《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）：	计算机科学与技术（超算）
学 生 姓 名 ：	李钰
学　　号 ：	19335112
时　　间 ：	2020 年 12 月 18 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
2. 掌握单周期CPU的工作原理与逻辑功能实现
3. 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. 用 verilog 语言实现单周期CPU模拟，至少实现16条MIPS指令，至少包含:

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。

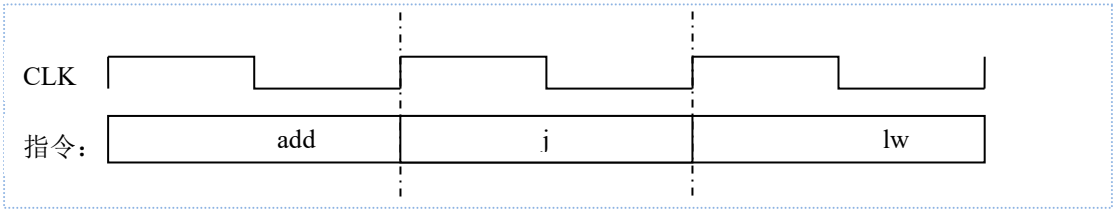
3. 可拓展添加其他指令。

三. 实验原理

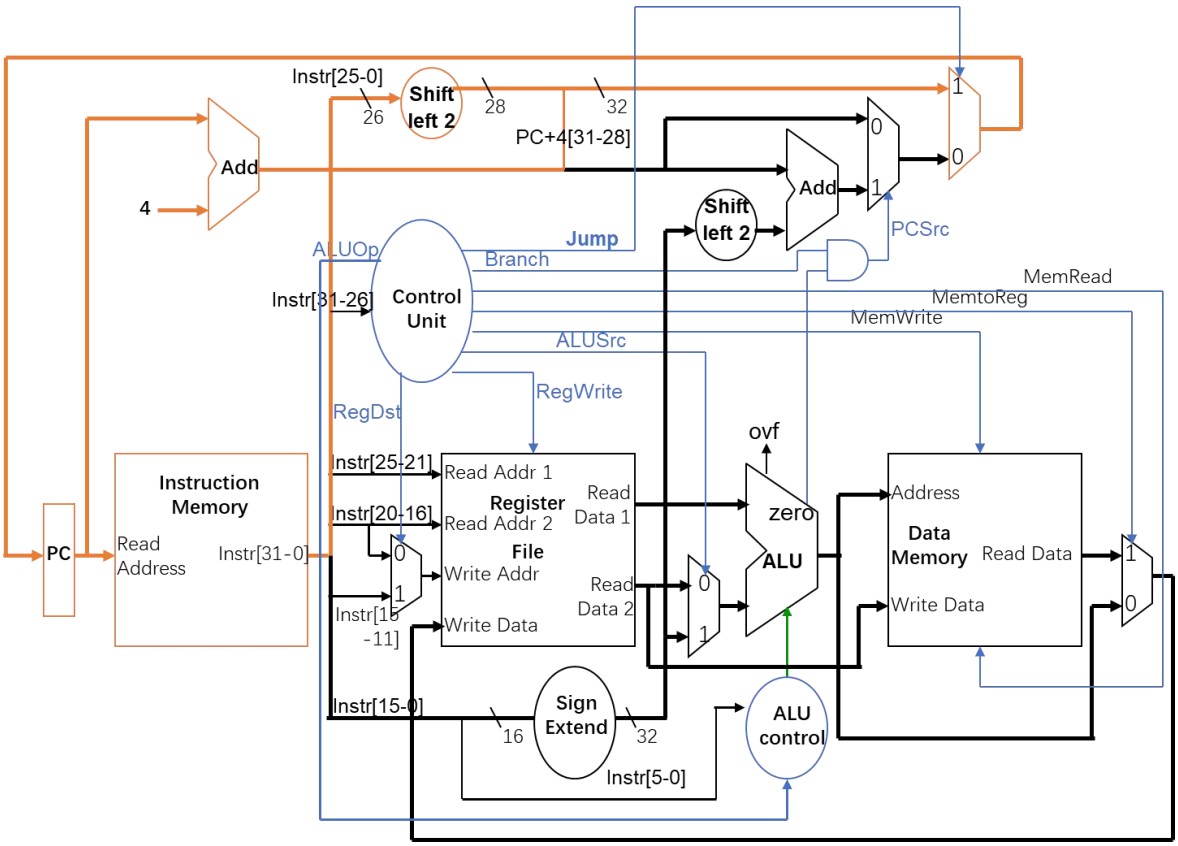
1. 单周期CPU

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。



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：



