《计算机组成原理实验》

实验报告

学 院 名 称 ： 数据科学与计算机学院

专业（班级）： 计算机类计科 1 班

学 生 姓 名 ： 陈琮昊

学 号 ： 18340013

时 间 ： 2019 年 12 月 14 日

成 绩 ：

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. CPU的设计：

该部分是处理器的总体设计，确定处理器由哪些部分组成：运算器(ALU)，寄存器(Reg)，控制单元(CU)；定义各个期间的控制信号以及控制方法。处理器的设计要结合存储器(Mem)单元进行设计，确定控制单元对存储器的控制信号及控制方法。

2. 仿真与测试：

对各个功能部件进行仿真测试，进行 FPGA 测试，数码管显示运算结果，至少反应 4 种指令的运行结果（R 型、条件跳转、lw、ori 等）

3. 要求：

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 至少能够完成 20 条 MIPS 指令，至少包含以下指令：

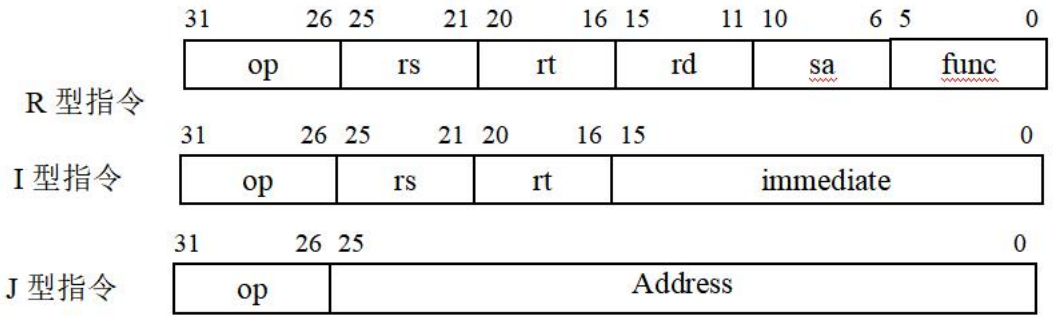
（i）支持基本的内存操作如lw，sw指令
（ii）支持基本的算术逻辑运算如add，sub，and，ori，slt，addi指令
（iii）支持基本的程序控制如beq，j指令

将其中的 ALU 运算结果在开发板数码管上显示出来。

另外可拓展添加其他指令。

MIPS 的指令格式为 32 位。图 1 给出了 MIPS 指令的 3 种格式。

图 1



其中Rs和Rt为两个源操作数寄存器，Rd为目的操作数寄存器。shamt为移位操作时的移位运算值，是一个立即数。func为R型指令的功能码。sa为位移量，移位指令用于指定移多少位。

immediate为I型指令的立即数。Address为J型指令的跳转地址。

设计所需要支持的指令集如表 1 所示：

表 1

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

三. 实验原理：

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 2。

图 2

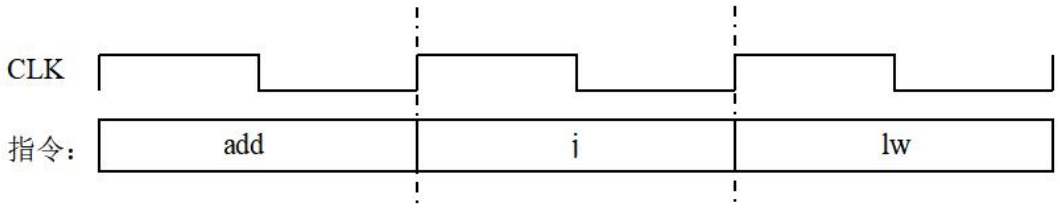


图 3 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即分为指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，而 PC 的改变则是在时钟上升沿进行。图中控制信号作用如表 2 所示，表 3 是 ALU 运算功能表。

图 3

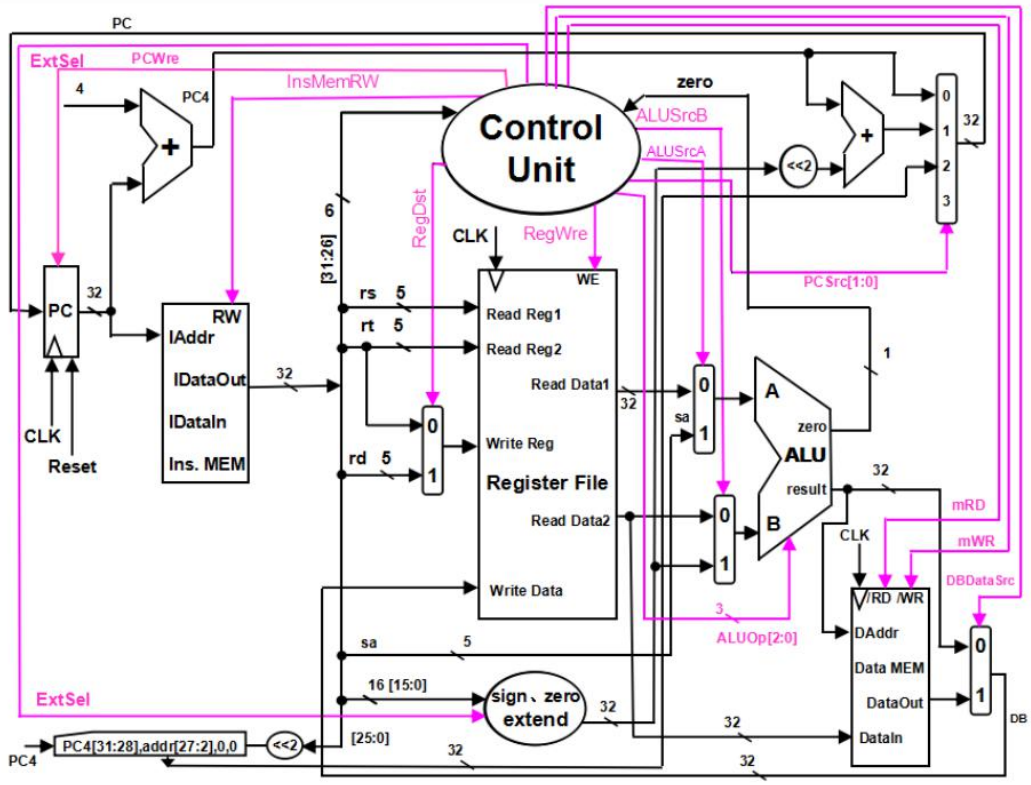


表2 确定需要的控制信号及其含义

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改	PC 更改
ALUSrcA	来自寄存器堆 data1 输出	来自移位数 sa
ALUSrcB	来自寄存器堆 data2 输出	来自扩展后的立即数
DBDataSrc	来自 ALU 运算结果的输出	来自数据存储器的输出
RegWre	寄存器组不需写使能	寄存器组写使能
InsMemRW	写指令存储器	读指令存储器
mRD	高阻态	读数据存储器
mWR	无	写数据存储器
RegDst	写寄存器组寄存器的地址来自 rt	写寄存器组寄存器的地址来自 rd
ExtSel	(zero-extend)immediate	(sign-extend)immediate
PCSrc[1..0]	00：PC<—PC+4 01：PC<—PC+4+(sign-extend)immediate 10：PC<—(PC+4)[31:28],addr[27:2],concat“00” 11：无	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，见后表	

表3 ALU功能表

ALUop[2：0]	对应操作
000	加
001	减
010	左移
011	或
100	与
101	小于置1
110	右移
111	异或

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：

实验过程：

1.撰写 Verilog 代码，代码分为几个模块，分别是:pcAdd、PC、InsMEM、InstructionCut、

ControlUnit、ALU、RegisterFile、DataMEM、SignZeroExtend、顶层文件 SingleCycleCPU 以及仿真代码模块 TestSingleCycleCpu;

2.通过 MARS 得到指令的机器码，将机器码写入 romData.txt 文件中，存入指令存储器中;

3.进行仿真，测试代码是否正确，各条指令是否能够正确运行;

4.综合、实现、设置好 I/O ports，生成 Bitstream 文件下载进开发板中测试数码管是否能正确显示 ALU 结果。

（I）撰写Verilog代码：

单周期CPU通过划分为如下几部分模块进行实现：

pcAdd、PC、InsMEM、InstructionCut、 ControlUnit、ALU、RegisterFile、DataMEM、SignZeroExtend.

(i)PC：根据控制信号PCWre，判断PC是否改变以及根据Reset信号判断是否重置。将时钟信号的上升沿和控制信号Reset作为变量，使得PC在上升沿的时候发生改变或被重置。

代码：

```
module PC(
      input CLK,              //时钟
      input Reset,            //重置
      input PCWre,            //是否接受新地址
      input [1:0] PCSrc,      //数据选择器输入
      input [31:0] nextPC,  //新指令地址
      output reg[31:0] curPC //当前指令的地址
   );
initial begin
   curPC <= 0;
end

always@(posedge CLK or negedge Reset)
begin
   if(!Reset) //重置 PC = 0
       begin
           curPC <= 0;
       end
```

```
    else
        begin
            if(PCWre) // PCWre == 1时，接受新地址
                begin
                    curPC <= nextPC;
                end
            else    // PCWre == 0，则停机（halt）
                begin
                    curPC <= curPC;
                end
        end
end
endmodule
```

(ii)pcAdd:根据控制信号PCSrc，计算获得下一个PC以及控制信号Reset重置。为了能够确保下一条PC能够正确得到，选择在时钟的下降沿以及控制信号Reset的下降沿时进行计算PC地址。

代码:

```
module pcAdd(
        input Reset,
        input CLK,              //时钟
        input [1:0] PCSrc,      //数据选择器输入
        input [31:0] immediate,  //跳转至新地址需要加的立即数
        input [25:0] addr,
        input [31:0] curPC,       //当前指令地址
        output reg[31:0] nextPC  //新指令地址
    );

    initial begin
        nextPC <= 0;
    end
```

```
    reg [31:0] pc;


    always@(negedge CLK or negedge Reset)
    begin
        if(!Reset) begin
            nextPC <= 0;
        end
        else begin
            pc <= curPC + 4;
            case(PCSrc)
                2'b00: nextPC <= curPC + 4;//正常
                2'b01: nextPC <= curPC + 4 + immediate * 4;//分支
                2'b10: nextPC <= {pc[31:28],addr,2'b00};//j型
                2'b11: nextPC <= nextPC; //停机
            endcase
        end
    end
endmodule
```

(iii)InstructionCut:将指令32位机器码进行分割，根据R、I、J型指令的指令结构，将指令分割，得到相对应的信息。

代码：

```
module InstructionCut(
        input [31:0] instruction,  //机器码
        output reg[5:0] op,        //下面是分割机器码得到的相应信息
        output reg[4:0] rs,
        output reg[4:0] rt,
        output reg[4:0] rd,
        output reg[4:0] sa,
        output reg[15:0] immediate,
```

```
        output reg[25:0] addr
    );

    initial begin
        op = 5'b00000;
        rs = 5'b00000;
        rt = 5'b00000;
        rd = 5'b00000;
    end

    always@(instruction)
    begin
        op = instruction[31:26];
        rs = instruction[25:21];
        rt = instruction[20:16];
        rd = instruction[15:11];
        sa = instruction[10:6];
        immediate = instruction[15:0];
        addr = instruction[25:0];
    end
endmodule
```

(iv)InsMEM:依据当前PC，读取指令寄存器中对应地址的指令，即当PC发生改变的时候，则进行指令的读取，根据相关的地址，输出指令寄存器中相对应的指令。

代码：

```
module InsMEM(
        input [31:0] IAddr,
        input InsMemRW,             //状态为'0'，写指令寄存器，否则为读指令寄存器
        output reg[31:0] IDataOut
    );
```

```
    reg [7:0] rom[128:0];  // 存储器定义用reg类型，存储器存储单元8位长度，共128个存储单元，可以存32条指令


    // 此处为绝对地址，注意斜杠方向
    initial
    begin
  $readmemb("C:\\Users\\czh\\Desktop\\CPU\\code\\cpu\\romData.txt",rom);
//文件路径
    end



    always@(IAddr or InsMemRW)
    begin
        //取指
        if(InsMemRW)
            begin
                IDataOut[7:0] = rom[IAddr + 3];
                IDataOut[15:8] = rom[IAddr + 2];
                IDataOut[23:16] = rom[IAddr + 1];
                IDataOut[31:24] = rom[IAddr];
            end
            //$display("iaddr: %d insmemrw: %d inst; %d",IAddr, InsMemRW, IDataOut);
    end


endmodule
```

