢出标志 OF = 0。



2 ori 指令

如图所示，PC = 0x00000004，指令为 0x34020002，即 ori $2,$0,2

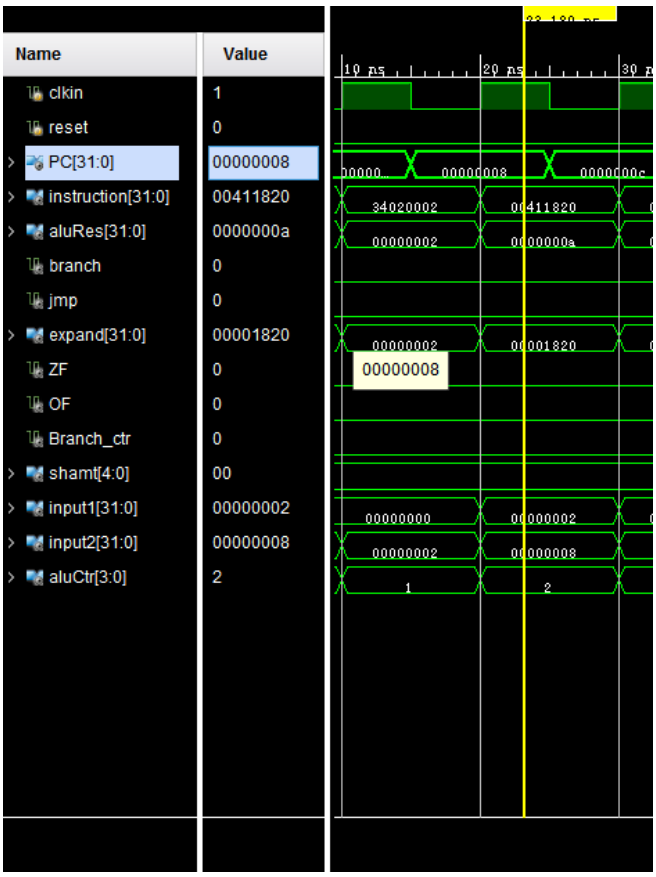
由图可知，input1 = 0（寄存器已初始化），input2 = 2（立即数），逐位或操作后 aluRes = 2。



3 add 指令

如图所示，PC = 0x00000008，指令为 0x00411820,即 add $3, $2, $1。

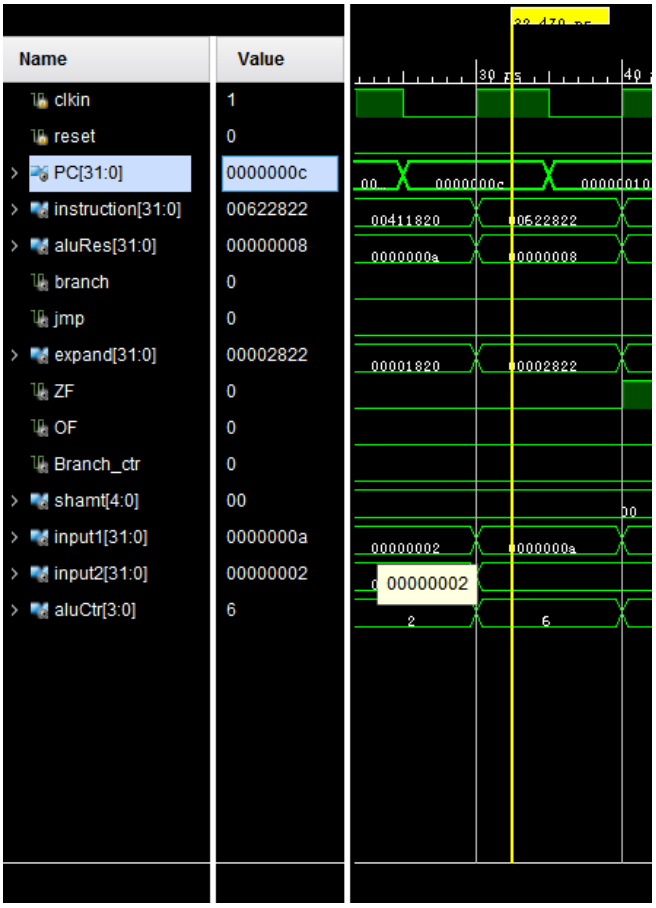
input1 = $2 = 2 （前面的指令已更新），input2 = $1 = 8，相加后 aluRes = 10 = 0xa