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；

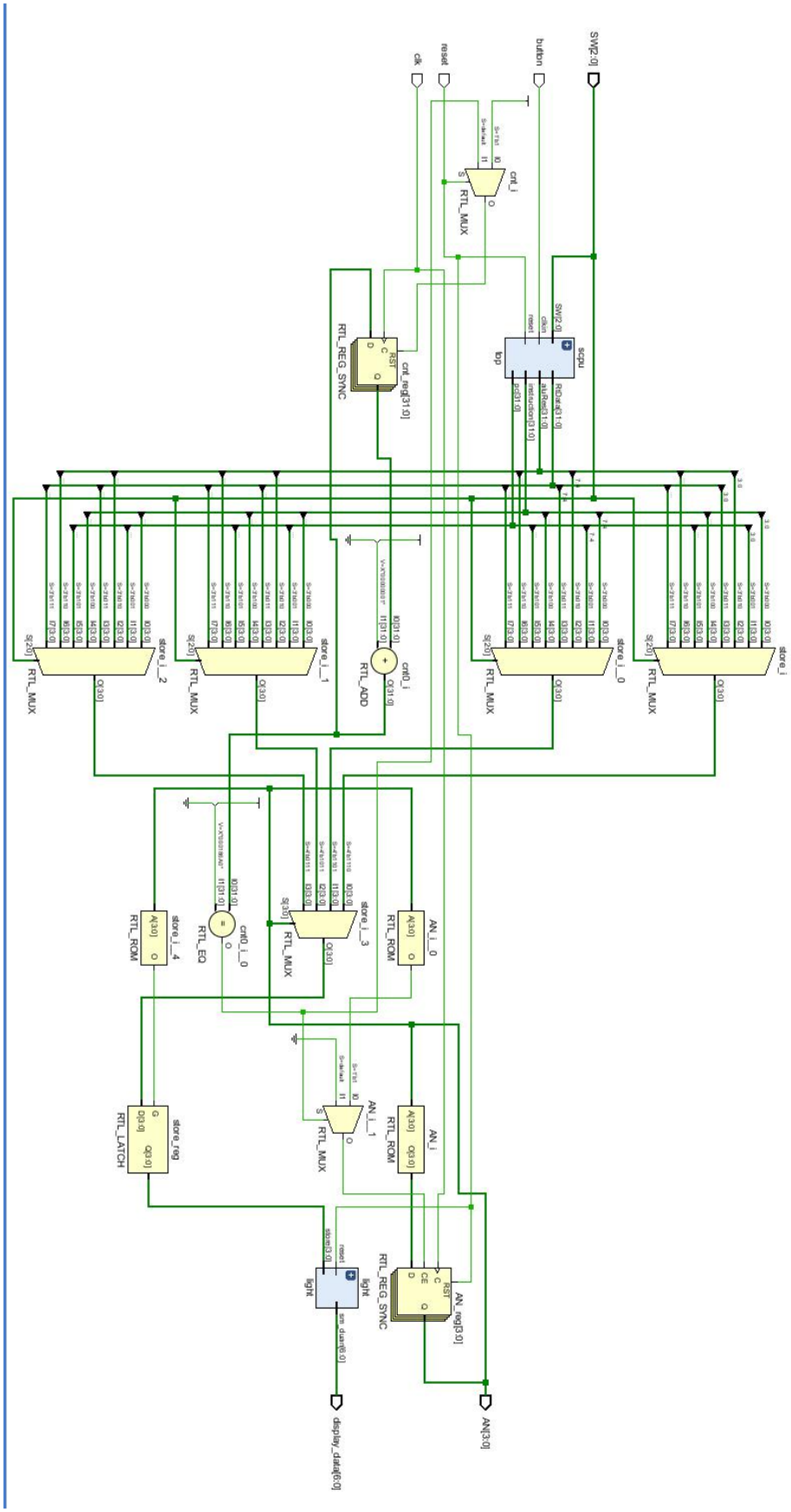
（6）ALUSrc：ALU 操作数 B 的选择，为 1 时，选择扩展的立即数；为 0 时，选择寄存器数据；

（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。

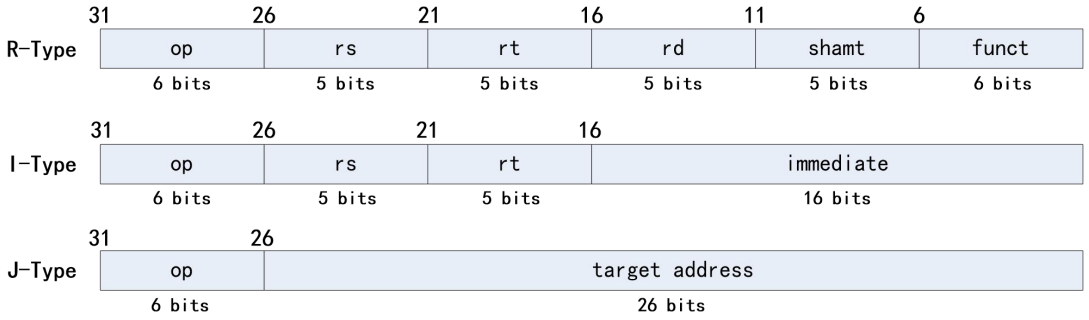
下图为 verilog 语言编写后实现的单周期 CPU 原理图



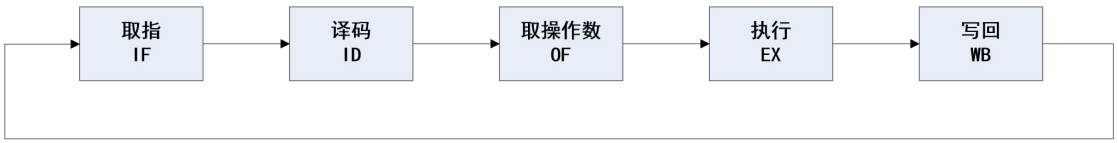
2．MIPS指令集

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）；



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：



四．实验器材

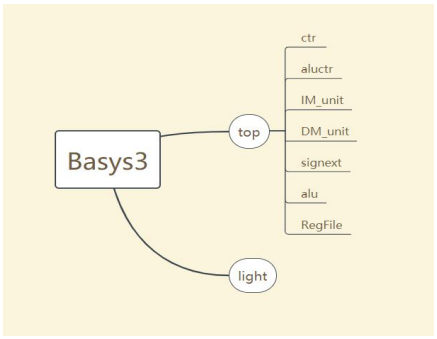
电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五．实验过程与结果