(v)ControlUnit：控制单元，在这一部分将依据指令的操作码(op)以及标记符(ZERO)来确定该条指令对应的控制信号（操作码用来确定是什么指令，ZERO则用来判断是否跳转），从而达到控制各指令的目的，并输出相关的控制信号。

代码:

```
module ControlUnit(
```

```
        input zero,         //ALU运算结果是否为0，为0时候为1
        input [5:0] op,     //指令的操作码
        output reg PCWre,       //PC是否更改的信号量，为0时候不更改，否则可以更改
        output reg ExtSel,      //立即数扩展的信号量，为0时候为0扩展，否则为符号扩展
        output reg InsMemRW,    //指令寄存器的状态操作符，为0的时候写指令寄存器，否则为读指令寄存器
        output reg RegDst,      //写寄存器组寄存器的地址，为0的时候地址来自rt，为1的时候地址来自rd
        output reg RegWre,      //寄存器组写使能，为1的时候可写
        output reg ALUSrcA,     //控制ALU数据A的选择端的输入，为0的时候，来自寄存器堆data1输出，为1的时候来自移位数sa
        output reg ALUSrcB,     //控制ALU数据B的选择端的输入，为0的时候，来自寄存器堆data2输出，为1时候来自扩展过的立即数
        output reg [1:0]PCSrc,  //获取下一个pc的地址的数据选择器的选择端输入
        output reg [2:0]ALUOp,  //ALU 8种运算功能选择(000-111)
        output reg mRD,         //数据存储器读控制信号，为0读
        output reg mWR,         //数据存储器写控制信号，为0写
        output reg DBDataSrc    //数据保存的选择端，为0来自ALU运算结果的输出，为1来自数据寄存器（Data MEM）的输出
    );

    initial begin
        InsMemRW = 1;
        PCWre = 1;
        mRD = 0;
        mWR = 0;
        DBDataSrc = 0;
    end

    always@(op or zero)
```

```
begin
    PCWre = (op == 6'b111111) ? 0 : 1;   //halt
    InsMemRW = (op == 6'b111111) ? 0 : 1;   //halt
    mWR = (op == 6'b100110) ? 1 : 0;     //sw
    mRD = (op == 6'b100111) ? 1 : 0;     //lw
    DBDataSrc = (op == 6'b100111) ? 1 : 0;   //lw

    case(op)
       //addi
        6'b000001:
            begin
                ExtSel = 1;
                RegDst = 0;
                RegWre = 1;
                ALUSrcA = 0;
                ALUSrcB = 1;
                PCSrc = 2'b00;
                ALUOp = 3'b000;
            end
        //addiu
          6'b011111:
              begin
                      ExtSel = 1;
                      RegDst = 0;
                      RegWre = 1;
                      ALUSrcA = 0;
                      ALUSrcB = 1;
                      PCSrc = 2'b00;
                      ALUOp = 3'b000;
               end
```

```
        //ori
        6'b010000:
           begin
                ExtSel = 1;
                RegDst = 0;
                RegWre = 1;
                ALUSrcA = 0;
                ALUSrcB = 1;
                PCSrc = 2'b00;
                ALUOp = 3'b011;
            end
            //xori
            6'b011010:
               begin
                    ExtSel = 1;
                    RegDst = 0;
                    RegWre = 1;
                    ALUSrcA = 0;
                    ALUSrcB = 1;
                    PCSrc = 2'b00;
                   ALUOp = 3'b111;
                 end
//andi
   6'b010110:
          begin
                    ExtSel = 1;
                    RegDst = 0;
                    RegWre = 1;
                    ALUSrcA = 0;
                    ALUSrcB = 1;
```

```
            PCSrc = 2'b00;
            ALUOp = 3'b100;
      end
    //add
    6'b000000:
        begin
            ExtSel = 0;
            RegDst = 1;
            RegWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 0;
            PCSrc = 2'b00;
            ALUOp = 3'b000;
        end
   //addu
       6'b001001:
       begin
          ExtSel = 0;
          RegDst = 1;
          RegWre = 1;
          ALUSrcA = 0;
          ALUSrcB = 0;
          PCSrc = 2'b00;
          ALUOp = 3'b000;
       end
    //sub
    6'b000010:
        begin
            ExtSel = 1;
            RegDst = 1;
```

```
            RegWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 0;
            PCSrc = 2'b00;
            ALUOp = 3'b001;
        end
    //subu
        6'b000111:
        begin
          ExtSel = 1;
          RegDst = 1;
          RegWre = 1;
          ALUSrcA = 0;
          ALUSrcB = 0;
          PCSrc = 2'b00;
          ALUOp = 3'b001;
end
  //and
  6'b010001:
      begin
          ExtSel = 0;
          RegDst = 1;
          RegWre = 1;
          ALUSrcA = 0;
          ALUSrcB = 0;
          PCSrc = 2'b00;
          ALUOp = 3'b100;
      end
    //xor
    6'b010011:
```

```
        begin
            ExtSel = 0;
            RegDst = 1;
            RegWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 0;
            PCSrc = 2'b00;
            ALUOp = 3'b111;
      end
//sllv
 6'b011001:
 begin
              ExtSel = 0;
              RegDst = 1;
              RegWre = 1;
              ALUSrcA = 0;
              ALUSrcB = 0;
              PCSrc = 2'b00;
              ALUOp = 3'b010;
 end
 //or
 6'b010010:
    begin
        ExtSel = 0;
        RegDst = 1;
        RegWre = 1;
        ALUSrcA = 0;
        ALUSrcB = 0;
        PCSrc = 2'b00;
        ALUOp = 3'b011;
```

```
        end
      //srlv
          6'b000011:
             begin
                      ExtSel = 0;
                      RegDst = 1;
                      RegWre = 1;
                      ALUSrcA = 0;
                      ALUSrcB = 0;
                      PCSrc = 2'b00;
                      ALUOp = 3'b110;
               end
      //sll
      6'b011000:
           begin
           ExtSel = 0;
           RegDst = 1;
           RegWre = 1;
           ALUSrcA = 1;
           ALUSrcB = 0;
           PCSrc = 2'b00;
           ALUOp = 3'b010;
end
 //srl
 6'b011110:
     begin
           ExtSel = 0;
           RegDst = 1;
           RegWre = 1;
           ALUSrcA = 1;
```

```
        ALUSrcB = 0;
        PCSrc = 2'b00;
        ALUOp = 3'b110;
    end
//bne
6'b110001:
    begin
        ExtSel = 1;
        RegDst = 0;
        RegWre = 0;
        ALUSrcA = 0;
        ALUSrcB = 0;
        PCSrc = zero ? 2'b00 : 2'b01;
        ALUOp = 3'b001;
    end
    //sltiu
          6'b011101:
               begin
                    ExtSel = 1;
                    RegDst = 0;
                    RegWre = 1;
                    ALUSrcA = 0;
                    ALUSrcB = 1;
                    PCSrc = 2'b00;
                    ALUOp = 3'b101;
                 end
//slti
6'b011011:
    begin
        ExtSel = 1;
```

```
            RegDst = 0;
            RegWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 1;
            PCSrc = 2'b00;
            ALUOp = 3'b101;
        end
         //slt
                  6'b010101:
                      begin
                          ExtSel = 1;
                          RegDst = 0;
                          RegWre = 1;
                          ALUSrcA = 0;
                          ALUSrcB = 0;
                          PCSrc = 2'b00;
                          ALUOp = 3'b101;
                      end
    //beq
    6'b110000:
        begin
            ExtSel = 1;
            RegDst = 0;
            RegWre = 0;
            ALUSrcA = 0;
            ALUSrcB = 0;
            PCSrc = zero ? 2'b01 : 2'b00;
            ALUOp = 3'b001;
        end
    //sw
```

```
6'b100110:
    begin
        ExtSel = 1;
        RegDst = 0;
        RegWre = 0;
        ALUSrcA = 0;
        ALUSrcB = 1;
        PCSrc = 2'b00;
        ALUOp = 3'b000;
    end
//lw
6'b100111:
    begin
        ExtSel = 1;
        RegDst = 0;
        RegWre = 1;
        ALUSrcA = 0;
        ALUSrcB = 1;
        PCSrc = 2'b00;
        ALUOp = 3'b000;
    end
//j
6'b111000:
    begin
        ExtSel = 0;
        RegDst = 0;
        RegWre = 0;
        ALUSrcA = 0;
        ALUSrcB = 0;
        PCSrc = 2'b10;
```

```
                ALUOp = 3'b000;
            end
        //halt
        6'b111111:
            begin
                ExtSel = 0;
                RegDst = 0;
                RegWre = 0;
                ALUSrcA = 0;
                ALUSrcB = 0;
                PCSrc = 2'b11;
                ALUOp = 3'b000;
            end
    endcase
end
endmodule
```

(vi)ALU:算术逻辑单元，对两个输入根据前面所列的ALU功能表进行相对应的运算。

代码：

```
module ALU(
    input ALUSrcA,
    input ALUSrcB,
    input [31:0] ReadData1,
    input [31:0] ReadData2,
    input [4:0] sa,
    input [31:0] extend,
    input [2:0] ALUOp,
    output reg zero,
    output reg[31:0] result   //结果
);
```

```
    reg [31:0] A;  //对应的操作数
    reg [31:0] B;  //对应的操作数

    always@(ReadData1 or ReadData2 or ALUSrcA or ALUSrcB or ALUOp)
    begin
        //根据控制信号来确定从哪里输入
        A = (ALUSrcA == 0) ? ReadData1 : sa;
        B = (ALUSrcB == 0) ? ReadData2 : extend;
        case(ALUOp)
            3'b000: result = A + B;   //加
            3'b001: result = A - B;   //减
            3'b010: result = B << A;  //左移
            3'b011: result = A | B;  //或
            3'b100: result = A & B;  //与
            3'b101: result = (A < B) ? 1 : 0;  //小于置1
            3'b110: result = B >> A;  //右移
            3'b111: result = A ^ B;   //异或
        endcase
        zero = (result == 0) ? 1 : 0;
    end
endmodule
```

(vii)DataMEM：数据存储器，通过控制信号，对数据寄存器进行读或者写操作，并且此模块额外合并了输出DB的数据选择器，此模块同时输出写回寄存器组的数据DB。

代码：

```
module DataMEM(
        input mRD, //数据存储器读控制信号，为0读
        input mWR, //数据存储器写控制信号，为0写
        input CLK, //时钟
        input DBDataSrc,
        input [31:0] DAddr, //数据存储器地址输入端口
```

```
    input [31:0] DataIn, //数据存储器数据输入端口
    output reg[31:0] DataOut, //数据存储器数据输出端口
    output reg[31:0] DB
);

initial begin
    DB <= 16'b0;
end

 reg [7:0] ram [0:31];

always@(mRD or DAddr or DBDataSrc)
begin
    //读
    DataOut[7:0] = mRD ? ram[DAddr + 3] : 8'bz; // z代表高阻态
    DataOut[15:8] = mRD ? ram[DAddr + 2] : 8'bz;
    DataOut[23:16] = mRD ? ram[DAddr + 1] : 8'bz;
    DataOut[31:24] = mRD ? ram[DAddr] : 8'bz;

    DB = (DBDataSrc == 0) ? DAddr : DataOut;
end

always@(negedge CLK)
begin
    //写
    if(mWR)
        begin
            ram[DAddr] = DataIn[31:24];
            ram[DAddr + 1] = DataIn[23:16];
            ram[DAddr + 2] = DataIn[15:8];
```

```
            ram[DAddr + 3] = DataIn[7:0];
        end
    //$display("mwr: %d $12 %d %d %d %d", mWR, ram[12], ram[13], ram[14], ram[15]);
end
endmodule
```

(viii)RegisterFile：寄存器组，通过控制单元输出的控制信号，进行相对应的读/写操作。

代码：

```
module RegisterFile(
        input CLK,                  //时钟
        input [4:0] ReadReg1,    //rs寄存器地址输入端口
        input [4:0] ReadReg2,    //rt寄存器地址输入端口
        input [31:0] WriteData,     //写入寄存器的数据输入端口
        input [4:0] WriteReg,       //将数据写入的寄存器端口，其地址来源rt或rd字段
        input RegWre,               //WE，写使能信号，为1时，在时钟边沿触发写入
        output reg[31:0] ReadData1,  //rs寄存器数据输出端口
        output reg[31:0] ReadData2   //rt寄存器数据输出端口
    );

    initial begin
        ReadData1 <= 0;
        ReadData2 <= 0;
    end



    reg [31:0] regFile[0:31];
    integer i;
    initial begin
        for (i = 0; i < 32; i = i+ 1) regFile[i] <= 0;
    end
```

```
        always@(ReadReg1 or ReadReg2)
        begin
            ReadData1 = regFile[ReadReg1];
            ReadData2 = regFile[ReadReg2];
            //$display("regfile %d %d\n", ReadReg1, ReadReg2);
        end

        always@(negedge CLK)
        begin
            //$0恒为0，所以写入寄存器的地址不能为0
            if(RegWre && WriteReg)
                begin
                    regFile[WriteReg] <= WriteData;
                end
        end
    endmodule
```