4 sub 指令

如图所示，PC = 0x0000000C，指令为 0x00622822，即 sub $5,$3,$2。

input1 = $3 = 0xa
input2 = $2 = 2
aluRes = 8

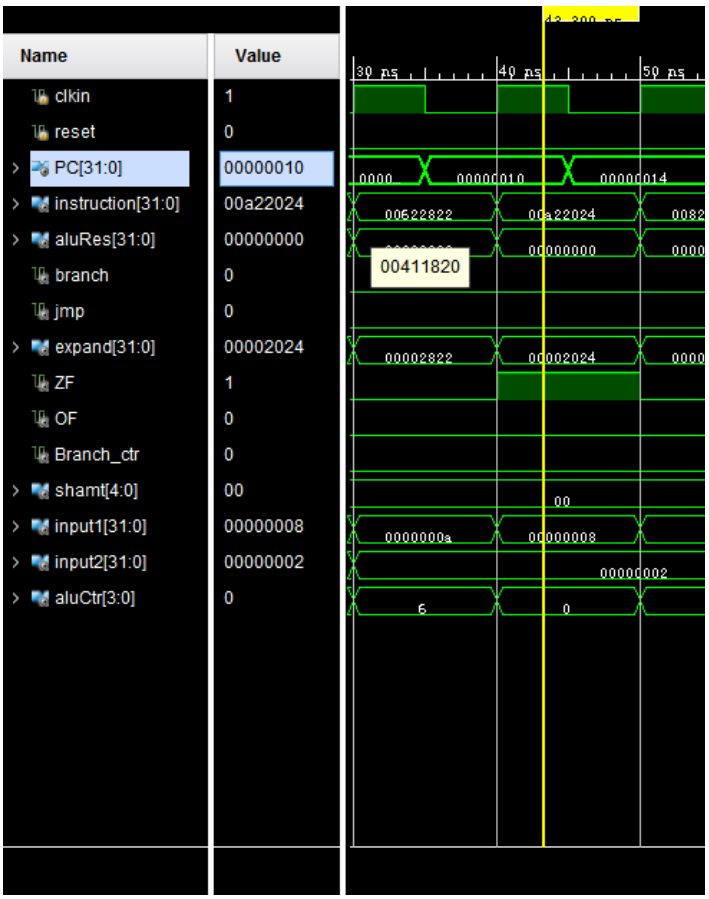


5 and 指令

如图所示，PC = 0x00000010，指令为 0x00a22024，即 and $4,$5,$2。

input1 = $5 = 8
input2 = $2 = 2
按位与操作，aluRes = 0

ALU 运算结果为零，ZF = 1

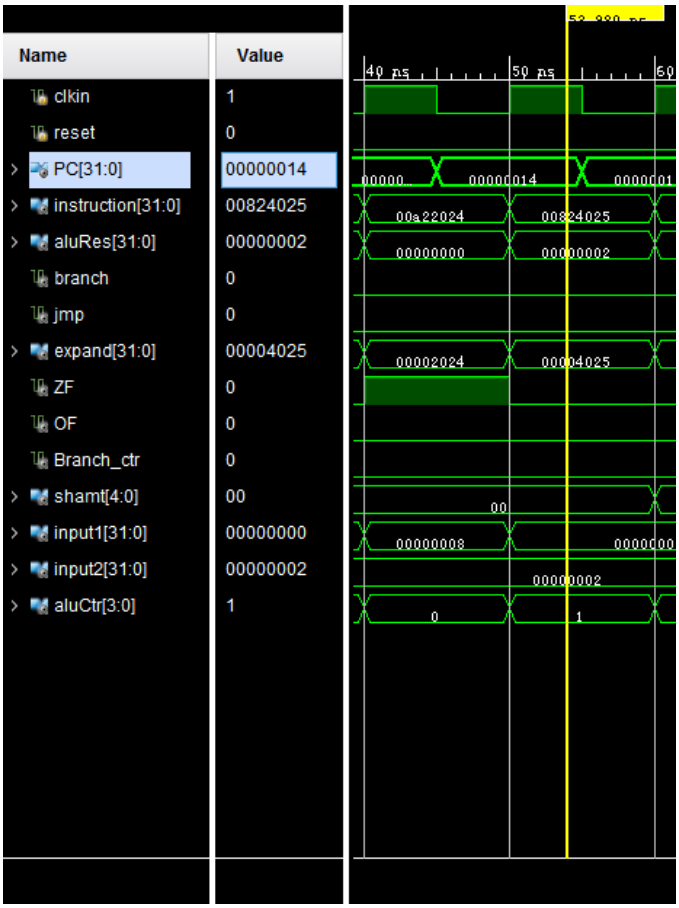


6 or 指令

如图所示，PC = 0x00000014，指令为 0x00824025，即 or $8,$4,$2。

input1 = $4= 0
input2 = $2 = 2
按位或操作，aluRes = 2

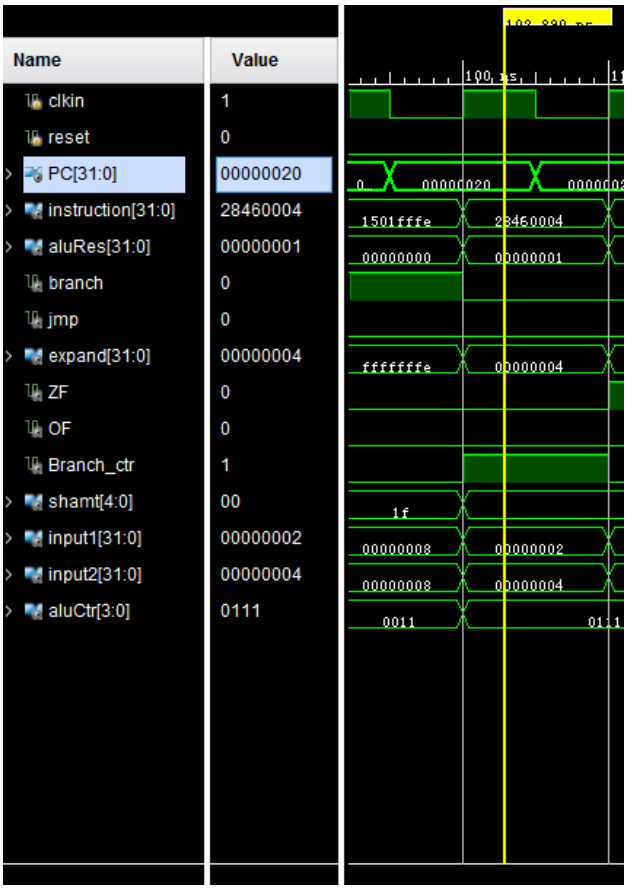
ALU 运算结果不为零，ZF = 0



7 slti 指令

如图所示，PC = 0x00000020，指令为 0x28460004，即 slti $6,$2,4。

input1 = $2 = 2
input2 = 4
判断 input1 是否小于 input2，
aluRes = 1 （input1<input2）

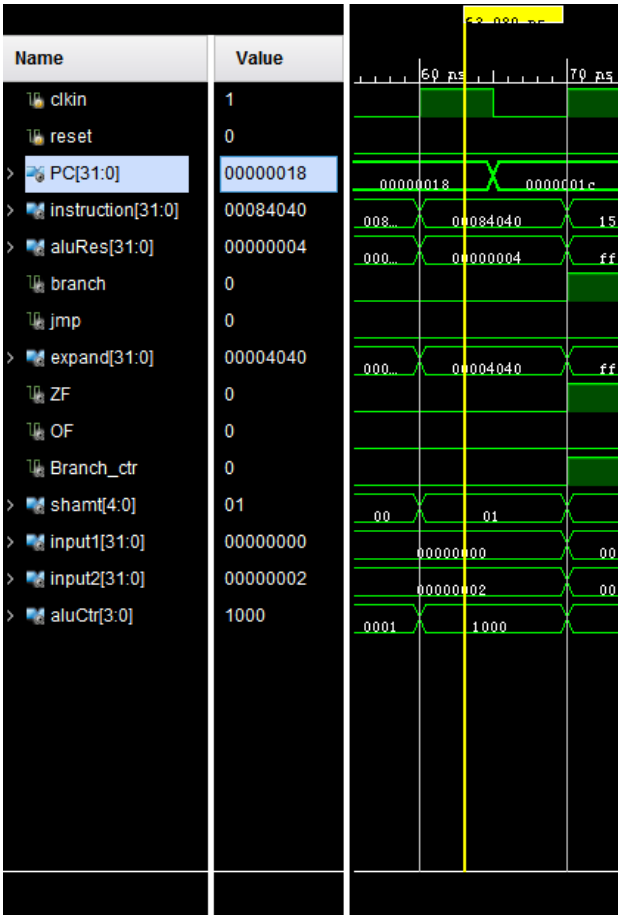


8 sll 指令

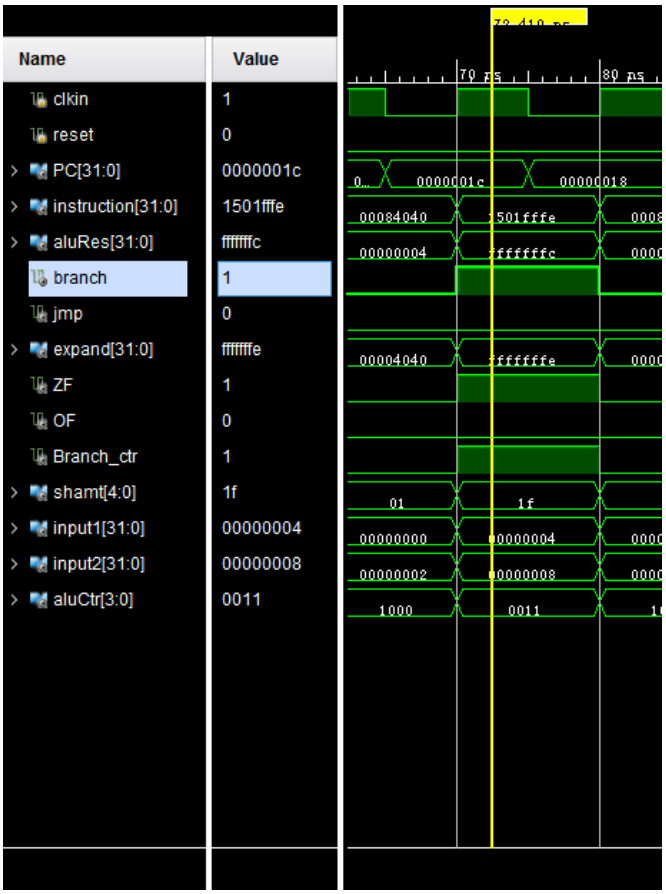
如图所示，PC = 0x00000018，指令为 0x00084040，即 sll $8,$8,1。