1. CPU的设计

对于此次 CPU 的设计，我一共设计了，三层、10 个模块，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。

各模块结构如下：



A. 最顶层文件—— Basys3 ：主要进行时钟分频操作，以及确定四位七段数码管每一位的输出。其中涉及到的参数

```
`timescale 1ns / 1ps
    module Basys3(
    input clk,//for display
    input[2:0] SW,
    input reset,
    input button,   //clkin
    output reg[3:0] AN,  //sm_wei
    output[6:0] display_data//sm_duan
);
    parameter T1MS=100000; //消除抖动
    integer cnt;
    wire [31:0] aluRes,instruction, RtData, pc;
        reg [3:0] store;    //存每一位对应的值

    //实例化top 略
```

```
     always@(posedge clk)
         begin
         if(reset==1)begin
           cnt=  0;
           AN<=  4'b1111; //若reset有效，则输出为四个灯全灭
         end else begin
             cnt = cnt+1;
             if(cnt==T1MS) //调整频率为人眼可分辨
                 begin
                     cnt=    0;
                     case(AN) //对应哪个七段数码管亮
                         4'b1110:    AN<=    4'b1101;
                         4'b1101:    AN<=    4'b1011;
                         4'b1011:    AN<=    4'b0111;
                         4'b0111:    AN<=    4'b1110;
                         4'b0000:    AN<=    4'b0111;
                     endcase
                 end
             end
         end

     //实例化light 略
```

接下来，将每一位七段数码管分情况赋值，这里不全列举了

```
always@(*)begin//将对应的数据写入store
        case(AN)
            4'b1110:    begin//亮右一数码管
                case(SW)
                    3'b000:    store<=    instruction[3:0];
                    3'b001:    store<=    pc[3:0];
                    3'b010:    store<=    aluRes[3:0];
                    3'b011:    store<=    RtData[3:0];
                    3'b100:    store<=    instruction[19:16];
                    3'b101:    store<=    pc[19:16];
                    3'b110:    store<=    aluRes[19:16];
                    3'b111:    store<=    RtData[19:16];
                endcase
            end
```

Basys3 文件中下设两个子文件，命名分别为 light 和 top。

B. light 模块：主要的作用是将数值转为七段数码管的段码，以便将来亮灯显示出正确的数字；

```
module light(
input [3:0] store,
input reset,
output reg [6:0] sm_duan
    );
    always@(*)
//always@(store or reset)
        begin
            if(reset) begin
                sm_duan=    7'b0000000;
            end
            else begin
                case(store)
                    4'b0000:    sm_duan=    7'b100_0000;//0
```

C. top 模块：包含了 7 个子文件，细化 CPU 内部功能，其详细介绍如下：

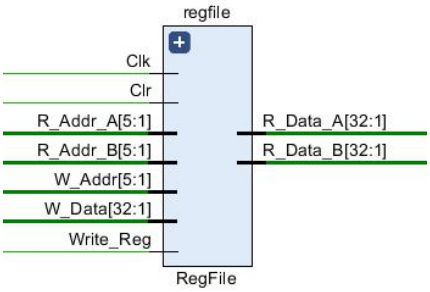
（1）指令存储器模块：具备基本的读写功能，用于存放指令。该模块将32PC值输入和32位数据输出，首先将24条指令写入存储单元中，然后根据PC值的第2~9位输入选择相应的单元指令内容，将数据写入到输出变量中。

```
module IM_unit(
    input clk,
    input reset,
    input [31:0] pc,        //指令存储器地址编码
    output [31:0]  instruction// 寄存器的值
);

reg [31:0] regs [0:29]; // 寄存器组

initial
        begin
                        regs[0]  = 32'h20110001;// addi $s1,$0,1   #s1 = 1, aluRes = 1
```

指令均以以上方式存放在该存储器中， 以下语句进行取指令操作

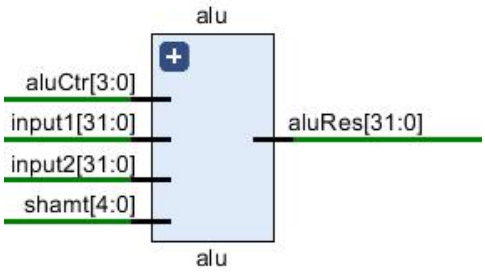
```
assign instruction = regs[pc[9:2]] ; //  取指令
```

（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。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;//寄存器数据位宽
    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@(posedge 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,检测写控制，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，使用数据流建模(组合逻辑电路,读不需要时钟控制)
    assign R_Data_A=REG_Files[R_Addr_A];
    assign R_Data_B=REG_Files[R_Addr_B];
endmodule
```

（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。其结构如下：



input1：操作数 32 位，输入；

input2：操作数，32 位，输入；

aluCtr：4 位操作码，输入，控制 ALU 做何种运算

aluRes：运算结果，32 位，输出；

ZF：零标志，1位；当运算结果为0时，该位为1，否则为0；