(ix)SignZeroExtend：根据指令相关的控制信号ExtSel，看是否需要对立即数进行符号扩展。

代码：

```
module SignZeroExtend(
        input wire [15:0] immediate,    //立即数
        input ExtSel,                   //最高位（符号位）
        output [31:0] extendImmediate   //扩展后立即数
    );

    always@(extendImmediate)
    begin
        $display("%d", extendImmediate[31]);
    end
```

```
    assign extendImmediate[15:0] = immediate;
    assign extendImmediate[31:16] = ExtSel ? (immediate[15] ? 16'hffff : 16'h0000) :
16'h0000; //最高位是1则补1，否则补0
endmodule
```

（x）顶层模块：SingleCycleCPU. 在顶层模块中将各个已实现的底层模块进行实例化，并将各个模块连接起来。

代码：

```
module SingleCycleCPU(
        input CLK,
        input Reset,
        output [31:0] curPC,
        output [31:0] nextPC,
        output [31:0] instruction,
        output [5:0] op,
        output [4:0] rs,
        output [4:0] rt,
        output [4:0] rd,
        output [31:0] DB,
        output [31:0] A,
        output [31:0] B,
        output [31:0] result,
        output [1:0] PCSrc,
        output zero,
        output PCWre,       //PC是否更改的信号量，为0时候不更改，否则可以更改
        output ExtSel,      //立即数扩展的信号量，为0时候为0扩展，否则为符号扩展
        output InsMemRW,    //指令寄存器的状态操作符，为0的时候写指令寄存器，否则为读指令寄
存器
        output RegDst,      //写寄存器组寄存器的地址，为0的时候地址来自rt，为1的时候地址来
自rd
```

```
        output RegWre,      //寄存器组写使能，为1的时候可写
        output ALUSrcA,     //控制ALU数据A的选择端的输入，为0的时候，来自寄存器堆data1输出，为1的时候来自移位数sa
        output ALUSrcB,     //控制ALU数据B的选择端的输入，为0的时候，来自寄存器堆data2输出，为1时候来自扩展过的立即数
        output [2:0]ALUOp,  //ALU 8种运算功能选择(000-111)
        output mRD,         //数据存储器读控制信号，为0读
        output mWR,         //数据存储器写控制信号，为0写
        output DBDataSrc    //数据保存的选择端，为0来自ALU运算结果的输出，为1来自数据寄存器（Data MEM）的输出
    );

    wire [31:0] extend;
    wire [31:0] DataOut;
    wire[4:0] sa;
    wire[15:0] immediate;
    wire[25:0] addr;

    pcAdd pcAdd(.Reset(Reset),
                .CLK(CLK),
                .PCSrc(PCSrc),
                .immediate(extend),
                .addr(addr),
                .curPC(curPC),
                .nextPC(nextPC));

    PC pc(.CLK(CLK),
          .Reset(Reset),
          .PCWre(PCWre),
          .PCSrc(PCSrc),
```

```
      .nextPC(nextPC),
      .curPC(curPC));

InsMEM InsMEM(.IAddr(curPC),
            .InsMemRW(InsMemRW),
            .IDataOut(instruction));

InstructionCut InstructionCut(.instruction(instruction),
                              .op(op),
                              .rs(rs),
                              .rt(rt),
                              .rd(rd),
                              .sa(sa),
                              .immediate(immediate),
                              .addr(addr));

ControlUnit ControlUnit(.zero(zero),
                        .op(op),
                        .PCWre(PCWre),
                        .ExtSel(ExtSel),
                        .InsMemRW(InsMemRW),
                        .RegDst(RegDst),
                        .RegWre(RegWre),
                        .ALUSrcA(ALUSrcA),
                        .ALUSrcB(ALUSrcB),
                        .PCSrc(PCSrc),
                        .ALUOp(ALUOp),
                        .mRD(mRD),
                        .mWR(mWR),
                        .DBDataSrc(DBDataSrc));
```

```
RegisterFile RegisterFile(.CLK(CLK),
                          .ReadReg1(rs),
                          .ReadReg2(rt),
                          .WriteData(DB),
                          .WriteReg(RegDst ? rd : rt),
                          .RegWre(RegWre),
                          .ReadData1(A),
                          .ReadData2(B));


ALU alu(.ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB),
        .ReadData1(A),
        .ReadData2(B),
        .sa(sa),
        .extend(extend),
        .ALUOp(ALUOp),
        .zero(zero),
        .result(result));


DataMEM DataMEM(.mRD(mRD),
                .mWR(mWR),
                .CLK(CLK),
                .DBDataSrc(DBDataSrc),
                .DAddr(result),
                .DataIn(B),
                .DataOut(DataOut),
                .DB(DB));


SignZeroExtend SignZeroExtend(.immediate(immediate),
```

```
                                .ExtSel(ExtSel),
                                .extendImmediate(extend));


Endmodule
```

（II）将机器码写入romData.txt文件：

```
01111100
00000001
00000000
00001000
01000000
00000010
00000000
00000010
00000000
01000001
00011000
00000000
00001000
01100010
00101000
00000000
01000100
10100010
00100000
00000000
01001000
10000010
01000000
00000000
01100000
```

00001000

01000000

01000000

11000101

00000001

11111111

11111110

01101100

01000110

00000000

00001000

01101100

11000111

00000000

00000000

00000100

11100111

00000000

00001000

11000000

11100001

11111111

11111110

10011000

00100010

00000000

00000100

10011100

00101001

00000000

```
00000100
01001100
00100010
00011000
00000000
01100100
11000001
00010000
00000000
00001100
11000001
00011000
00000000
11100000
00000000
00000000
00010011
00000100
00001010
00000000
00001010
00011100
00000010
00011000
00000000
01111000
00001000
01000000
01000000
00100100
```

```
00000010

00011000

00000000

01010100

01101000

00001000

00000000

01110101

00000001

00000000

00001100

01101000

01000001

00000000

00001100

01011000

01000001

00000000

00000011

11111100

00000000

00000000

00000000
```

（III）仿真：

首先先写一个简单的仿真文件：

```
module TestSingleCycleCpu();

    // Inputs

     reg CLK;

     reg Reset;
```

```
    // Outputs
    wire [1:0] PCSrc;
    wire [5:0] op;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [31:0] DB;
    wire [31:0] result;
    wire [31:0] curPC;
   wire [31:0] nextPC;
   wire [31:0] instruction;
   wire [31:0] A;
   wire [31:0] B;
   wire zero;
   wire PCWre;       //PC是否更改的信号量，为0时候不更改，否则可以更改
   wire ExtSel;      //立即数扩展的信号量，为0时候为0扩展，否则为符号扩展
   wire InsMemRW;    //指令寄存器的状态操作符，为0的时候写指令寄存器，否则为读指令寄存器
   wire RegDst;      //写寄存器组寄存器的地址，为0的时候地址来自rt，为1的时候地址来自rd
   wire RegWre;      //寄存器组写使能，为1的时候可写
   wire ALUSrcA;     //控制ALU数据A的选择端的输入，为0的时候，来自寄存器堆data1输出，为1
的时候来自移位数sa
   wire ALUSrcB;     //控制ALU数据B的选择端的输入，为0的时候，来自寄存器堆data2输出，为1
时候来自扩展过的立即数
   wire [2:0]ALUOp;  //ALU 8种运算功能选择(000-111)
   wire mRD;         //数据存储器读控制信号，为0读
   wire mWR;         //数据存储器写控制信号，为0写
   wire DBDataSrc;    //数据保存的选择端，为0来自ALU运算结果的输出，为1来自数据寄存器（Data
MEM）的输出
    // Instantiate the Unit Under Test (UUT)
    SingleCycleCPU uut (
```

```
        .CLK(CLK),
        .Reset(Reset),
        .curPC(curPC),
        .nextPC(nextPC),
        .instruction(instruction),
        .op(op),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .DB(DB),
        .A(A),
        .B(B),
        .result(result),
        .PCSrc(PCSrc),
        .zero(zero),
        .PCWre(PCWre),
        .ExtSel(ExtSel),
        .InsMemRW(InsMemRW),
        .RegDst(RegDst),
        .RegWre(RegWre),
        .ALUSrcA(ALUSrcA),
        .ALUSrcB(ALUSrcB),
        .ALUOp(ALUOp),
        .mRD(mRD),
        .mWR(mWR),
        .DBDataSrc(DBDataSrc)
    );

    initial begin
        // Initialize Inputs
```

```
        CLK = 1;
        Reset = 0;

        CLK = !CLK;  // 下降沿，使PC先清零
        Reset = 1;  // 清除保持信号
        forever #5
        begin // 产生时钟信号，周期为50s
            CLK = !CLK;
        end
    end
endmodule
```

（IV）在FPGA板上实现：

首先新建一个project，然后将前面所列的文件复制过来，然后再添加如下文件：

（i）消抖文件：

```
module Debounce(
        input clk,
        input key,
        output out
    );

    reg delay1,delay2,delay3;
    assign out = delay1&delay2&delay3;
    always@(posedge clk)//CLK 100M
    begin
        delay1 <= key;
        delay2 <= delay1;
        delay3 <= delay2;
    end
endmodule
```

（ii）实例化CPU模块：

```
module SingleCycleCPUBasys(
        input CLK,
        input Reset,
        output [31:0] curPC,
        output [31:0] nextPC,
        output [31:0] instruction,
        output [5:0] op,
        output [4:0] rs,
        output [4:0] rt,
        output [4:0] rd,
        output [31:0] DB,
        output [31:0] A,
        output [31:0] B,
        output [31:0] result,
        output [1:0] PCSrc
    );

    wire zero;
    wire PCWre;       //PC是否更改的信号量，为0时候不更改，否则可以更改
    wire ExtSel;      //立即数扩展的信号量，为0时候为0扩展，否则为符号扩展
    wire InsMemRW;    //指令寄存器的状态操作符，为0的时候写指令寄存器，否则为读指令寄存器
    wire RegDst;      //写寄存器组寄存器的地址，为0的时候地址来自rt，为1的时候地址来自rd
    wire RegWre;      //寄存器组写使能，为1的时候可写
    wire ALUSrcA;     //控制ALU数据A的选择端的输入，为0的时候，来自寄存器堆data1输出，为1
的时候来自移位数sa
    wire ALUSrcB;     //控制ALU数据B的选择端的输入，为0的时候，来自寄存器堆data2输出，为1
时候来自扩展过的立即数
   // wire [1:0]PCSrc;  //获取下一个pc的地址的数据选择器的选择端输入
    wire [2:0]ALUOp;  //ALU 8种运算功能选择(000-111)
    wire mRD;         //数据存储器读控制信号，为0读
```

```
    wire mWR;          //数据存储器写控制信号，为0写
    wire DBDataSrc;    //数据保存的选择端，为0来自ALU运算结果的输出，为1来自数据寄存器（Data
MEM）的输出
    wire [31:0] extend;
    wire [31:0] DataOut;
    wire[4:0] sa;
    wire[15:0] immediate;
    wire[25:0] addr;

    pcAdd pcAdd(.Reset(Reset),
                .CLK(CLK),
                .PCSrc(PCSrc),
                .immediate(extend),
                .addr(addr),
                .curPC(curPC),
                .nextPC(nextPC));

    PC pc(.CLK(CLK),
          .Reset(Reset),
          .PCWre(PCWre),
          .PCSrc(PCSrc),
          .nextPC(nextPC),
          .curPC(curPC));
    InsMEM InsMEM(.IAddr(curPC),
                .InsMemRW(InsMemRW),
                .IDataOut(instruction));

    InstructionCut InstructionCut(.instruction(instruction),
                                  .op(op),
                                  .rs(rs),
```

```
                .rt(rt),
                .rd(rd),
                .sa(sa),
                .immediate(immediate),
                .addr(addr));

ControlUnit ControlUnit(.zero(zero),
                .op(op),
                .PCWre(PCWre),
                .ExtSel(ExtSel),
                .InsMemRW(InsMemRW),
                .RegDst(RegDst),
                .RegWre(RegWre),
                .ALUSrcA(ALUSrcA),
                .ALUSrcB(ALUSrcB),
                .PCSrc(PCSrc),
                .ALUOp(ALUOp),
                .mRD(mRD),
                .mWR(mWR),
                .DBDataSrc(DBDataSrc));

RegisterFile RegisterFile(.CLK(CLK),
                .ReadReg1(rs),
                .ReadReg2(rt),
                .WriteData(DB),
                .WriteReg(RegDst ? rd : rt),
                .RegWre(RegWre),
                .ReadData1(A),
                .ReadData2(B));
```

```
    ALU alu(.ALUSrcA(ALUSrcA),
            .ALUSrcB(ALUSrcB),
            .ReadData1(A),
            .ReadData2(B),
            .sa(sa),
            .extend(extend),
            .ALUOp(ALUOp),
            .zero(zero),
            .result(result));

    DataMEM DataMEM(.mRD(mRD),
                    .mWR(mWR),
                    .CLK(CLK),
                    .DBDataSrc(DBDataSrc),
                    .DAddr(result),
                    .DataIn(B),
                    .DataOut(DataOut),
                    .DB(DB));

    SignZeroExtend SignZeroExtend(.immediate(immediate),
                                  .ExtSel(ExtSel),
                                  .extendImmediate(extend));

endmodule
```