input1 = $8= 2
input2 = 1
按位左移操作，aluRes = 4

ALU 运算结果不为零，ZF = 0



9 bne 指令

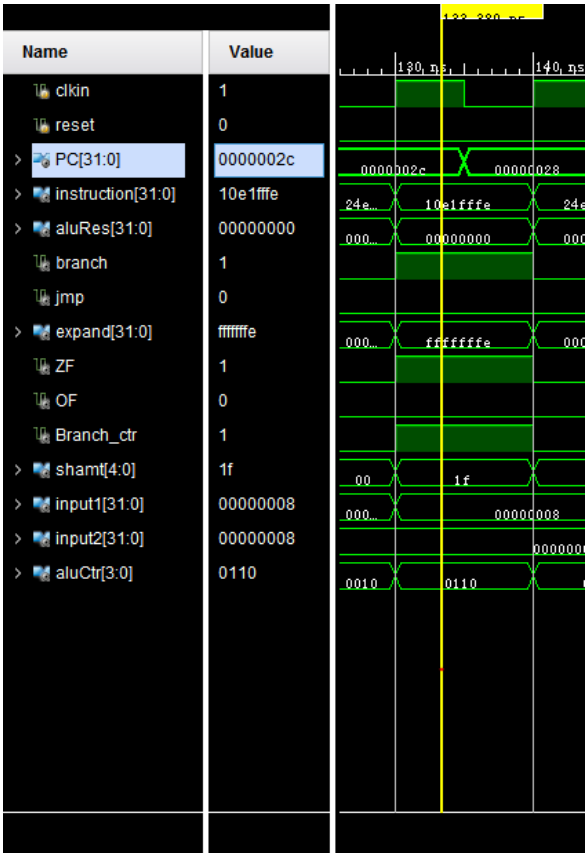


如图所示，PC = 0x0000001C，指令为 0x1501fffe，即 bne $8,$1,-2。

input1 = $8 = 4
input2 = $1 = 8
判断是否相等，aluRes = 1（不相等）

ALU 运算结果为零，ZF = 1
Branch_ctr = 1
branch = 1
条件跳转到 PC= 0x00000018

10 beq 指令



如图所示，PC = 0x0000002C，指令为 0x10e1fffe，即 beq $7,$1,-2。

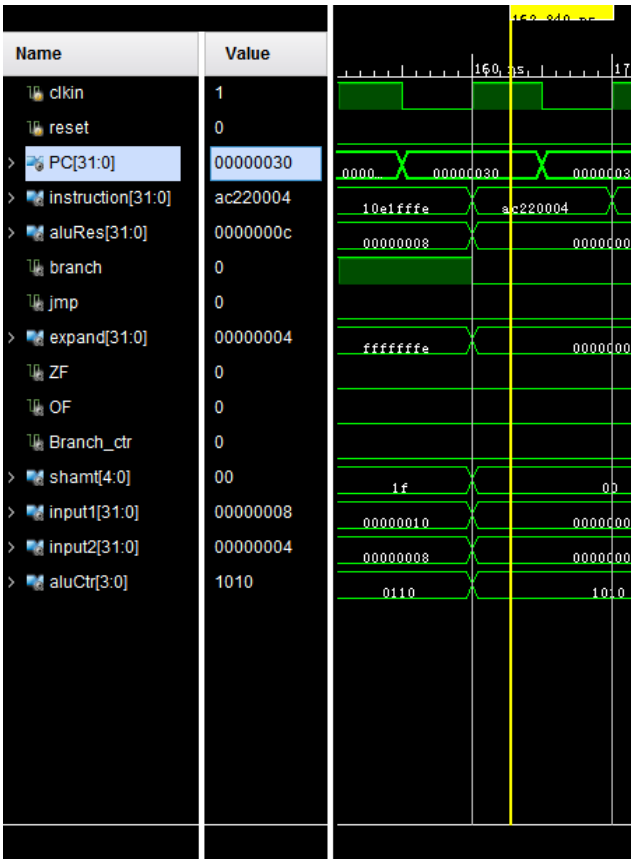
input1 = $7 = 8
input2 = $1 = 2
判断 input1 是否等于 input2，aluRes = 1 （input1==input2）

branch = 1
Branch_ctr = 1
跳转到 0x00000028

11 sw 指令

如图所示，PC = 0x00000030，指令为 0xac220004，即 sw $2, 4($1)。

input1 = $1 = 8
input2 = 4
aluRes = input1 + inpu2 = 0x0000000c

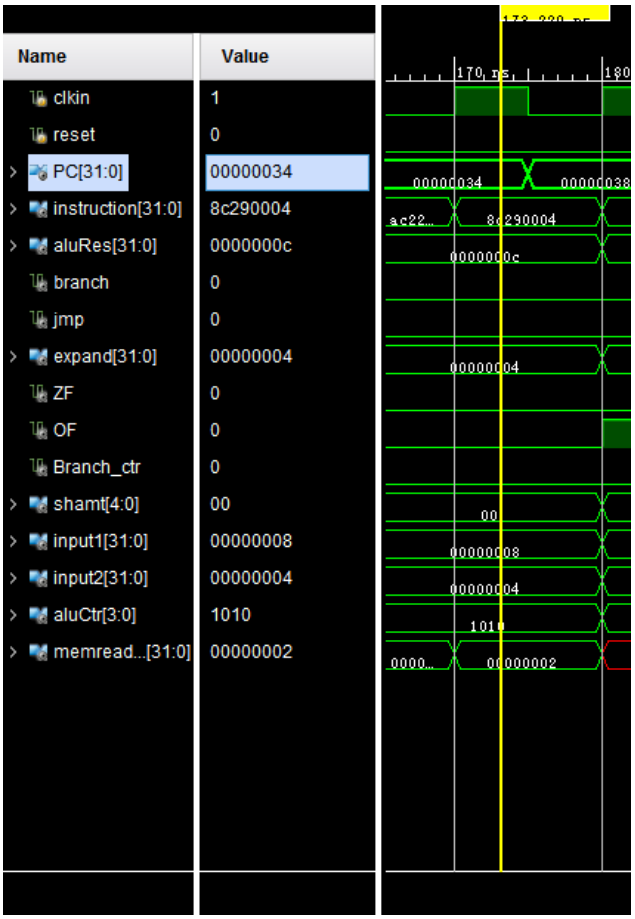


12 lw 指令

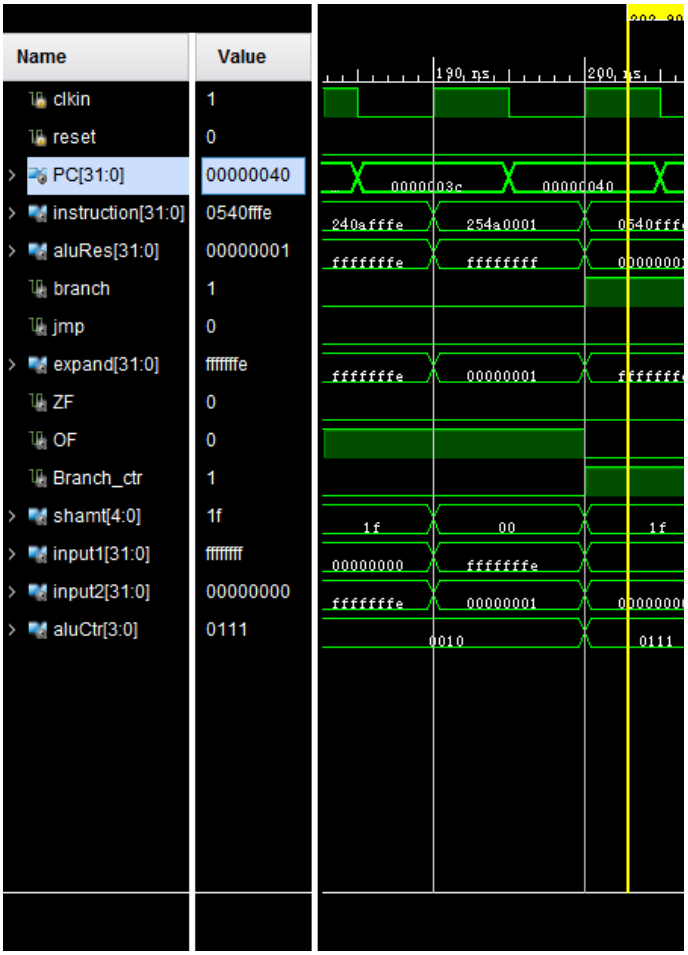
如图所示，PC = 0x00000034，指令为 0x8c290004，即 sw $2, 4($1)。

input1 = $1 = 8
input2 = 4
aluRes = input1 + inpu2 = 0x0000000c

从数据存储器 0x0c 地址读出数据，memreaddata = 2



13 bltz 指令

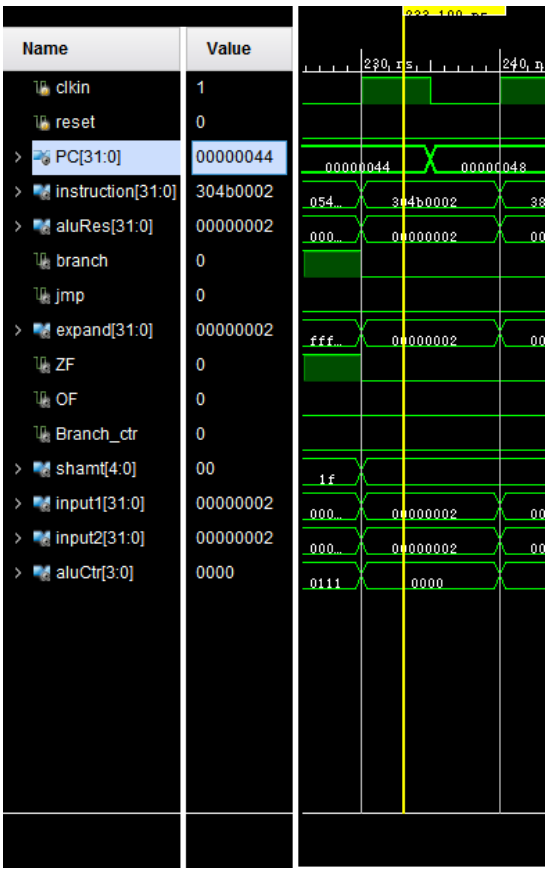


如图所示，PC = 0x00000040，指令为 0x0540fffe，即 bltz $10,-2。

input1 = 0
input2 = $10 = -1
aluCtr = 0b0111
判断 input2 是否小于 0，
aluRes = 1 （ input1<0 ）