```
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
);
always @(input1 or input2 or aluCtr) // 运算数或控制码变化时操作
begin
case(aluCtr) 4'b0000: //  与 and
begin
aluRes = input1 & input2;
if(aluRes==0) ZF=1;
else ZF=0;
end
```

其他运算如上按情况给出，这里不过多赘述，详见代码

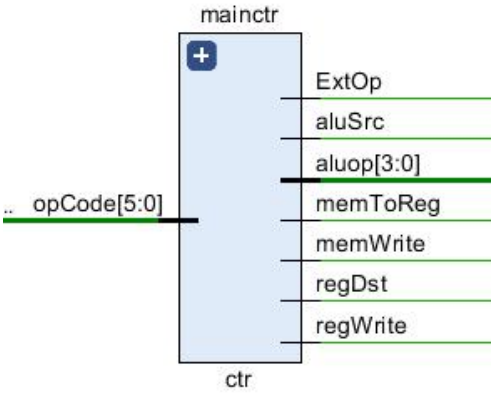
（4）立即数扩展模块：执行 I 型指令时需要立即数扩展，该模块用于 MIPS 符号扩展，将 16 位数据扩展为 32 位数据。根据符号补充符号位，如果符号位为 1，则补充 16 个 1，即 16'h ffff；如果符号位为 0，则补充 16 个 0。

```
module signext(
input [15:0] inst, // 输入16位
input  ExtOp,
output [31:0] data // 输出32位
);
// 根据符号补充符号位
 //  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
```

（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。 通过 6 位输入 opCode, 即指令的后 6 位，来判断其指令类型——如， R 型、J 型、lw、 sw 等等，进而输出相应的控制信号，并将 aluop 进行赋值，以便后续 alu 计算时用到，其控制信号为：

- jmp：为 1 时，选择跳转目标地址；为 0 时，选择由 Branch 选出的地址；
- memToReg：为 1 时，选择存储器数据；为 0 时，选择 ALU 输出的数据；
- branch：为 1 时，选择转移目标地址；为 0 时，选择 PC +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；
- aluop：ALU 控制码；
- 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；
- 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；
- ExtOp：符号扩展。为 1 时，符号扩展；为 0 时，0 扩展；
- RegDst：目的地址，为 1 时，选择 rd；为 0 时，选择 rt。

主控制器结构图如下：



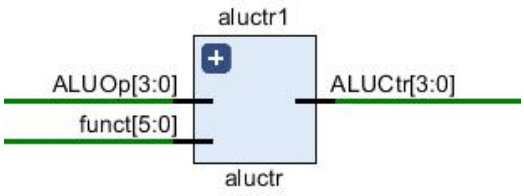
```
module ctr(
input [5:0] opCode, output reg regDst, output reg aluSrc, output reg memToReg, output reg regWrite, output
reg memRead, output reg memWrite, output reg branch,
output reg ExtOp, //符号扩展方式，1 为 sign-extend，0 为 zero-extend
output reg[3:0] aluop, // 经过 ALU 控制译码决定 ALU 功能
output reg jmp
);
always@(opCode) 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，无需 ALU
```

（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。

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alu控制信号

aluop[3:0]	运算	aluCtr[3:0]
XXXX	J型指令	x
0000	and	0000
0000	or	0001
0010	ori	0001
0011	lw、sw、addi	0010
0000	add	0010
0110	bne	0011
1000	xori	0100
0000	nor	0101
0000	sub	0110
0101	beq	0110
0000	slt	0111
0000	sll	1000
0000	srl	1001
0000	sra	1010
0000	srlv	1011
0111	lui	1101
0000	srav	1110
0000	subu	0110
0000	xor	0100
0011	addiu	0010

控制器设计如下：



代码构成