(iii)仿真文件：

```
module showCPU(
        input Reset,
        input clock_100Mhz, // 100 Mhz clock source on Basys 3 FPGA
        input S0, // control
        input S1, // control
```

```
    input click,  //
    output reg [3:0] Anode_Activate, // anode signals of the 7-segment LED display
    output reg [6:0] LED_out// cathode patterns of the 7-segment LED display
);

reg [3:0] LED_BCD;
reg [26:0] refresh_counter; // the first 19-bit for creating 190Hz refresh rate
// the other 2-bit for creating 4 LED-activating signals
wire [1:0] LED_activating_counter;
wire [31:0] curPC;
wire [31:0] nextPC;
wire [31:0] instruction;
wire [5:0] op;
wire [4:0] rs;
wire [4:0] rt;
wire [4:0] rd;
wire [31:0] DB;
wire [31:0] A;
wire [31:0] B;
wire [31:0] result;
wire [1:0] PCSrc;
wire CpuCLk;

Debounce Debounce(.clk(clock_100Mhz), .key(click), .out(CpuCLK));

SingleCycleCPUBasys SingleCycleCPUBasys(.CLK(CpuCLK),
                                        .Reset(Reset),
                                        .curPC(curPC),
                                        .nextPC(nextPC),
                                        .instruction(instruction),
```

```
                                    .op(op),
                                    .rs(rs),
                                    .rt(rt),
                                    .rd(rd),
                                    .DB(DB),
                                    .A(A),
                                    .B(B),
                                    .result(result),
                                    .PCSrc(PCSrc));

 //扫描频率
 always @(posedge clock_100Mhz)
 begin
     refresh_counter <= refresh_counter + 1;
 end
 assign LED_activating_counter = refresh_counter[20:19];

//显示板块
always @(*)
 begin
     case(LED_activating_counter)
         2'b00: begin
             Anode_Activate = 4'b0111;
             if(!S0 && !S1) begin
                 LED_BCD = curPC[7:4];
             end
             else if(!S0 && S1) begin
                 LED_BCD ={3'b000, rs[4]};
             end
             else if(S0 && S1) begin
```

```
            LED_BCD ={3'b000, rt[4]};
        end
        else begin
            LED_BCD = result[7:4];
        end
    end
    2'b01: begin
        Anode_Activate = 4'b1011;
        if(!S0 && !S1) begin
           LED_BCD = curPC[3:0];
        end
        else if(!S0 && S1) begin
            LED_BCD = rs[3:0];
        end
        else if(S0 && !S1) begin
            LED_BCD = rt[3:0];
        end
        else begin
            LED_BCD = result[3:0];
        end
    end
    2'b10: begin
        Anode_Activate = 4'b1101;
        if(!S0 && !S1) begin
           LED_BCD = nextPC[7:4];
        end
        else if(!S0 && S1) begin
            LED_BCD = A[7:4];
        end
        else if(S0 && !S1) begin
```

```
                LED_BCD = B[7:4];
            end
            else begin
                LED_BCD = DB[7:4];
            end
        end
        2'b11: begin
            Anode_Activate = 4'b1110;
            if(!S0 && !S1) begin
               LED_BCD = nextPC[3:0];
            end
            else if(!S0 && S1) begin
                LED_BCD = A[3:0];
            end
            else if(S0 && !S1) begin
                LED_BCD = B[3:0];
            end
            else begin
                LED_BCD = DB[3:0];
            end
        end
    endcase
end
 // Cathode patterns of the 7-segment LED display
always @(*)
begin
    case(LED_BCD)
        4'b0000: LED_out = 7'b0000001; // "0"
        4'b0001: LED_out = 7'b1001111; // "1"
        4'b0010: LED_out = 7'b0010010; // "2"
```

```
            4'b0011: LED_out = 7'b0000110; // "3"
            4'b0100: LED_out = 7'b1001100; // "4"
            4'b0101: LED_out = 7'b0100100; // "5"
            4'b0110: LED_out = 7'b0100000; // "6"
            4'b0111: LED_out = 7'b0001111; // "7"
            4'b1000: LED_out = 7'b0000000; // "8"
            4'b1001: LED_out = 7'b0000100; // "9"
            4'b1010: LED_out = 7'b0001000;  //A
            4'b1011: LED_out = 7'b1100000;  //B
            4'b1100: LED_out = 7'b0110001;  //C
            4'b1101: LED_out = 7'b1000010;  //D
            4'b1110: LED_out = 7'b0110000;  //E
            4'b1111: LED_out = 7'b0111000; //F
            default: LED_out = 7'b0000000;  //不亮
        endcase
    end

endmodule
```

（iv）约束文件：

```
set_property PACKAGE_PIN W5 [get_ports clock_100Mhz]
set_property IOSTANDARD LVCMOS33 [get_ports clock_100Mhz]
set_property PACKAGE_PIN R2 [get_ports S0]
set_property IOSTANDARD LVCMOS33 [get_ports S0]
set_property PACKAGE_PIN T1 [get_ports S1]
set_property IOSTANDARD LVCMOS33 [get_ports S1]
set_property PACKAGE_PIN V17 [get_ports Reset]
set_property IOSTANDARD LVCMOS33 [get_ports Reset]
set_property PACKAGE_PIN U18 [get_ports click]
set_property IOSTANDARD LVCMOS33 [get_ports click]
#seven-segment LED display
```

```
set_property PACKAGE_PIN W7 [get_ports {LED_out[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[6]}]
set_property PACKAGE_PIN W6 [get_ports {LED_out[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[5]}]
set_property PACKAGE_PIN U8 [get_ports {LED_out[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[4]}]
set_property PACKAGE_PIN V8 [get_ports {LED_out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[3]}]
set_property PACKAGE_PIN U5 [get_ports {LED_out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[2]}]
set_property PACKAGE_PIN V5 [get_ports {LED_out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[1]}]
set_property PACKAGE_PIN U7 [get_ports {LED_out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[0]}]

set_property PACKAGE_PIN U2 [get_ports {Anode_Activate[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[0]}]
set_property PACKAGE_PIN U4 [get_ports {Anode_Activate[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[1]}]
set_property PACKAGE_PIN V4 [get_ports {Anode_Activate[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[2]}]
set_property PACKAGE_PIN W4 [get_ports {Anode_Activate[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[3]}]
```

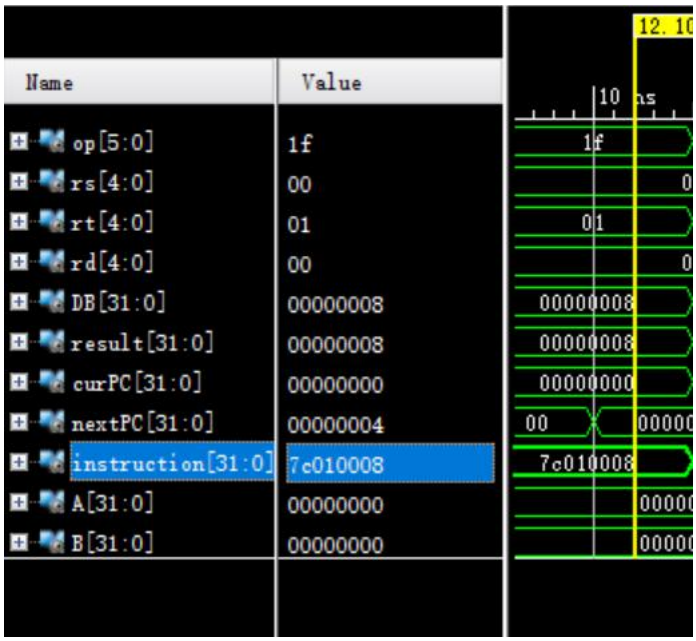
实验结果：

1.仿真结果：

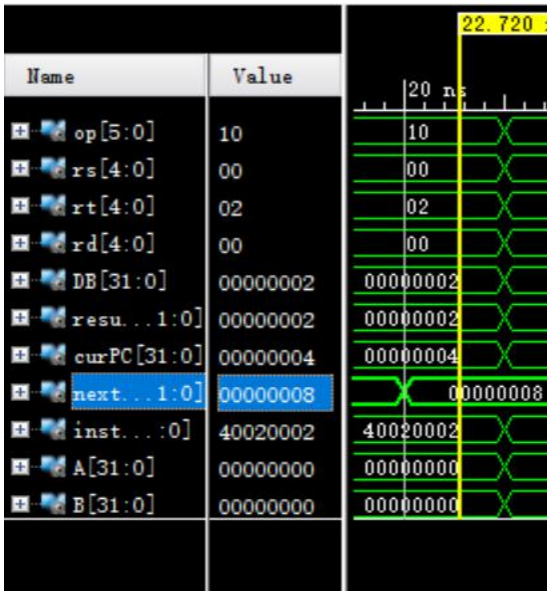
注：截图中所有显示数字均为16进制！！！resu...为结果result，next...为nextPC，inst...为指令的机器码，A，B为操作数。

对txt文件中每一条指令进行分析，如下：

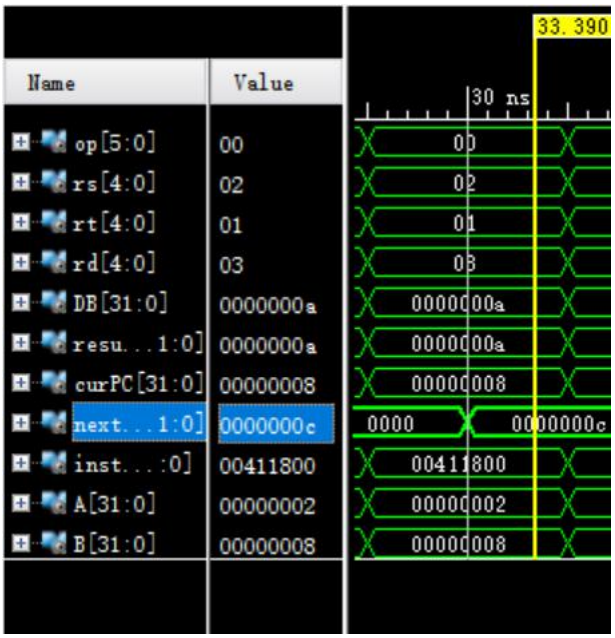
(1)addiu　$1,$0,8　7c010008（由$1=0+8=8=result可知结果正确）



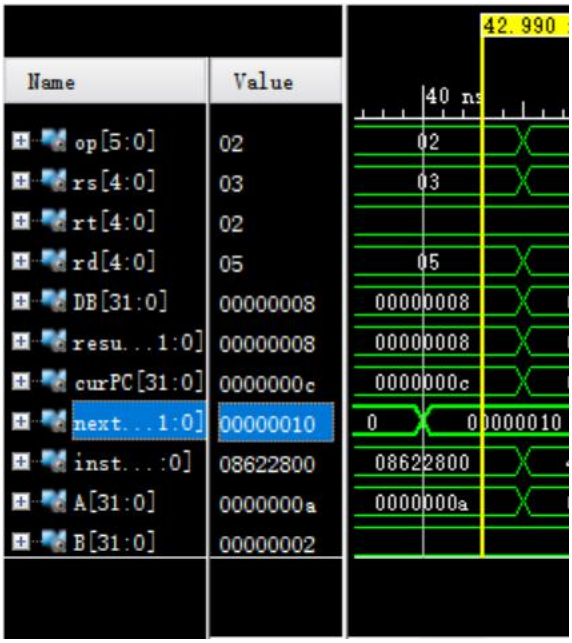
(2)ori　$2,$0,2　40020002（由$2=0|2=2=result可知结果正确）