branch = 1
Branch_ctr = 1
跳转到 0x0000003c

14 andi 指令



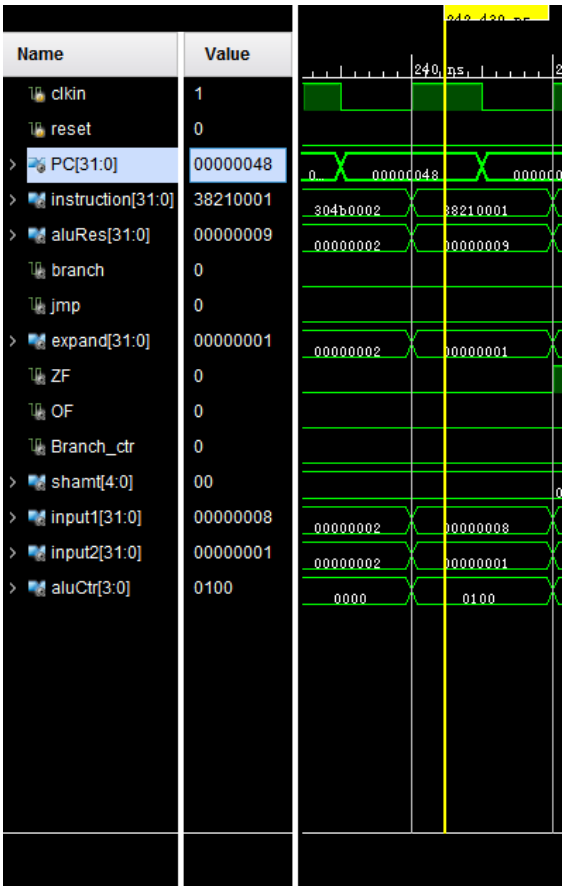
如图所示，PC = 0x00000044，指令为 0x304b0002，即 andi $11, $2, 2。

input1 = $2 = 2
input2 = 2
aluCtr = 0b0000
按位与操作，
aluRes = 2

15 xori 指令

如图所示，PC = 0x00000048，指令为 0x38210001，即 xori $1, $1, 1。

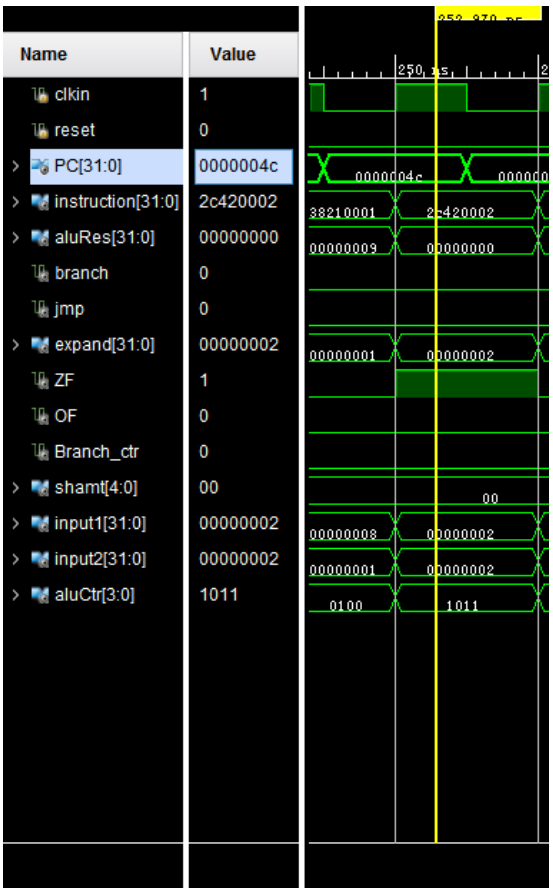
input1 = $1 = 8
input2 = 1
aluCtr = 0b0100
按位或操作，
aluRes = 9