```
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'b0000_100100: ALUCtr = 5'b0000; // and
```

（7）数据存储器模块，用于 LW/SW 指令数据存取，将输入的数据存到数据存储器中对应地址中去

（8）top 文件： 将所有小模块连接到一起，进行实例化

2. CPU功能正确性验证

测试代码

地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	aluRes
0x00000000	addi $s1,$0,1	0010_00	00_000	1_0001	0000_0000_0000_0001	20110001	00000001
0x00000004	add $s2,$0,$s1	0000_00	00_000	1_0001	1001_0	00119020	00000001
0x00000008	sll $s3, $s2, 2	0000_00	00_000	1_0010	1001_1000_1000_0000	00129880	00000004
0x0000000c	or $t1,$s1,$s3	0000_00	10_001	1_0011	0100_1	02334825	00000005
0x00000010	sw $t1, 0($t0)	1010_11	01_000	0_1001	0000_0000_0000_0000	ad090000	00000000
0x00000014	lw $s0,0($t0)	1000_11	01_000	1_0000	0000_0000_0000_0000	8d100000	00000000
0x00000018	**beq $t1,$s0,next1**	0001_00	01_001	1_0000	0000_0000_0000_0001	11300001	00000000
0x0000001c	nor $t1, $t3, $t2	0000_00	01_011	0_1010	0100_1	016a4827	