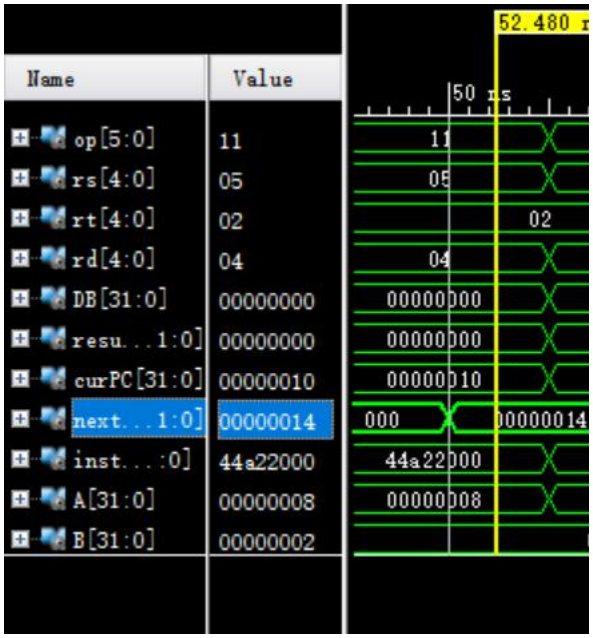
(3)add　$3,$2,$1　00411800（由$3=2+8=10=result可知结果正确）



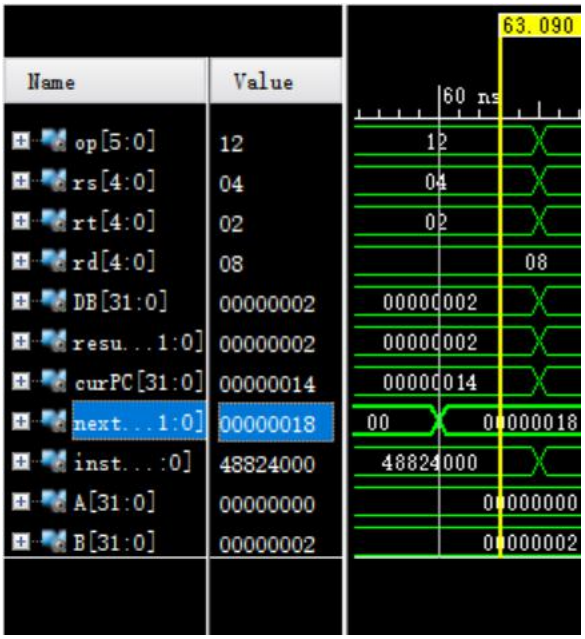
(4)sub　$5,$3,$2　08622800（由$5=10-2=8=result可知结果正确）



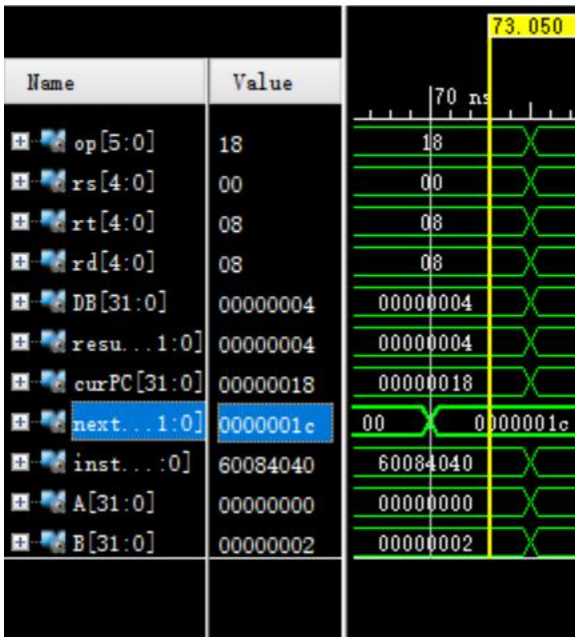
(5)and　$4,$5,$2　44a22000（由$4=8&2=0=result可知结果正确）



(6)or　$8,$4,$2　48824000（由$8=0|2=2=result可知结果正确）

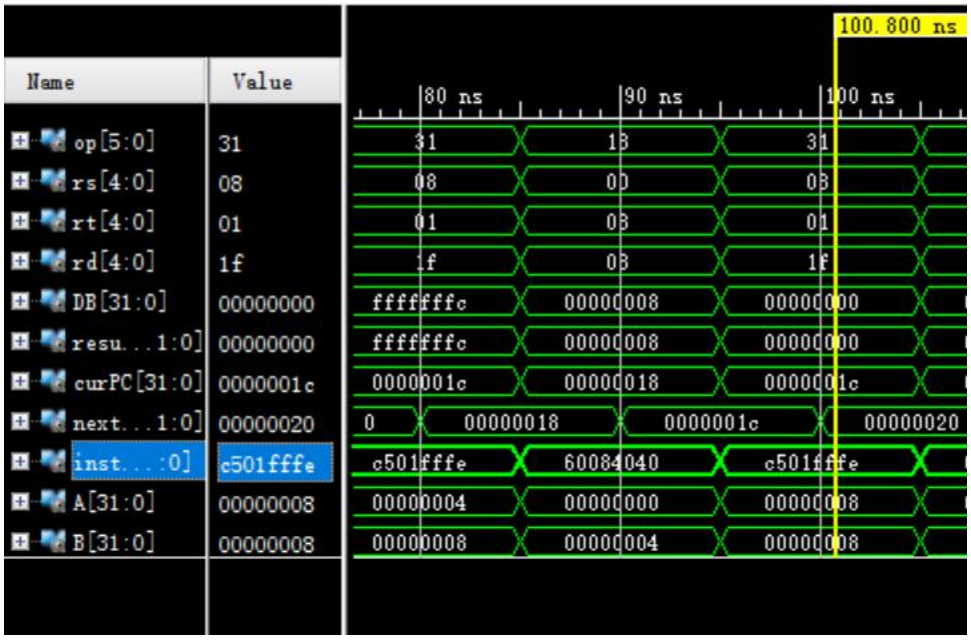


(7)sll　$8,$8,1　60084040　（由$8=2<<1=4=result可知结果正确）

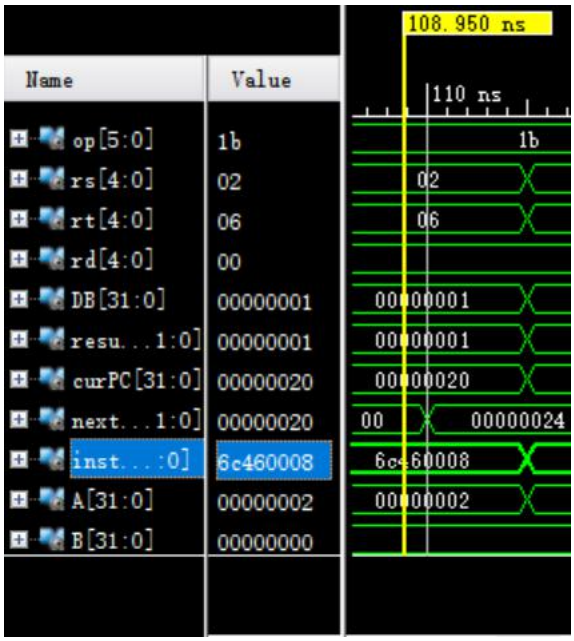


(8) bne $8,$1,-2 c501fffe

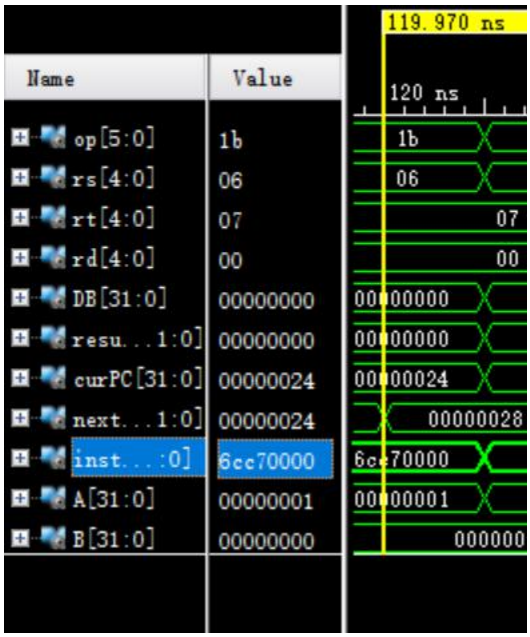
起初$8=4，$1=8不相等，跳转至上一条，进行左移1位后$8=8=$1,相等，于是继续执行。



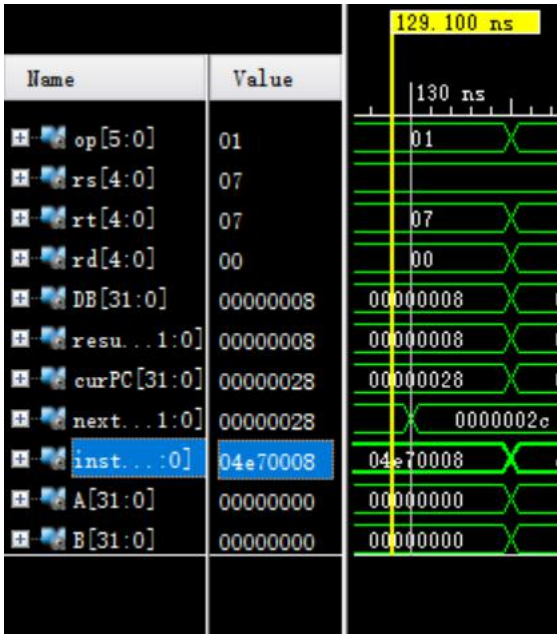
(9)slti $6,$2,4 6c460008 （$2=2<4置1，由result=1可知结果正确）