16 sltiu 指令

如图所示，PC = 0x0000004c，指令为 0x2c420002，即 sltiu $2, $2, 2。

input1 = $2 = 2
input2 = 2
aluCtr = 0b1011
小于置 1 操作，
aluRes = 0

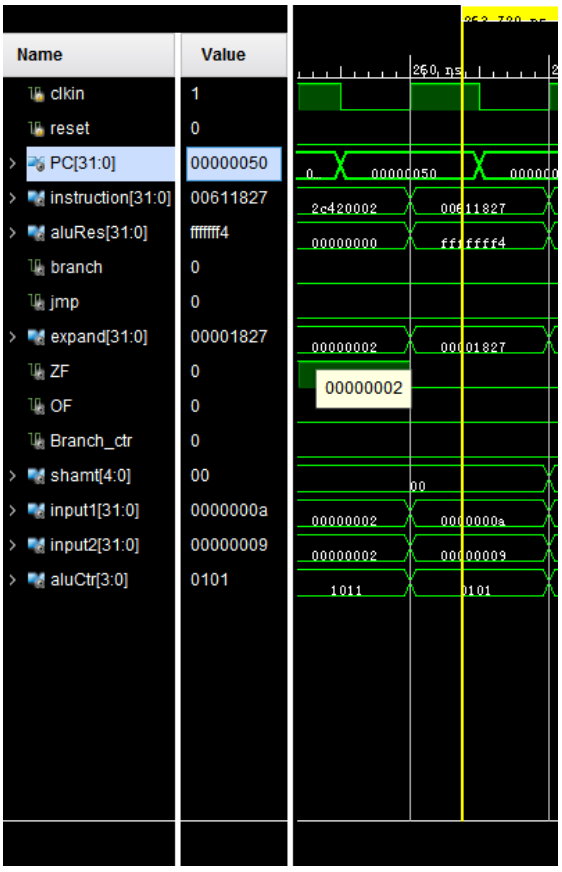


17 nor 指令

如图所示，PC = 0x00000050，指令为 0x00611827，即 nor $3, $3, $1。

input1 = $3 = 0xa
input2 = $1 = 9
aluCtr = 0b0101

按位或非操作，
aluRes = 0xffffffff4

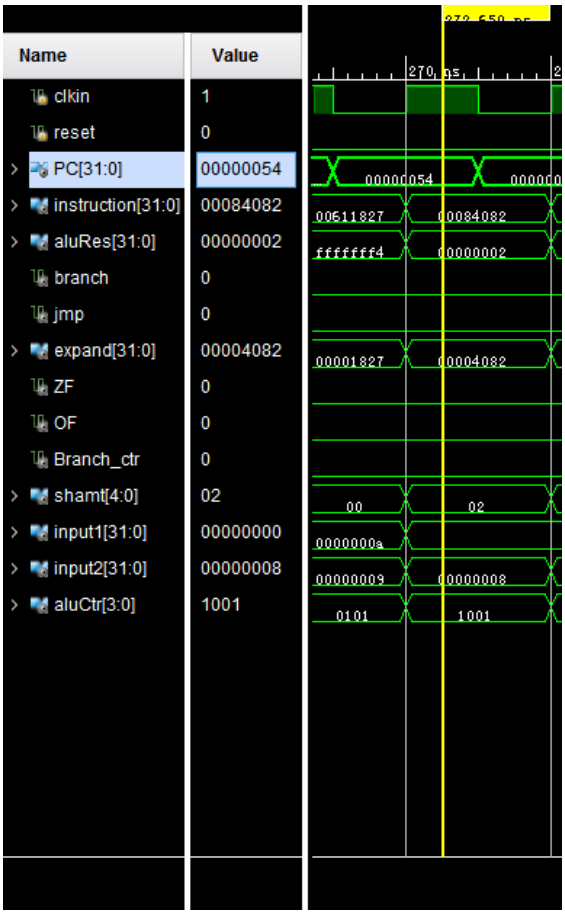


18 srl 指令

如图所示，PC = 0x00000054，指令为 0x00084082，即 srl $8, $8, 2。

input2 = $8 = 8
shamt = 2
aluCtr = 0b1001

右移（2 位）操作，
aluRes = 2

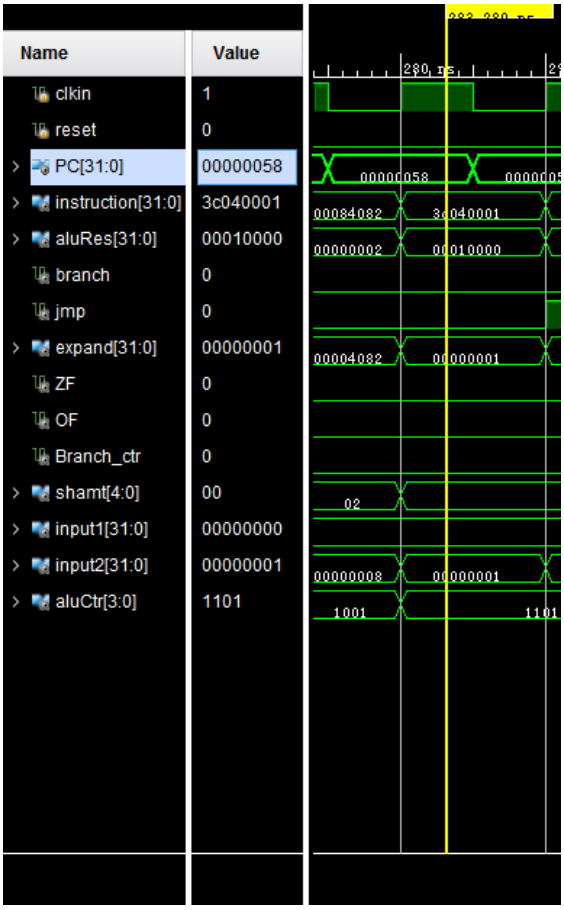


19 lui 指令

如图所示，PC = 0x00000058，指令为 0x3c040001，即 lui $4, 1。

input2 = 1
aluCtr = 0b1101

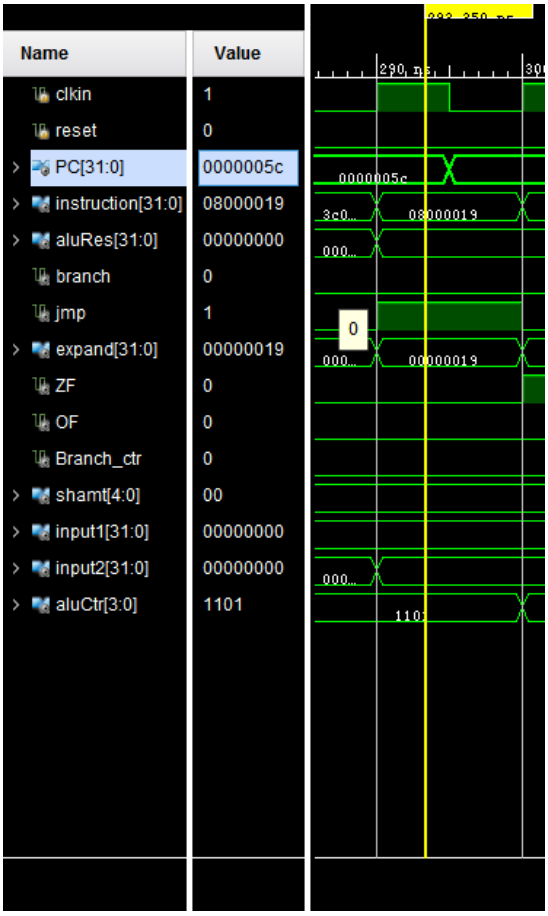
lui 操作，
aluRes = 0x00010000



20 j 指令

如图所示，PC = 0x0000005C，指令为 0x08000019，即
j 0x00000064。

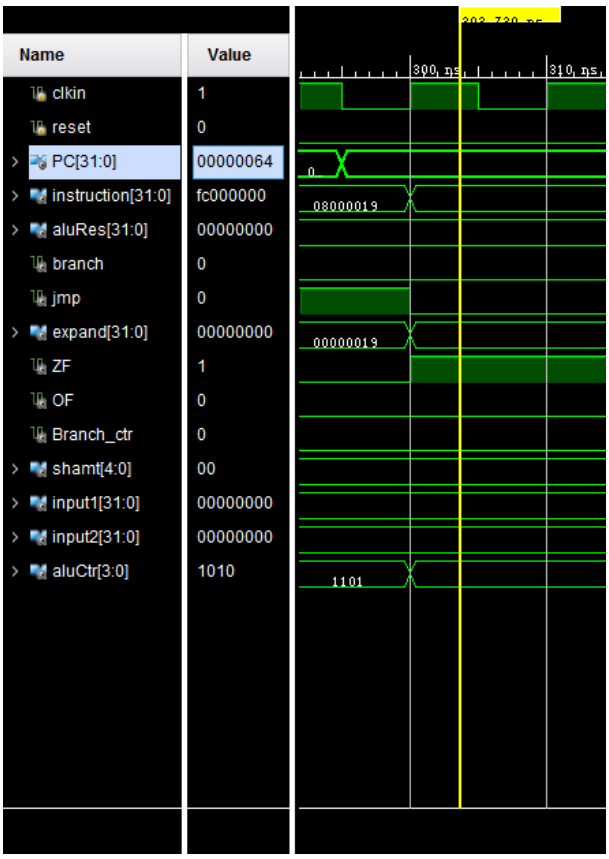
jmp = 1
跳转到地址 0x00000064



21 halt 指令

如图所示，PC = 0x00000064，指令为 0xFC000000，即
j 0x00000064。

jmp = 1
跳转到地址 0x00000064，并将 PC 停住。



3 实现

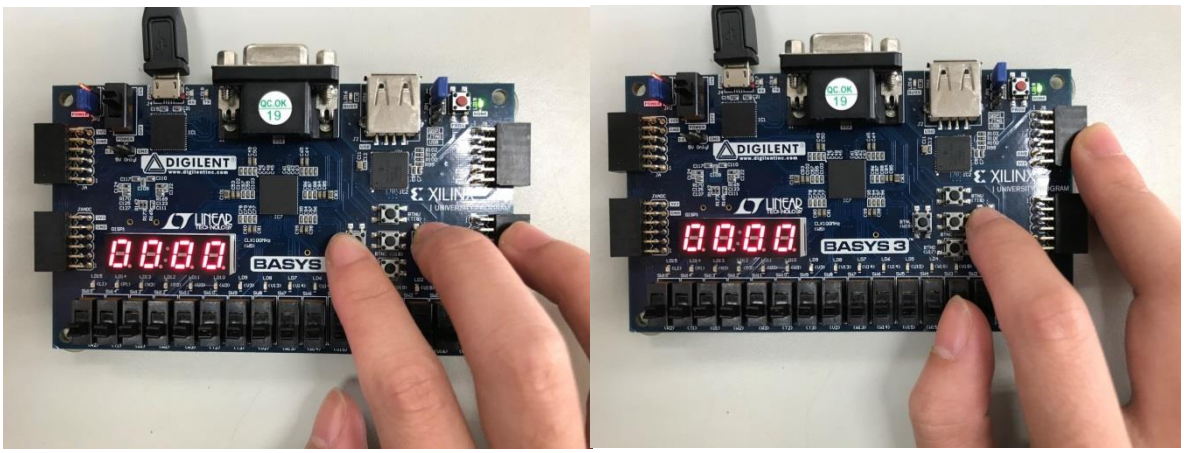
根据 top 模块中的设置，Basys 3 的数码管的显示模式如下：

```
//显示内容选择
 assign display_content = (SW[1:0] == 2'b00)? instruction:
                          (SW[1:0] == 2'b01)?  PC:
                          (SW[1:0] == 2'b10)? aluRes: memreaddata;
 //高低16位选择
 assign disp_num = (SW[2] == 1)? display_content[31:16]:
display_content[15:0];
```

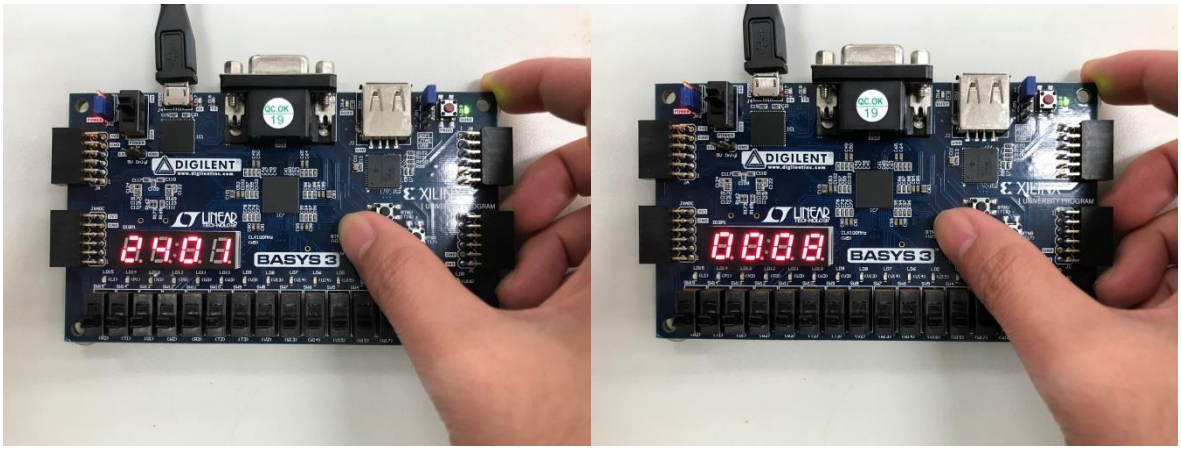
以下为仿真结果分析中的第 1-5 条指令在电路板上的实现结果，对其分析可参考仿真结果分析。