0x00000020	**next1:** sllv $t2,$s2,$s0	0000_00	10_000	1_0010	0101_0000_0000_0100	02125004	00000020
0x00000024	**j next2**	0000_10	00_000	1_0000	0000_0000_0000_1011	0810000b	00000000
0x00000028	nor $t1, $t3, $t2	0000_00	01_011	0_1010	0100_1	016a4827	
0x0000002c	**next2:** ori $t2, $t2, 2	0011_01	01_010	0_1010	0000_0000_0000_0010	354a0002	00000022
0x00000030	nor $t3, $t2, $t1	0000_00	01_001	0_1001	0101_1	01495827	Fffffd8
0x00000034	lui $t4, 12345	0011_11	00_000	0_1100	0001_0100_1110_0101	3c0c14e5	14e50000

0x00000038	xori $t4, $t1, 3	0011_10	01_001	0_1100	0000_0000_0000_0011	392c0003	00000006
0x0000003c	sub $t1, $t4, $s3	0000_00	01_100	1_0011	0100_1	01934822	00000002
0x00000040	slt $t1, $0, $t2	0000_00	00_000	0_1010	0100_1	000a482a	00000001
0x00000044	**bne $t1,$0, next3**	0001_01	01_001	0_0000	0000_0000_0000_0010	15200002	00000001
0x00000048	addi $s1,$0,1	0010_00	00_000	1_0001	0000_0000_0000_0001	20110001	