(10)slti　$7,$6,0　6cc70000　（$6=1>0置0，由result=0知结果正确）

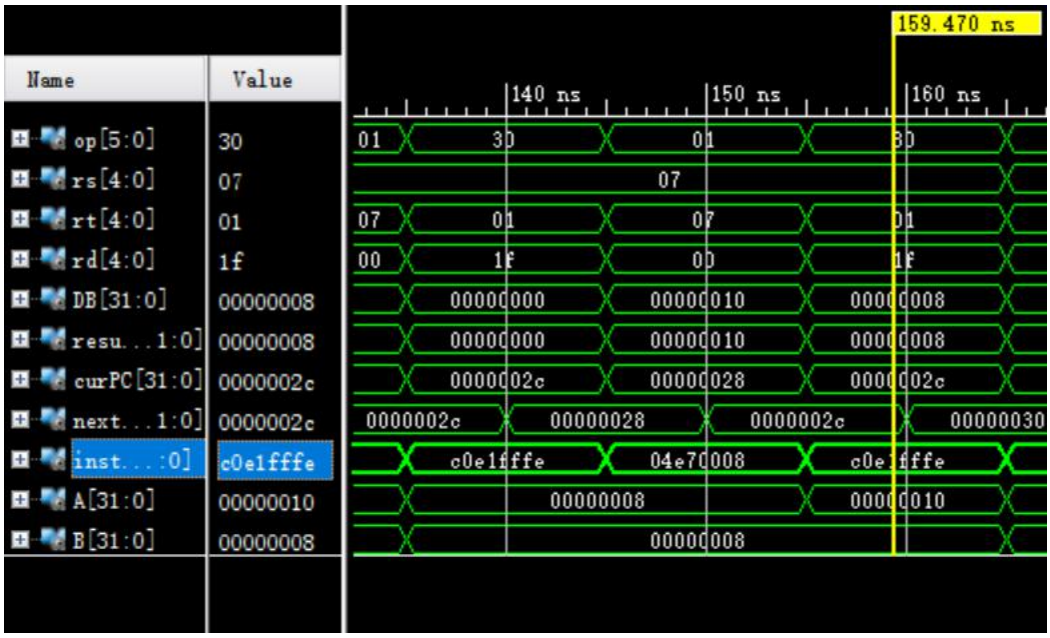


(11)addi　$7,$7,8　04e70008　（$7=0+8=8=result可知结果正确）

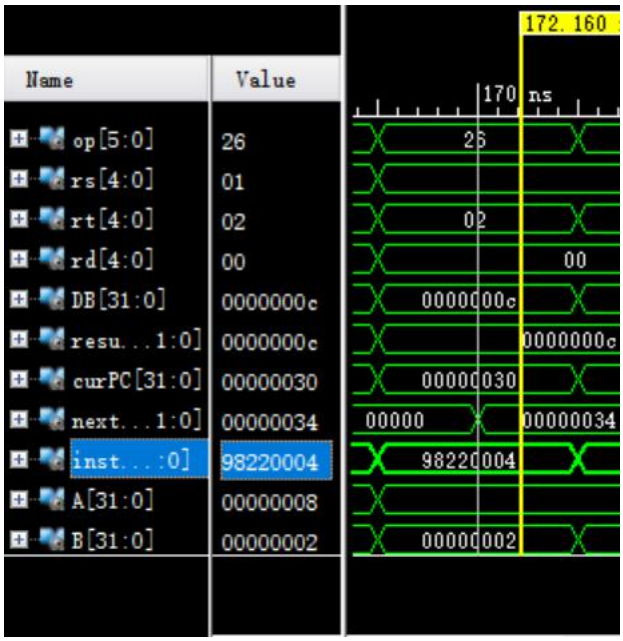


(12)beq　$7,$1,-2　c0e1fffe

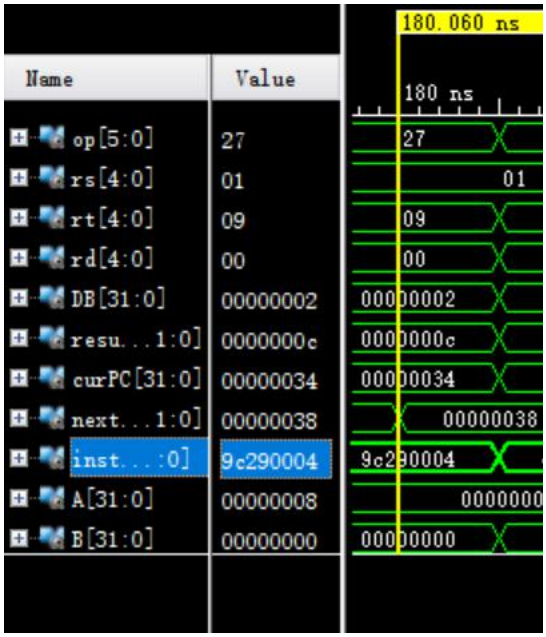
$7=$1=8,跳转至上一条，然后$7=16,$1=8,二者不相等，故继续执行



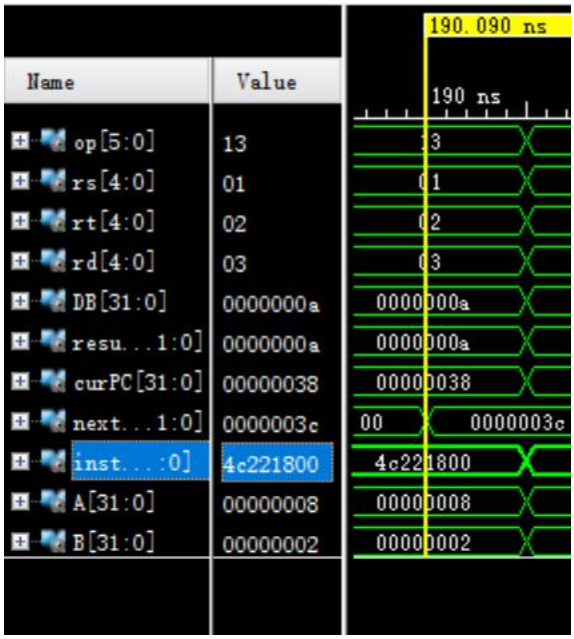
(13)sw　$2,4($1)　98220004（$1=8,8+4=12,故将$2=2存在数据寄存器中地址为12的地方）



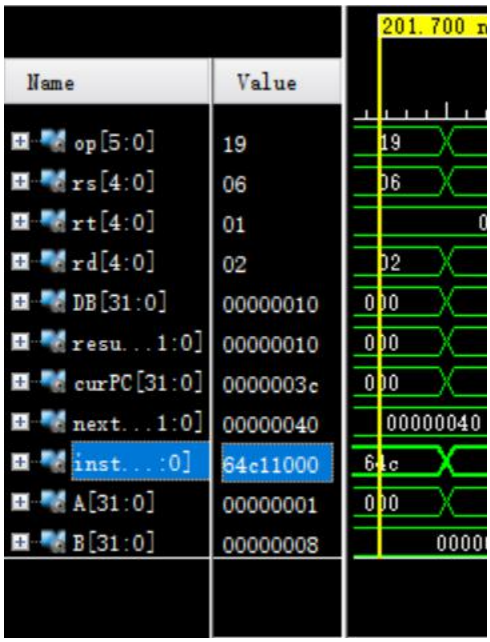
(14)lw　$9,4($1)　9c290004　（将数据寄存器中地址为 12 开始的内容存在$9 的地方）



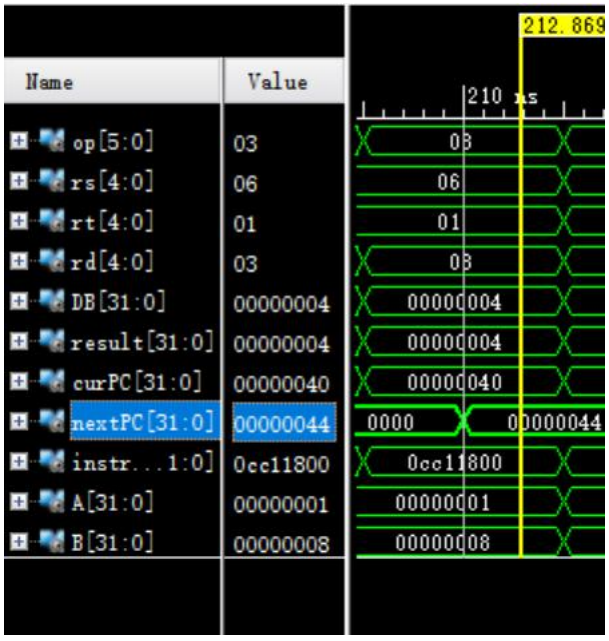
(15)xor　$3,$1,$2　4c221800（由$1=8,$2=2,$3=8^2=10=result故结果正确）



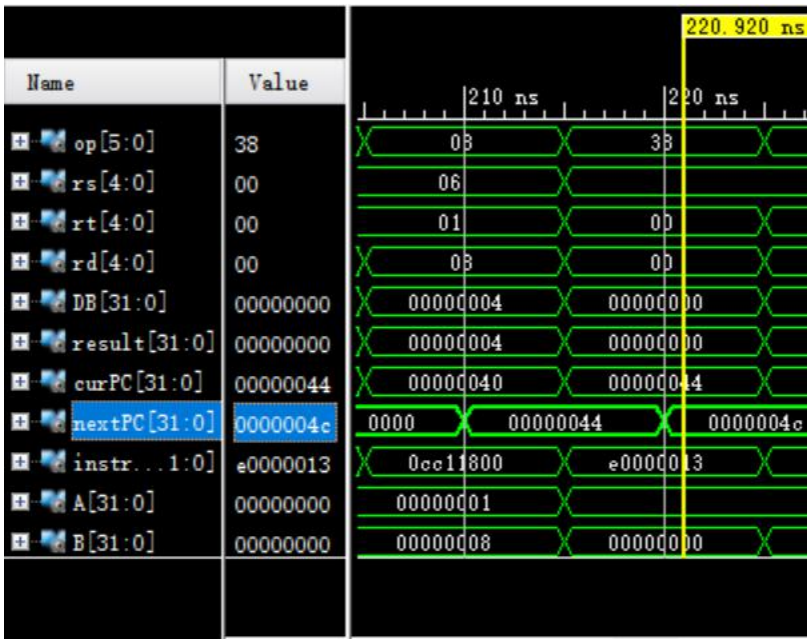
(16)sllv　$2,$1,$6　64c11000（由$1=8,$6=1,$2=8<<1=16=result故结果正确）



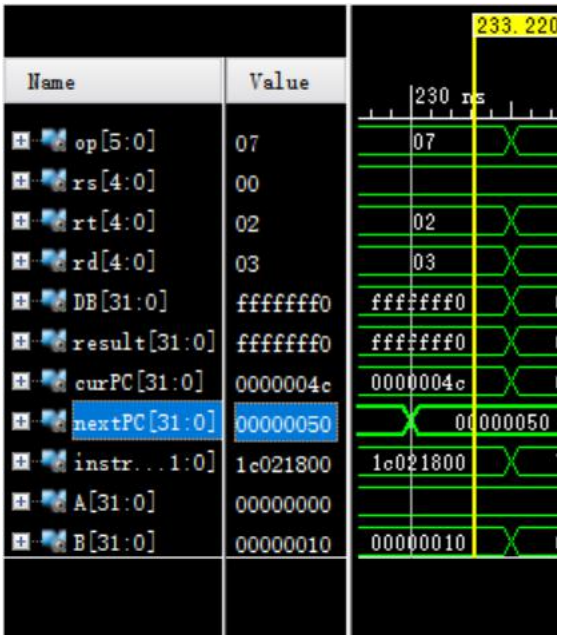
(17)srlv　$3,$1,$6　0cc11800（由$1=8,$6=1,$3=8>>1=4=result可知结果正确）



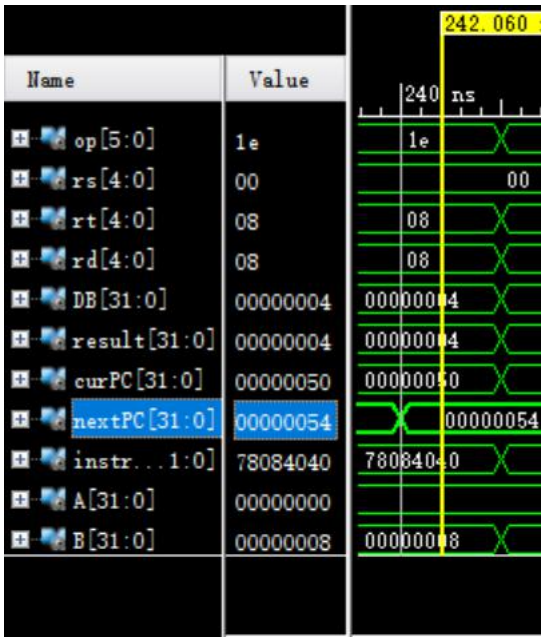
(18)j　0x0000004C　e0000013（由nextPC一栏可知成功跳转至0x0000004C，地址为0x00000048的指令被跳过）



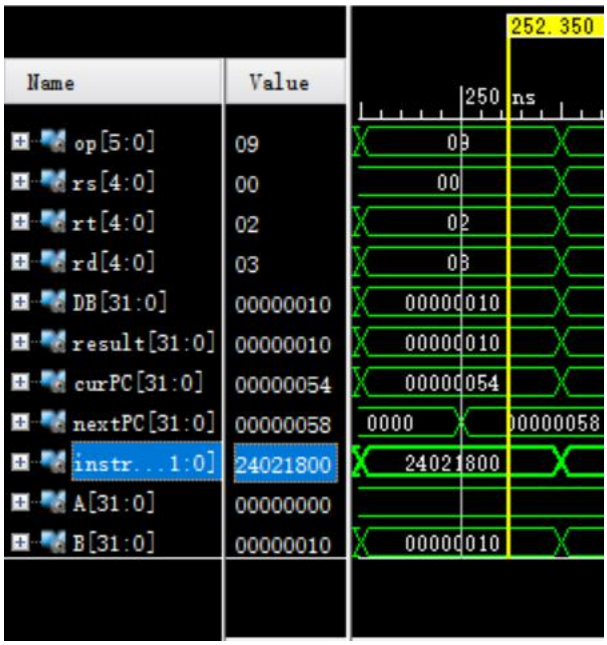
(19)subu　$3,$0,$2　1c021800（由$3=0-16=-16=0xfffffff0=result可知结果正确）



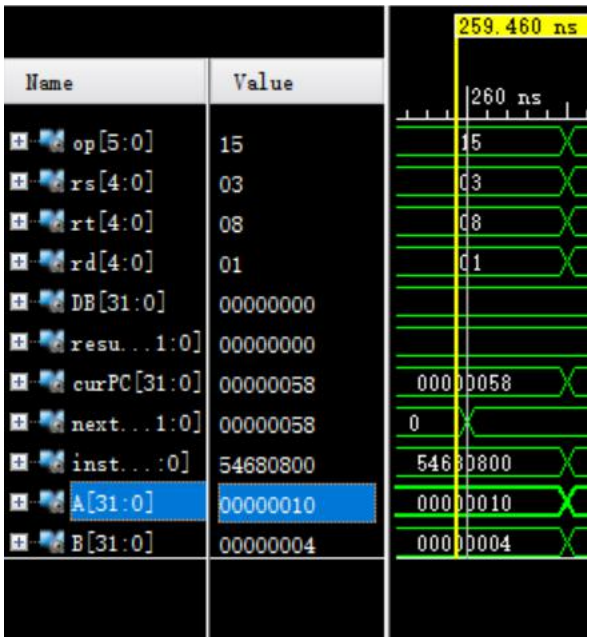
(20)srl　$8,$8,1　78084040（由$8=8>>1=4=result可知结果正确）