1 第 1 条指令

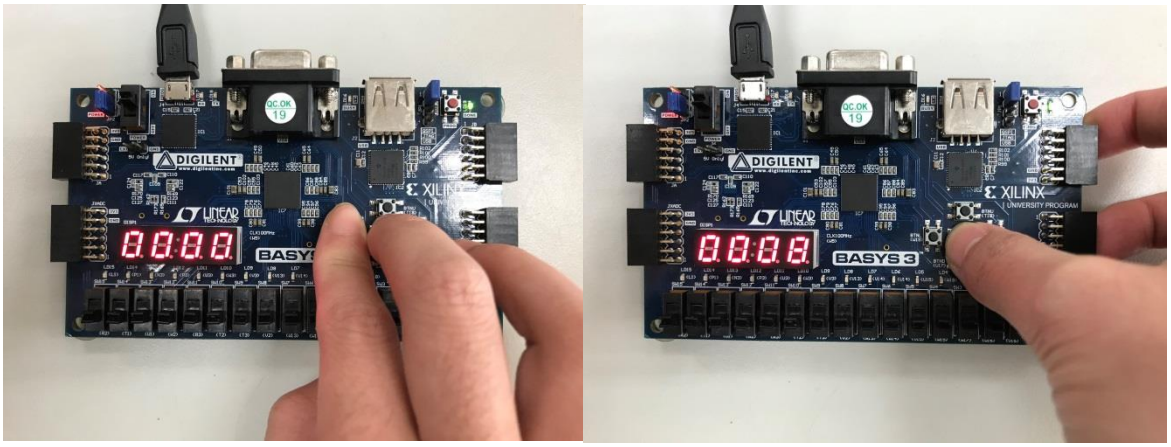
PC[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。



instruction[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

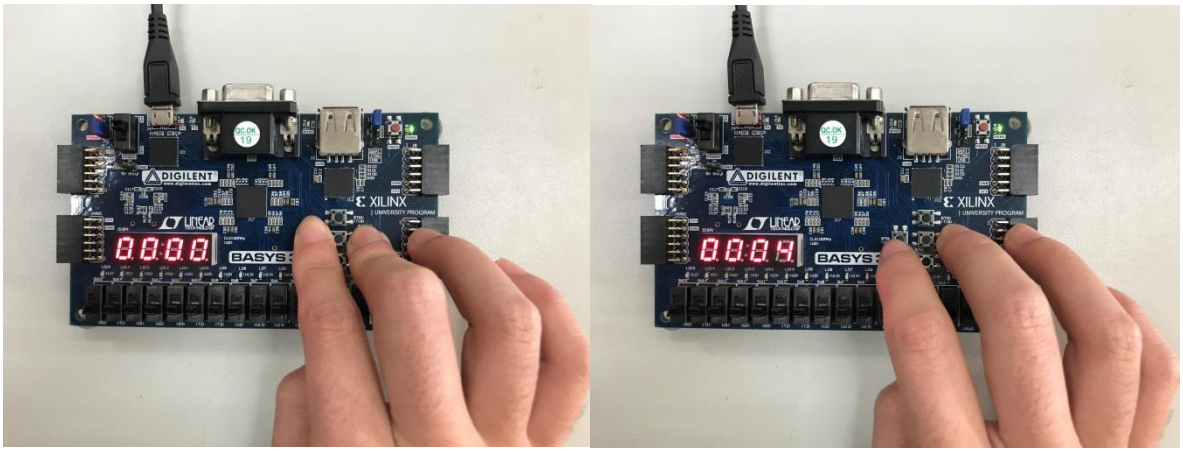


aluRes[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

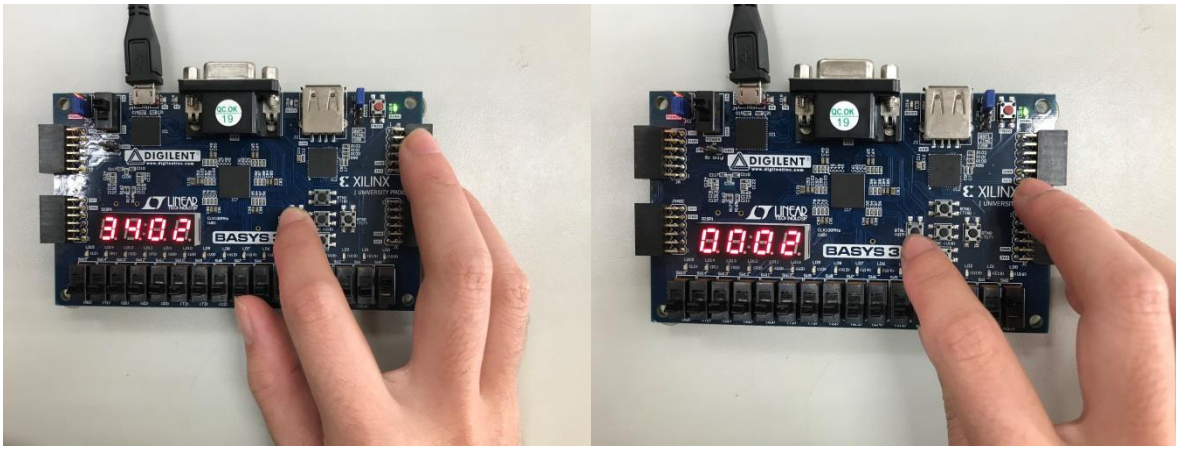


2 第 2 条指令

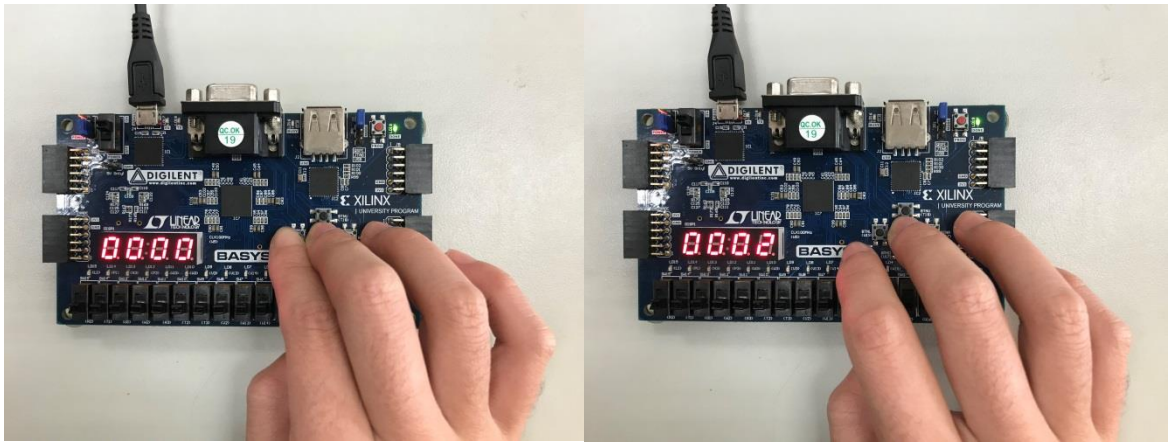
PC[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。