0x0000004c	add $s2,$0,$s1	0000_00	00_000	1_0001	1001_0	00119020	
0x00000050	**next3:** **srl $t3, $t4, 1**	0000_00	00_000	0_1100	0101_1000_0100_0010	000c5842	00000003
0x00000054	srav $t3, $t2, $s1	0000_00	10_001	0_1010	0101_1	022a5807	00000011
0x00000058	srlv $t3, $t2, $s1	0000_00	10_001	0_1010	0101_1	022a5806	00000011
0x0000005c	xor $t2, $t3, $s2	0000_00	01_011	1_0010	0101_0	01725026	00000010

仿真波形：

其中，button 对应时钟周期；reset 为复位信号，高电平有效；

SW 最高位控制板上显示结果的高位/低位，剩下两位组成 4 个组合：

00 时对应指令；

01 时对应为 PC 值；

10 时对应 alu 计算结果；

11 时为 rt 线路上的数据；

pc 为当前周期进行的指令对应的 pc 值；

aluRes 为 alu 计算结果，即总线输出上的值；

Instruction 对应当前周期执行的指令的十六进制表示；

RtData 为当前指令，rt 寄存器中存储的数

下面 7 个图为上表中所有指令的仿真结果

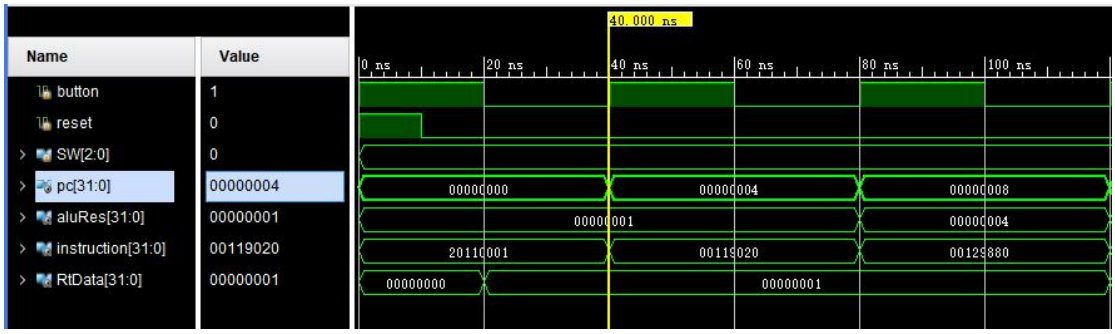


图 1

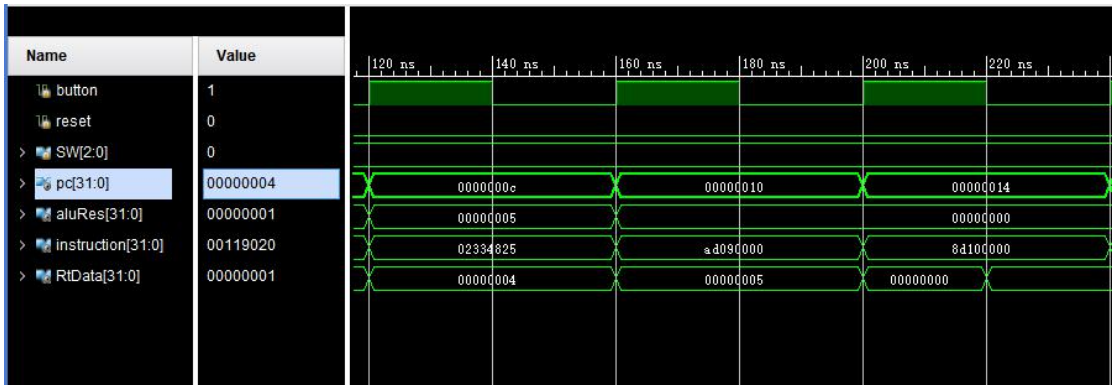


图 2

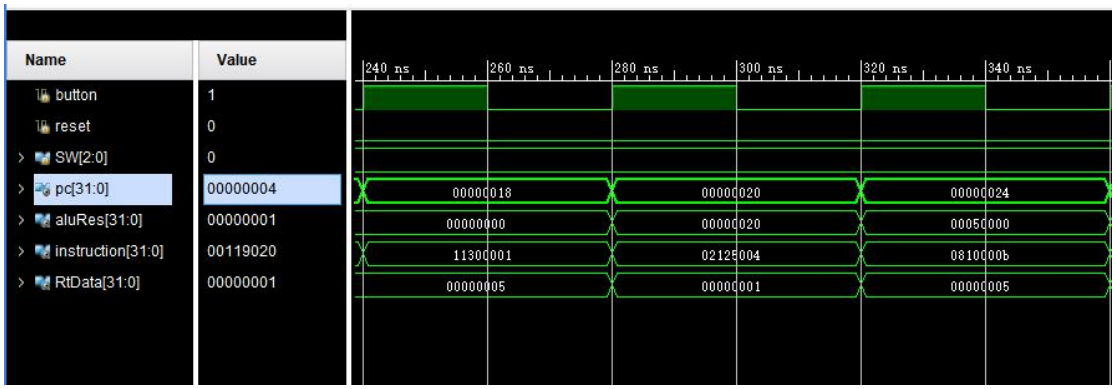


图 3

该图反映出了，当 pc 值为 18 时进行了跳转，下一条指令 pc 值为 20，说明其跳过了 1 条指令，对应表格中，我们可以看到 pc 为 18 时对应 beq 指令，跳转到了 next2 处