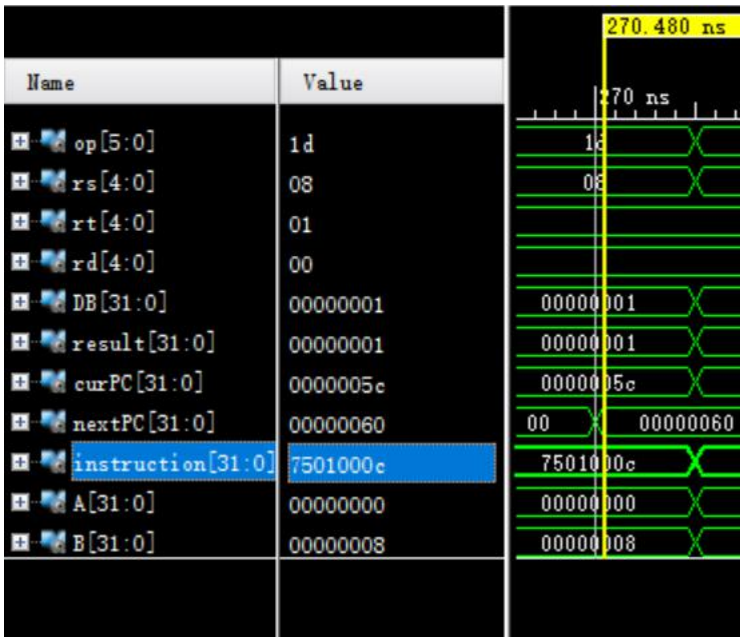
(21)addu　$3,$0,$2　24021800（由$3=0+16=16=result可知结果正确）



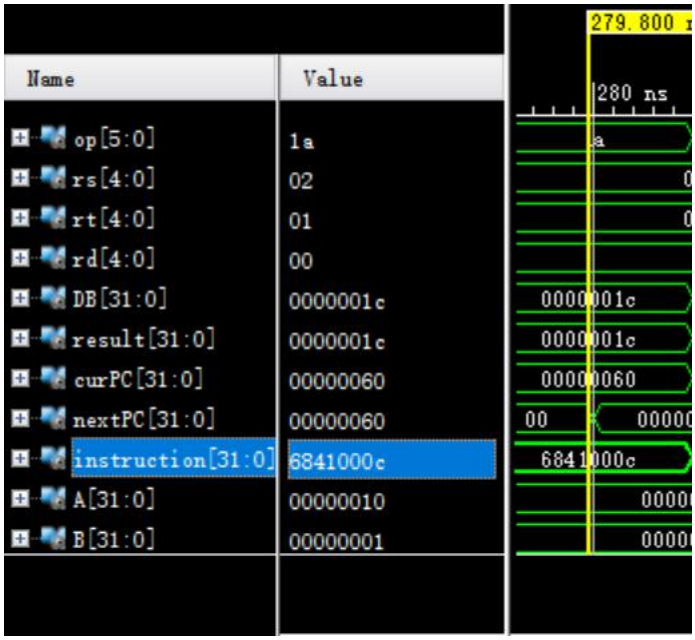
(22)slt　$1,$3,$8　54680800（由$3=16,$8=4,16>4置0，result=0可知结果正确）



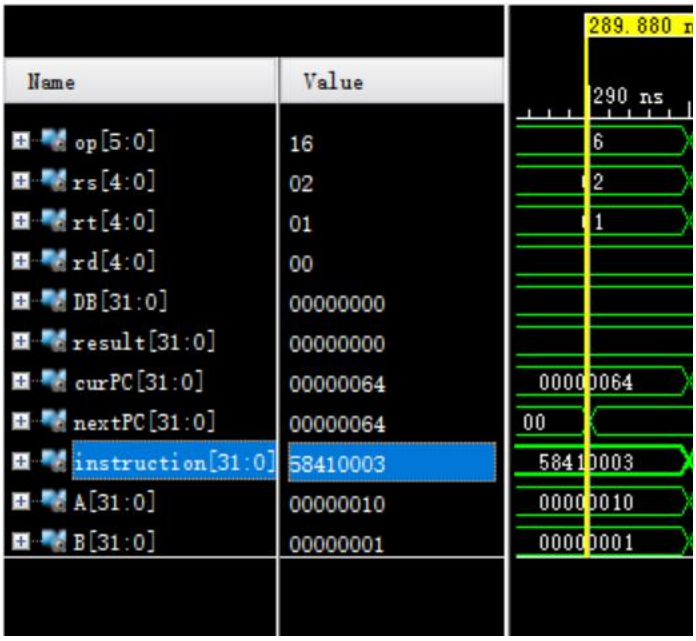
(23)sltiu　$1,$8,12　7501000c（由$8=4<12，$1=1=result可知结果正确）



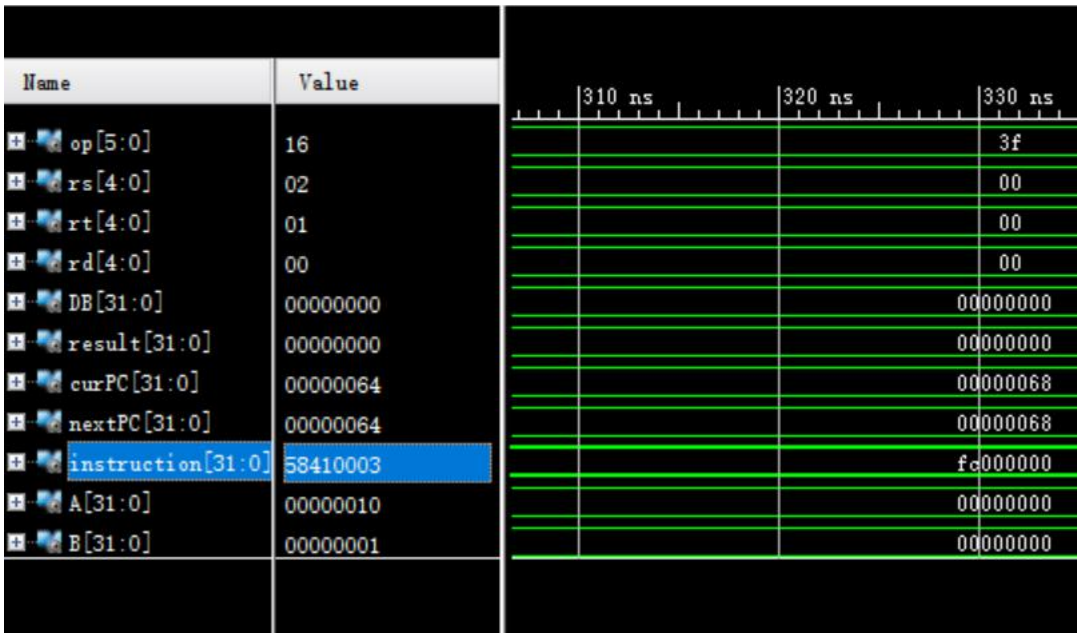
(24)xori $1,$2,12 6841000c （由$2=0x10,$1=0x10^0x0c=0x1c=result可知结果正确）



(25)andi $1,$2,3 58410003（由$2=0x10,$1=$2&3=0=result故结果正确）



(26)halt　FC000000（停机指令，正确）



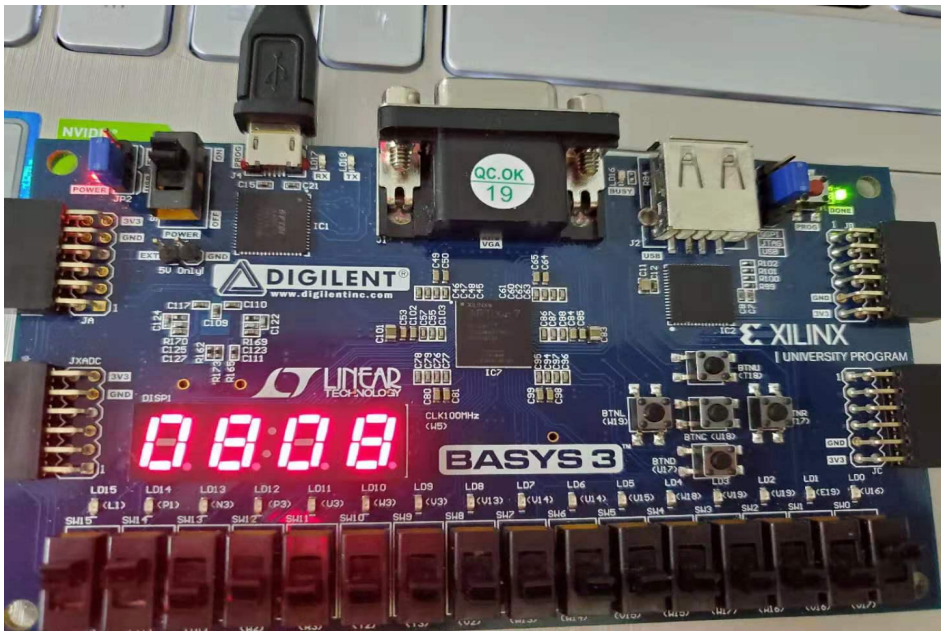
至此全部结束。

2. 烧板结果：

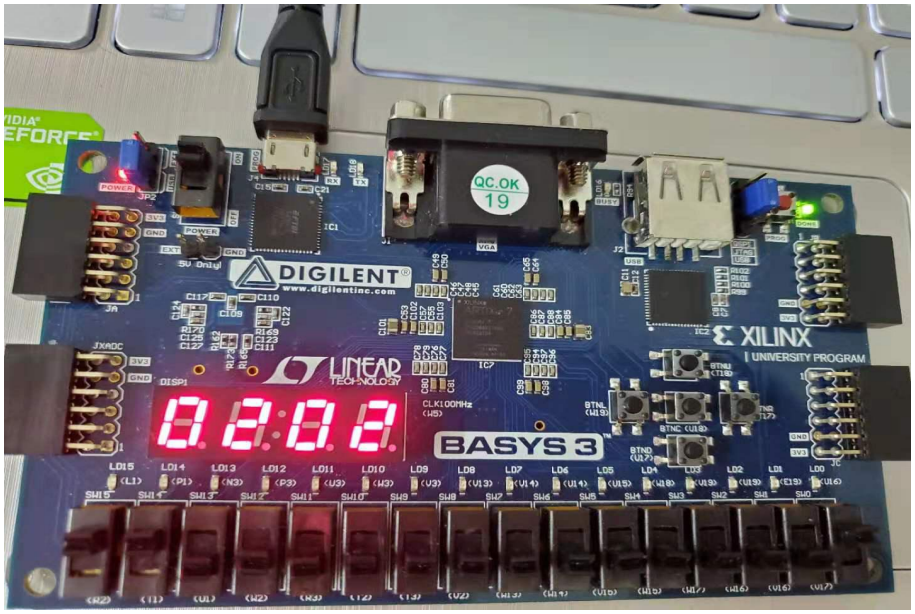
注：最左侧的两个引脚（R2、T1）为 00 时显示当前 PC 和 nextPC；为 01 时显示 rs 寄存器地址和 rs 寄存器里的内容；为 10 时显示 rt 寄存器地址和 rt 寄存器里的内容；为 11 时显示 ALU 结果输出与 DB 总线输出。在检验时需要将最右侧的两个引脚 V17、V16 拨上去；按动 U18 则执行下一条指令。