instruction[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

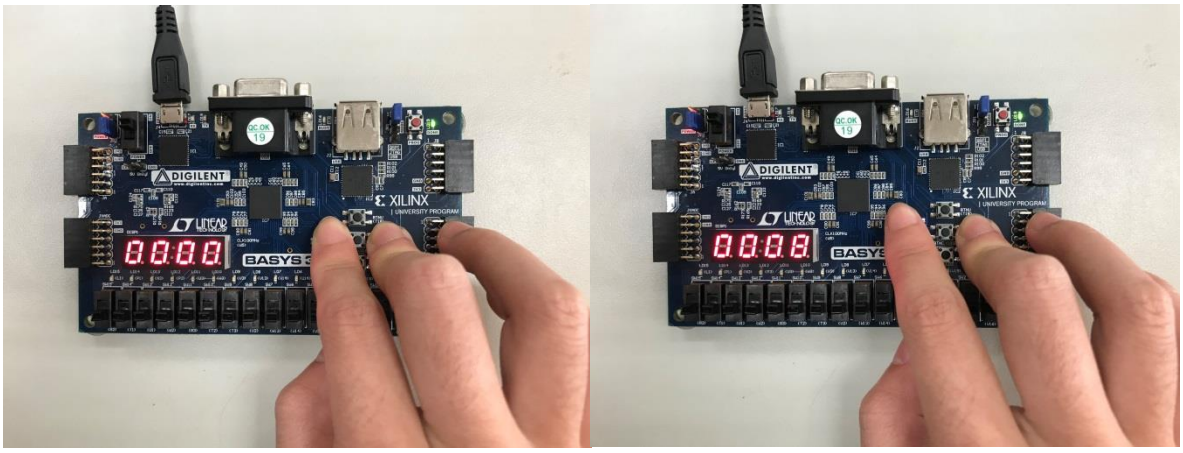


aluRes[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

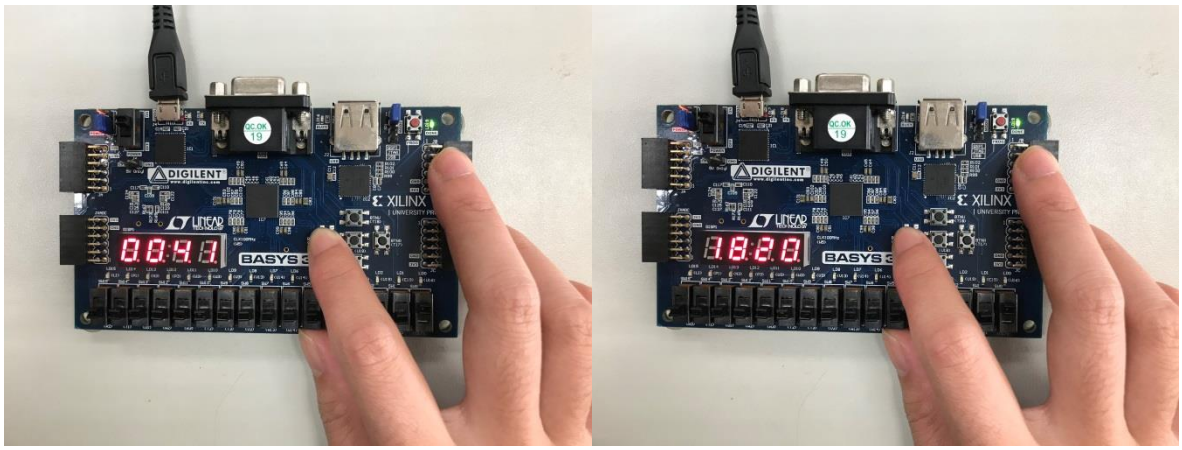


3 第 3 条指令

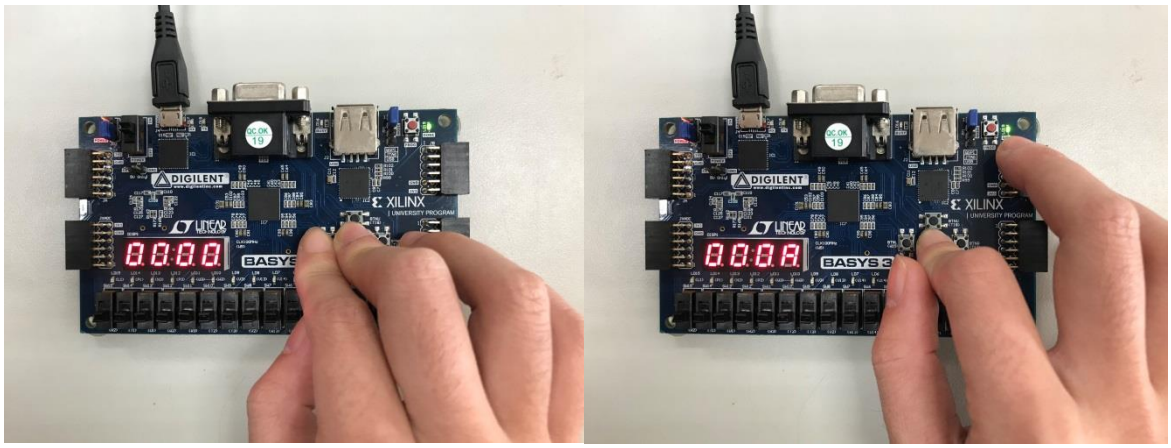
PC[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。



instruction[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

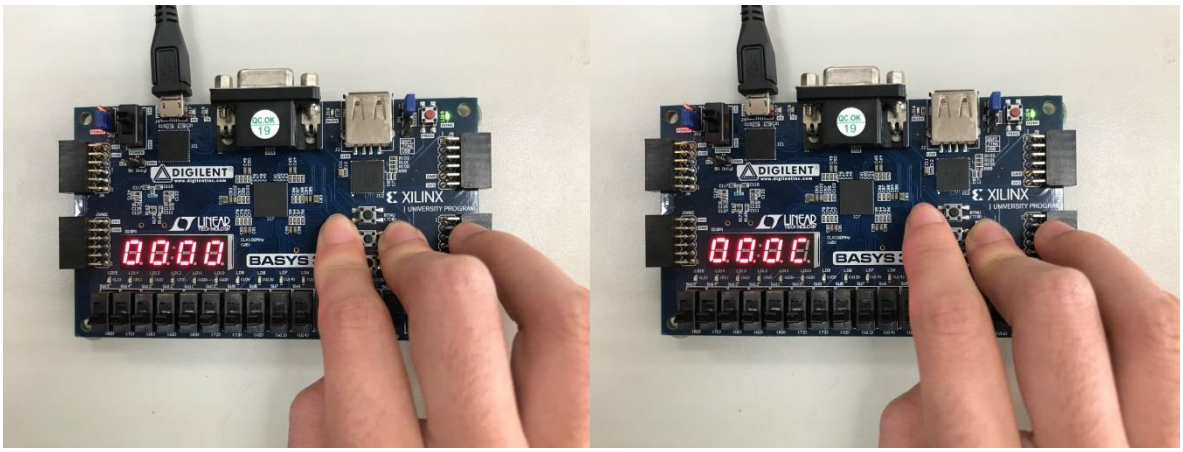


aluRes[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

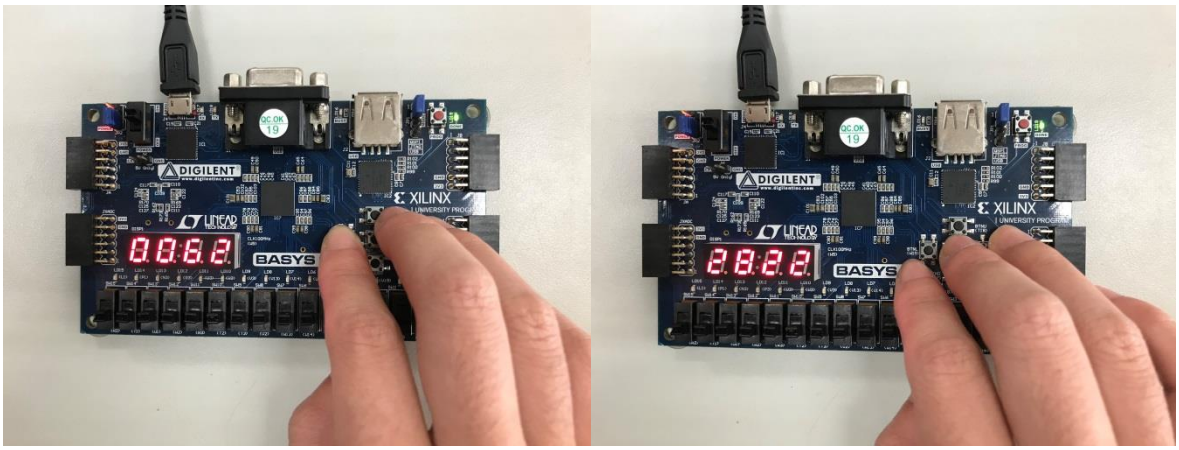


4 第 4 条指令

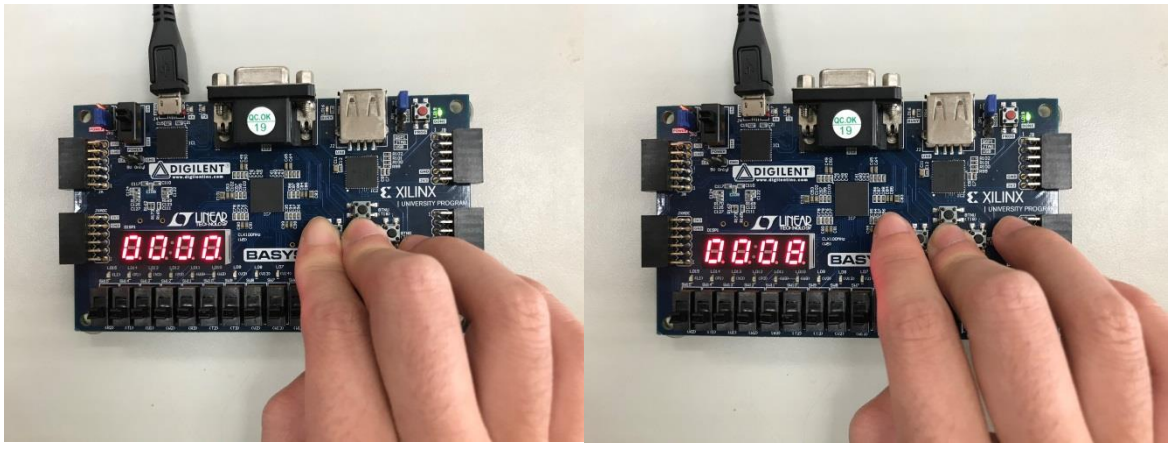
PC[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。



instruction[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

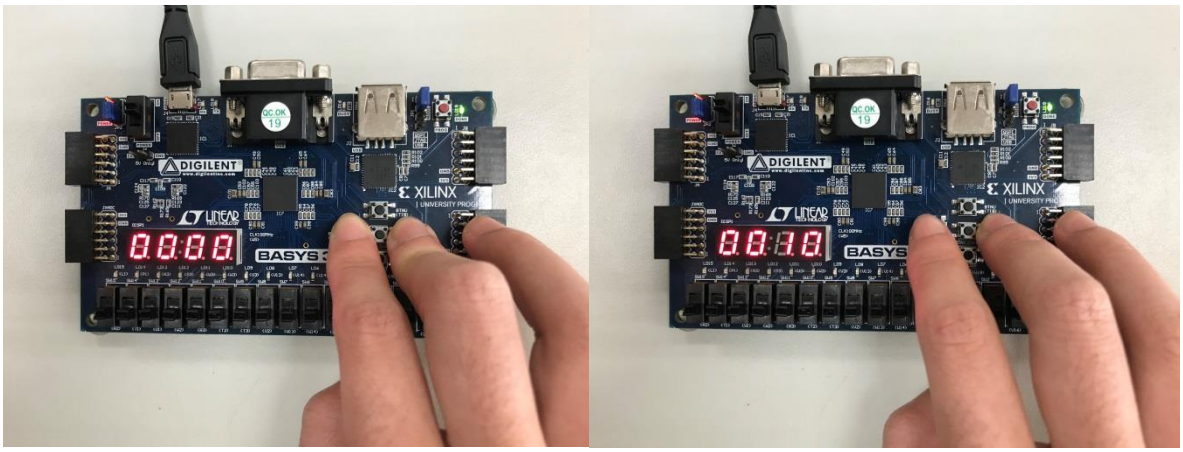


aluRes[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。

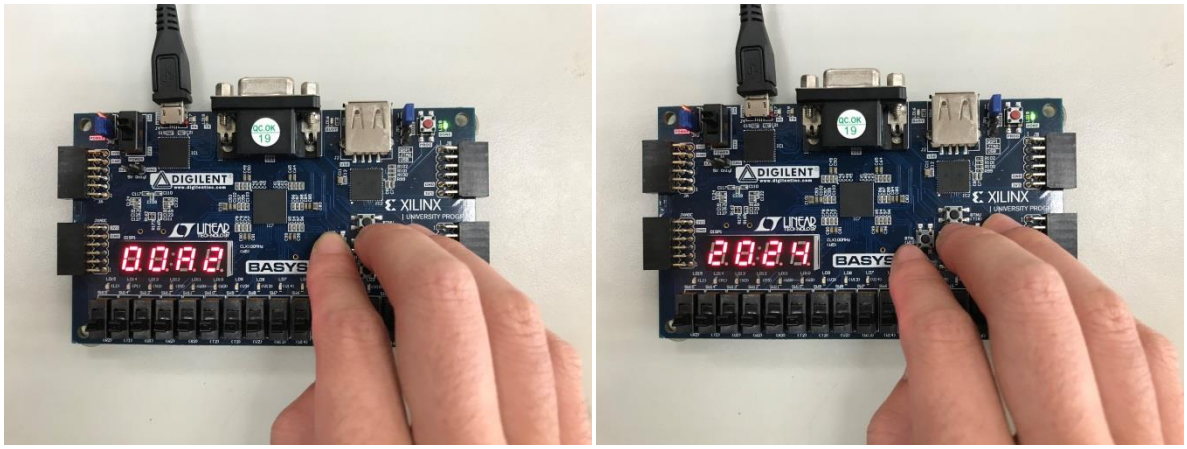


5 第 5 条指令

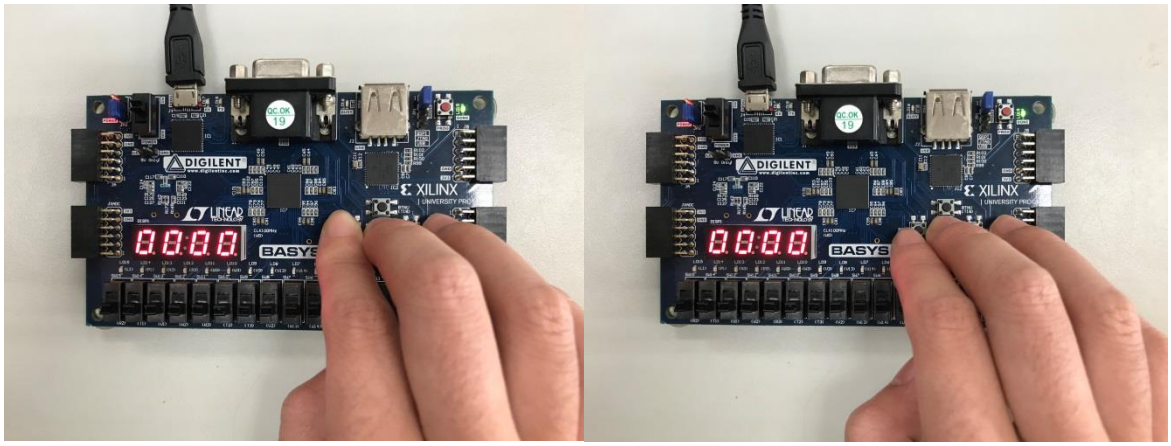
PC[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。



instruction[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。



aluRes[31:0]：下面左图为高 16 位的十六进制表示，右图为低 16 位。



六、实验心得

1 实验中遇到的问题及解决方法

a. 运行仿真时无法看见波形

在Vivado自行探索一段时间后找到解决方法：转到objects窗口。选中想要显示的信号，右键选择Show in Wave Window，即可在波形图中显示此信号。

b. 仿真运行ctrsim时有些信号出现了红色

解决方案：返回检查前面的步骤，发现opCode[5:0]的输入值不在代码定义的范围内，修改后运行成功了。

c. Vivado 自带的 Block Memory Generator 8.4 生成的 ROM 读出数据时会慢一个时钟

查阅资料发现，Block Memory总是会延时一个时钟，这不是设计中的错误。本人最终决定使用verilog定义寄存器堆的方式实现指令寄存器，以避免延迟（具体代码可见IM_Unit模块）。

参考链接：

https://forums.xilinx.com/t5/Simulation-and-Verification/ROM-delay-on-simulation-Block-memory-generator-8-4/td-p/1182685

d. [Synth 8-439] module … not found

运行综合时出现此错误。检查之前的步骤，发现top模块对其余模块进行例化时命名没有统一。修改后运行成功。

e. [Place 30-415] I/O Placement failed due to over utilization

在实现（implementation）一步遇到这一错误。查阅资料发现是设计中使用了过多端口导致的。在top模块之上再增加了一个CPU_Basys模块，精简top模块之中的输入输出端口用于烧板（保留top中的端口用于仿真）

参考链接：

https://forums.xilinx.com/t5/Implementation/Place-30-415-IO-Placement-failed-due-to-overutilization/td-p/1073764

f. Error: [Place 30-574] Poor placement for routing between an I/O pin and BUFG

在place_design一步遇到这一错误。检查之前的步骤并查阅资料发现是将CPU时钟端口设为了板子上的手动输入的端口，而不是系统默认的时钟导致的。因

为实验需要，根据链接中的指引修改了约束文件的代码：

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clkin_IBUF]
```

再次运行实现，却出现了如下错误：

CRITICAL WARNING: [Common 17-55] 'set_property' expects at least one object

再次检查并查阅资料，发现根据生成的电路图，clkin_IBUF应为clk_in，修改后运行成功。

参考链接：

https://www.xilinx.com/support/answers/64452.html

https://www.xilinx.com/support/answers/56169.html

2 体会和建议

本次实验中，我系统地复习了单周期CPU的设计思想与方法，体会到了前人的智慧与努力。同时我也锻炼了排查和解决问题，并且分析实验结果的能力，学会了如何与他人沟通，向他人请教等等。

设计单周期CPU还让我体会到了细节的重要性。尽管通常我们都是因为一些小的地方出现的问题而导致实验难以推进，但究其原因很多也是在大的方向上没有设计好，例如各个模块之间的关系与分工等等。一旦大体的结构设计清晰，中途进行大改的可能性就会降低，细节处的问题很多也就迎刃而解了。

非常感谢老师不厌其烦地回答我在实验中遇到的问题，希望老师能在我们开始设计实现前讲解更多的设计CPU的方法与经验，包括常见的错误和相应的 debug 的技巧等等，这样有助于我们在设计与调试过程中更有效率，快速找到自己的不足之处。