如图，列举了最前面7条指令的结果，其中数码管左侧两位为result，右侧两位为总线结果DB。其他指令的结果可以查看文件中的视频。

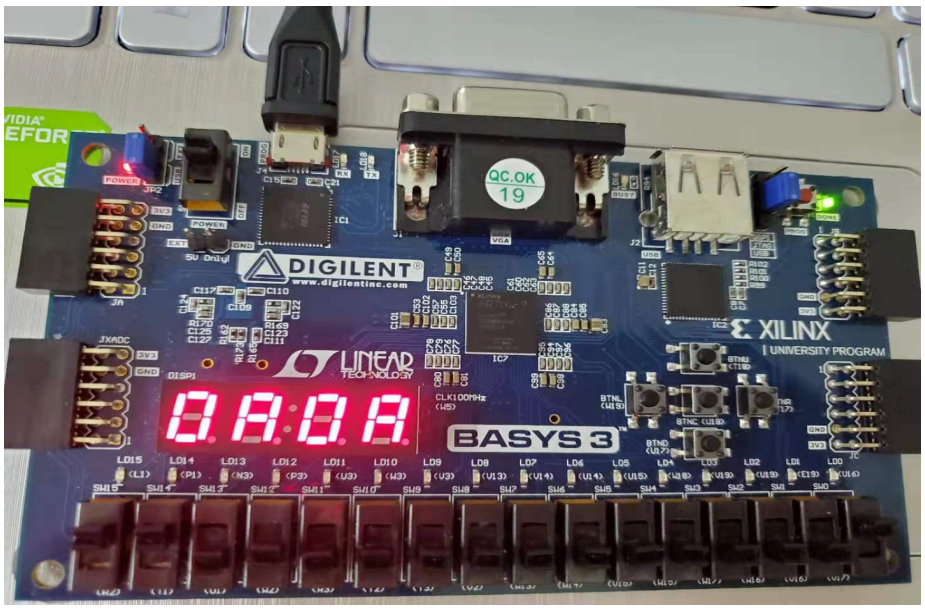
第一条的结果为0+8=8，正确；



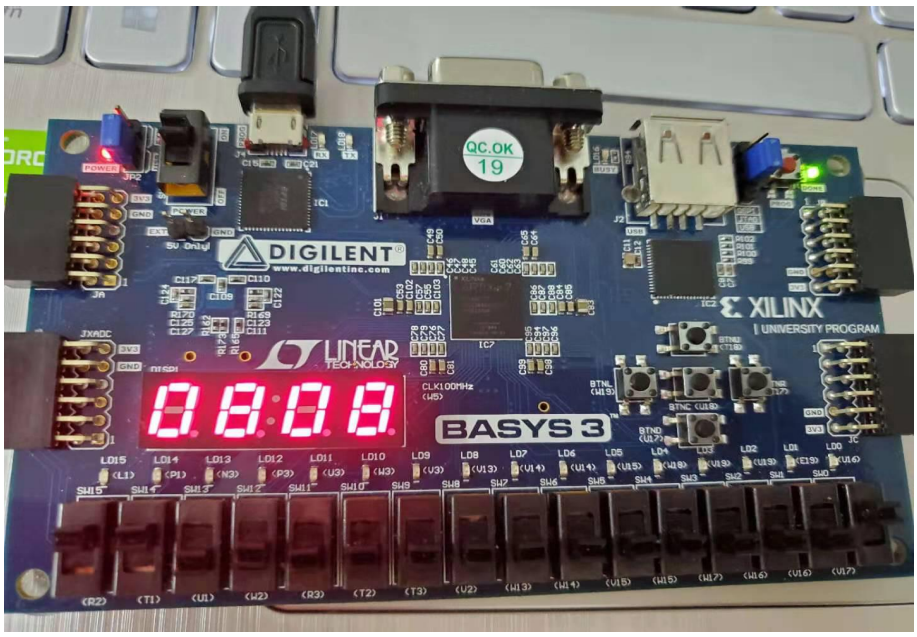
第二条指令的结果为0|2=2，正确；



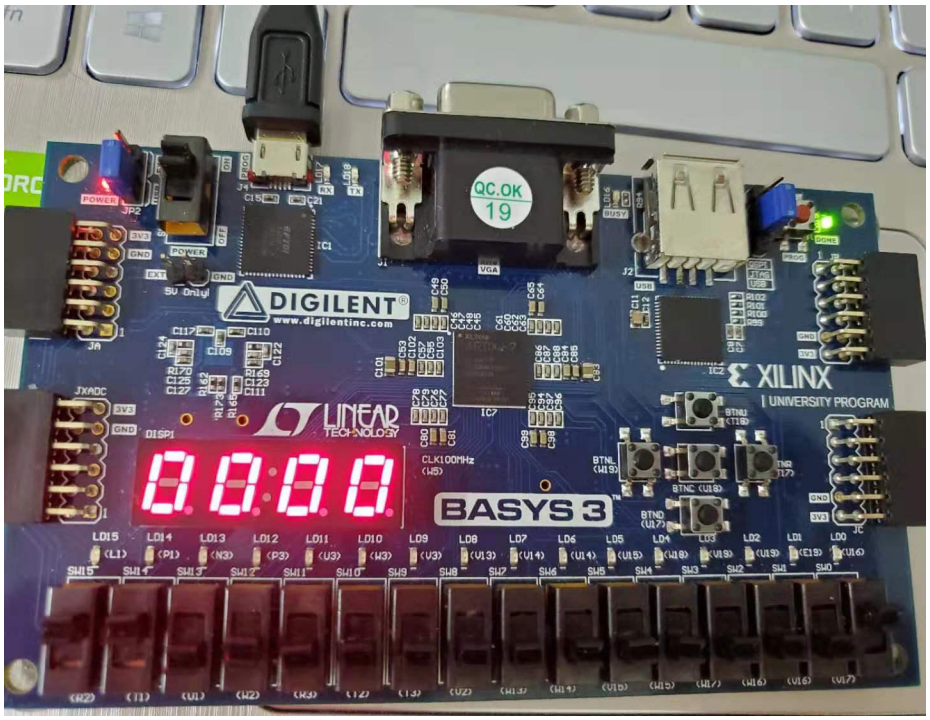
第三条指令的结果为：8+2=10(0x0A)，正确；



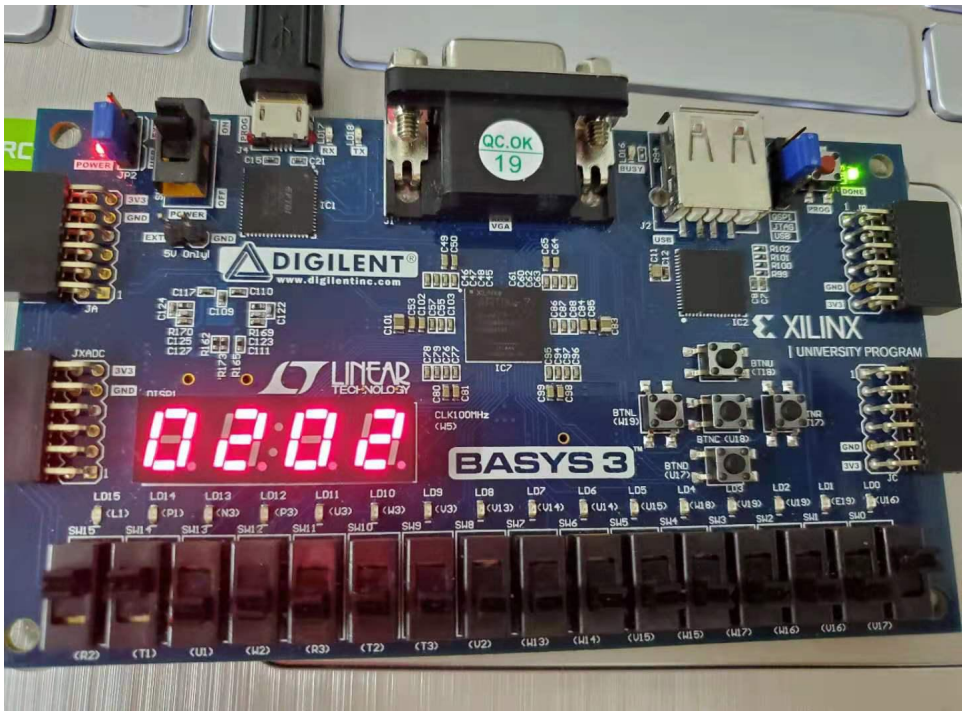
第四条指令的结果为：10-2=8，正确；



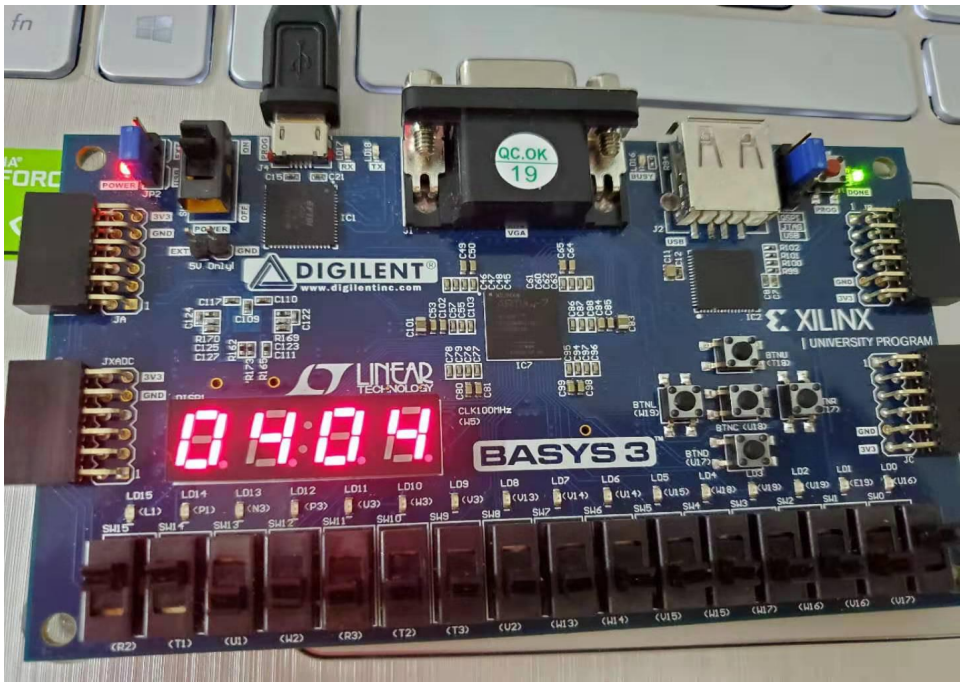
第五条指令的结果为：8&2=0，正确；



第六条指令的结果为：2|0=2，正确；



第七条指令的结果为：2<<1=4，正确；



六. 实验心得：

之前课上都是把 CPU 分解成了每个小的部分，实现其中一个小的模块，所以相对而言比较简单。而这次是第一次做这种大的 project，并不能简简单单的把之前写过的每个小部分合并在一起就能出来一个 CPU，这显然也不现实。所以本次实验中遇到的问题比较多。

首先是 Verilog 语言的问题。Verilog 语言前面没有详细的、系统的学习，所以每回做实验调试代码时基本上就是靠猜测，随便改改，没准什么时候就碰上了正确答案。这次也是遇到了代码的一些语法问题，通过向别人请教，得知有的地方要将 reg 改为 wire。同时我也知道了二者有什么不同：wire 是起到信号间连接的作用，例如顶层模块中，需要将各个模块连接起来，这时候只能用 wire 连接，不能使用 reg；而 reg 则可以用于存储数值，例如寄存器组、数据寄存器里面的存储器必须为 reg 类型。其次 wire 类型只能通过 assign 进行赋值，而 reg 类型只能在 always 里面被赋值。

其次是关于 CPU 的设计。就像前面所说，并不能把之前写的每个小的模块直接合并在一起，而要将他们建立起联系，这个过程是一个艰难的过程。还有就是众多的控制信号，每条指令对应的控制信号分别是什么，用到的 ALU 运算是什么，只有搞清楚了这两点才能实现每条指令，在此过程一定要细心！否则会出现问题。还有就是存放指令机器码的 txt 文件，如果在调试时需要修改 txt 文件，则需要将 vivado 重启后再进行仿真，否则 txt 的更改不会更新！

再者就是烧板的时候出现的问题。在进行移位操作时，仿真结果并没有什么问题，只移位了一次；但是到烧板的时候，每次遇到移位指令时，移位操作会执行两次，后来了解到是因为移位操作会在时钟上升沿和下降沿都触发，而其他指令则只会触发一次，所以烧板时只有移位操作才会显示两下。还有就是消抖：在网上寻找相关资料时看到好多都提到了这个问题，为了烧板时能保证 CPU 正常运行，因此也添加了消抖文件。

问题不少，但收获更多。

一、通过这次 project 更能体会到设计遵循的原则：将一项大的 project 分成多个小的模块进行完成，然后再将其组合起来。二、其次就是学会了更多有关 Verilog 语言的知识。三、对于一个小部分，有时可能会有不同的实现方法。比如控制器部分可以通过

用真值表的方法分析问题并写出逻辑表达式，或者用 case 语句方法逐个产生各指令控制信号；指令存储器的实现可以是直接在代码中加入存放指令的 txt 文件的绝对地址，或者在 vivado 中创建 coe 文件来存放待检测的指令。四、由于单周期 CPU 的理论课还是在期中考试以前讲完，到现在有一些忘却，正好通过这次实验能够复习下前面的知识，加深对单周期 CPU 的理解。为后面实现流水线 CPU 提供了一个良好的基础。