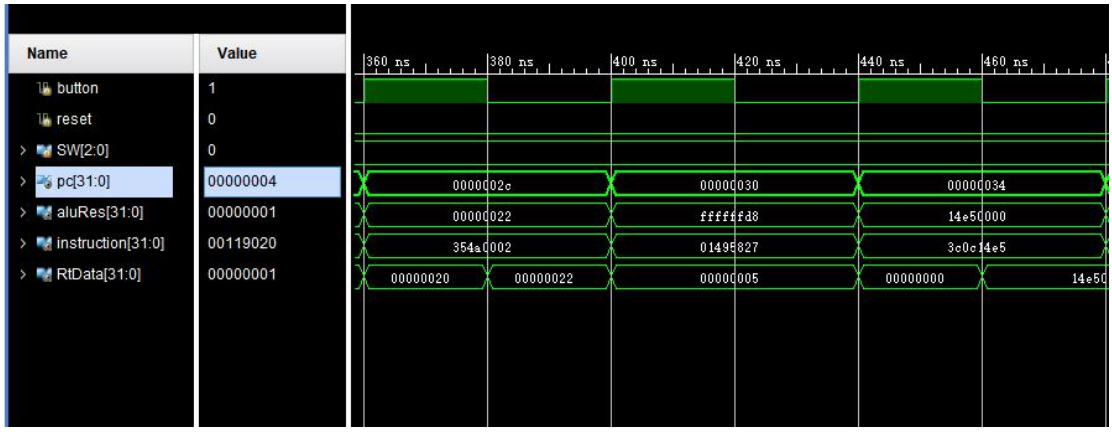


图 4

图 3 的最后一个时钟周期进行的指令为 j，对应 pc 值为 24，也进行了跳转，他的下一时钟周期运行的是 pc 值为 2c，跳过了 1 条指令，对应表格，其计算结果也正确

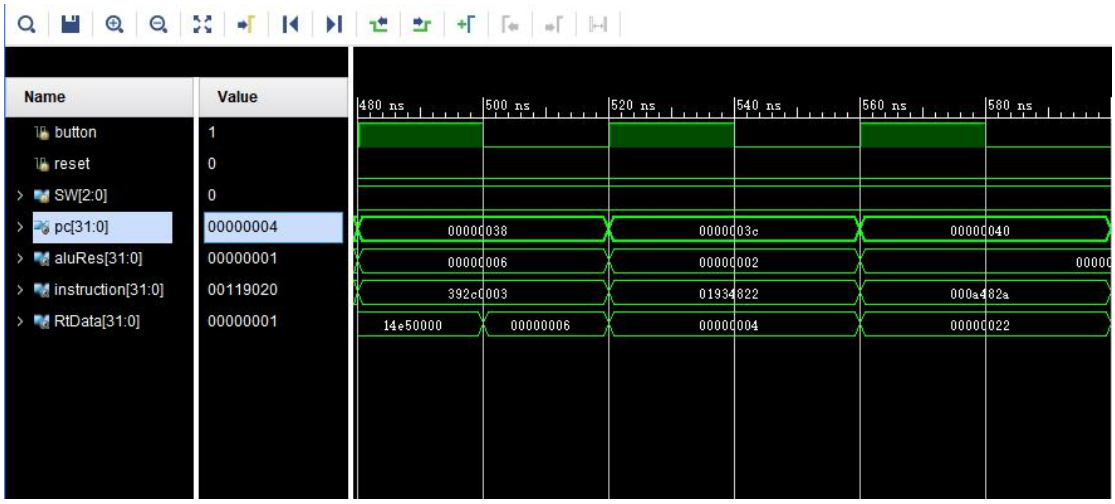


图 5

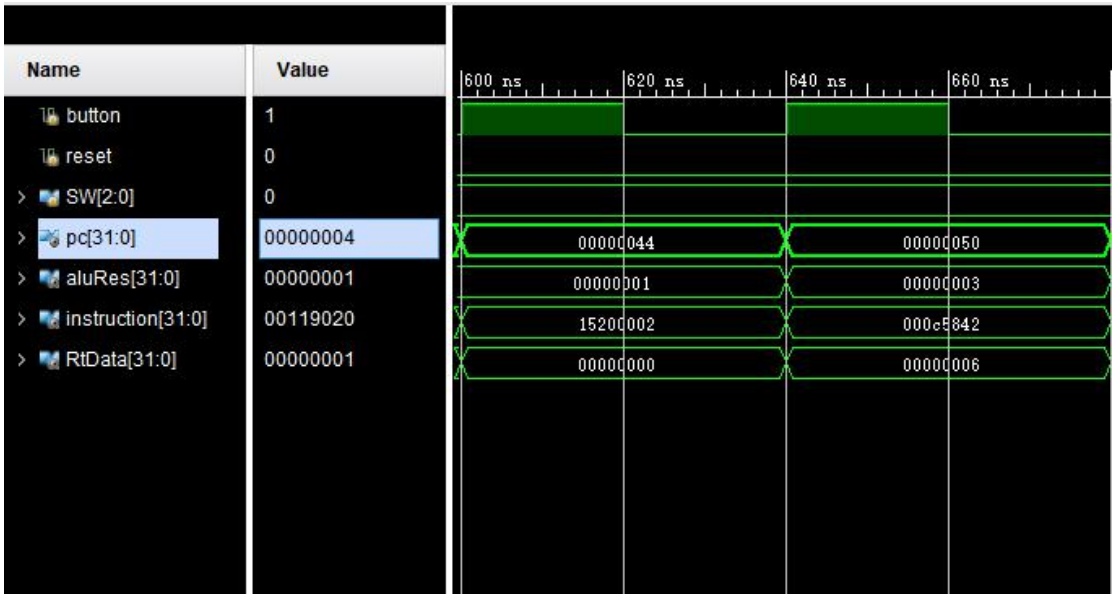


图 6

观察 pc 值变化，该图又进行了跳转，执行的是 bne 指令

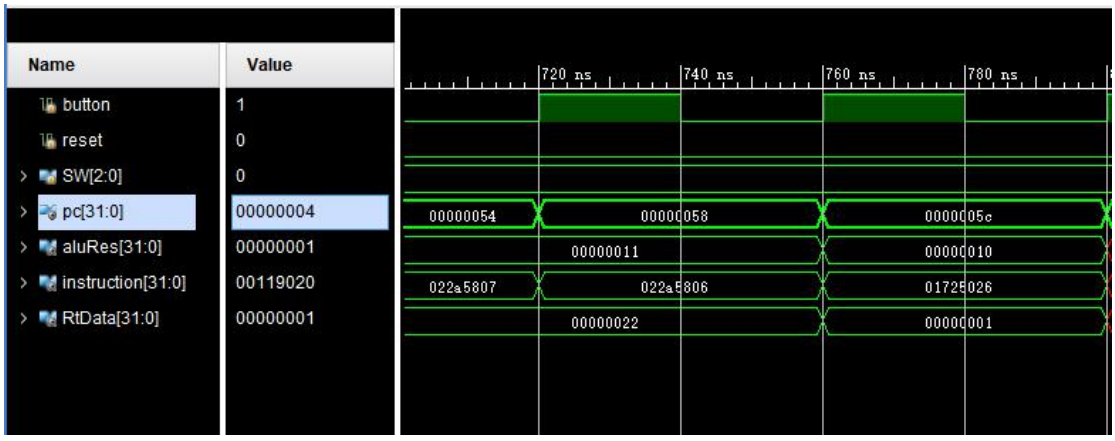


图 7

实物演示：

板子上共设五个开关按键，

最左端三位 SW15 为 0 时输出 32 位数的低 16 位， 为 1 时，输出 32 位数的高 16 位数

SW14 ~ SW13 ：

00 时对应指令；

01 时对应为 PC 值；

10 时对应 alu 计算结果；

11 时为 rt 线路上的数据

最右端 SW0 表示 reset， 高电平有效； SW1 为 button 上拨触发时钟上升沿

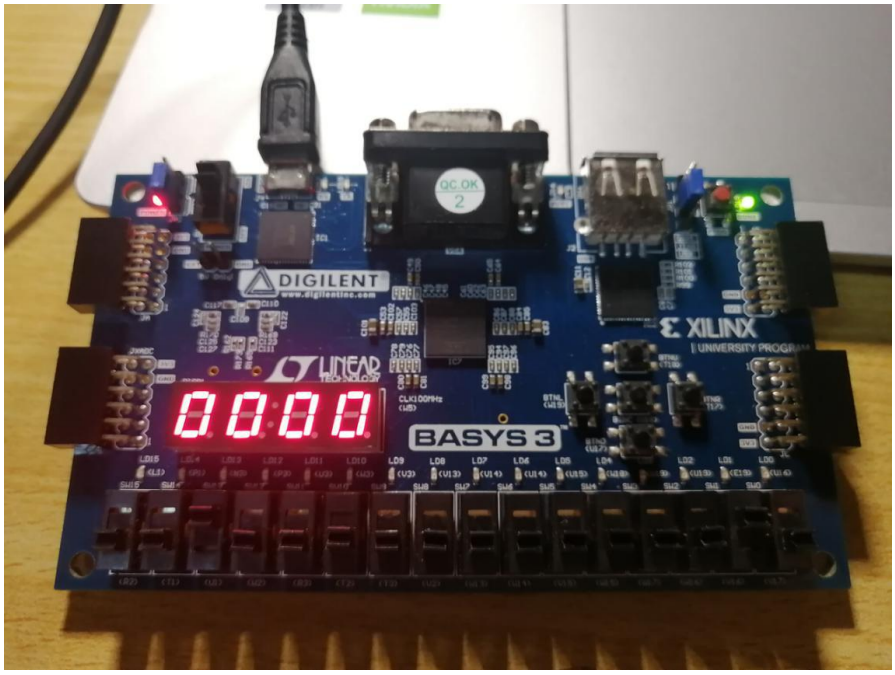


图 8

该图中

SW = 001

显示的是 PC 值的低 16 位，此时读取第一个指令寄存器中的指令，对应 PC 值为 0

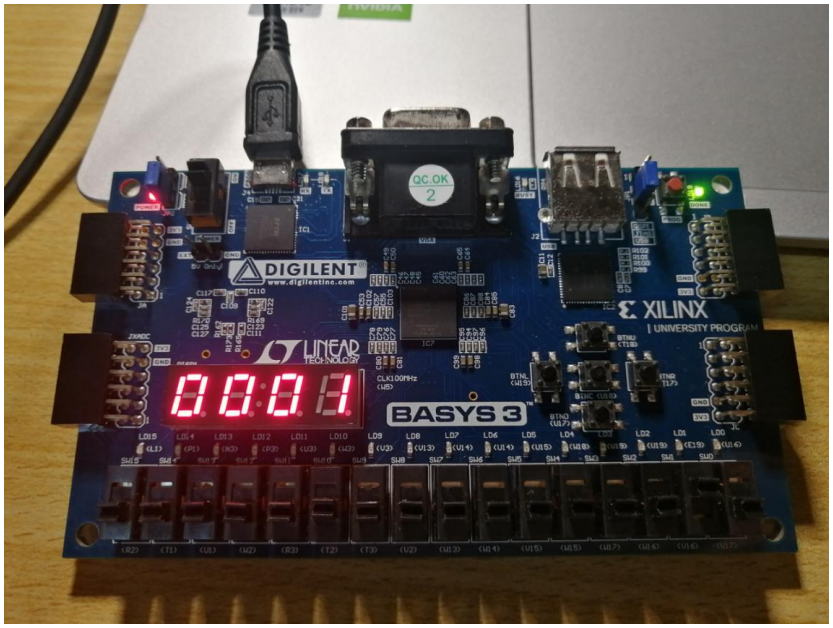


图 9

该图中

SW = 000

显示为第一条指令的低十六位，依照表格可以验证该显示正确。

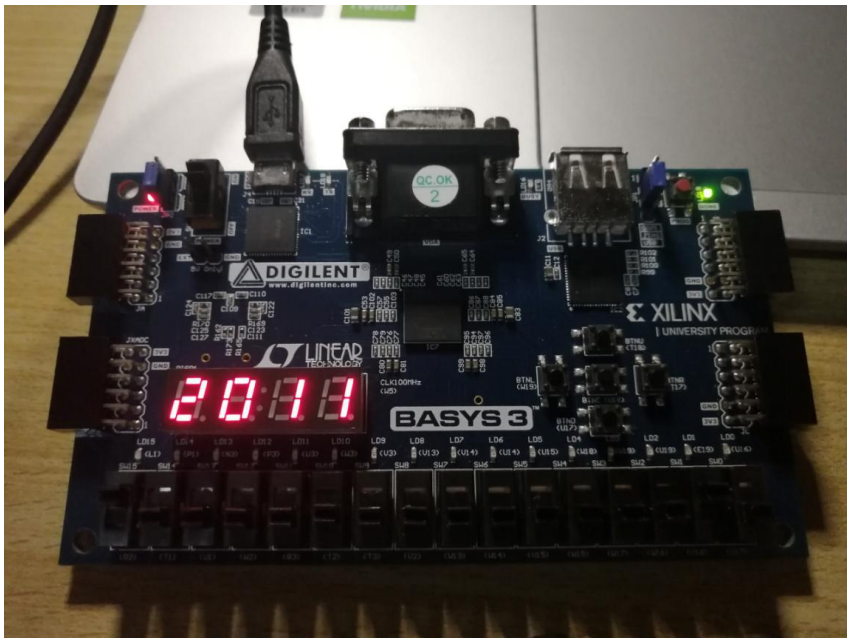


图 10

该图中

SW = 100

显示为第一条指令的高 16 为，对照表格第一条指令为 20110001，显示正确。

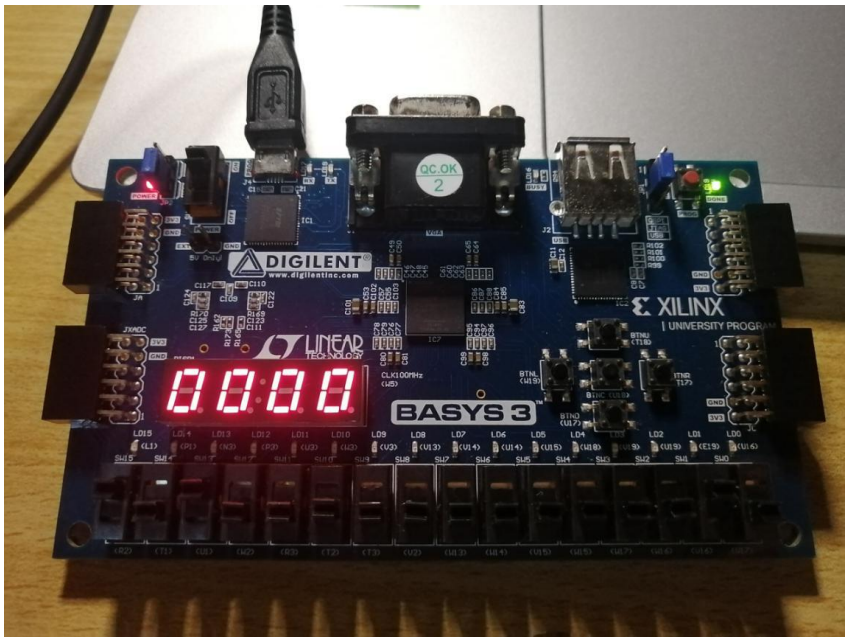


图 11

该图 SW = 101

显示为 aluRes 即 alu 运算结果的高十六位，对照表格，其正确值为 1，高 16 位均为 0，显示正确。

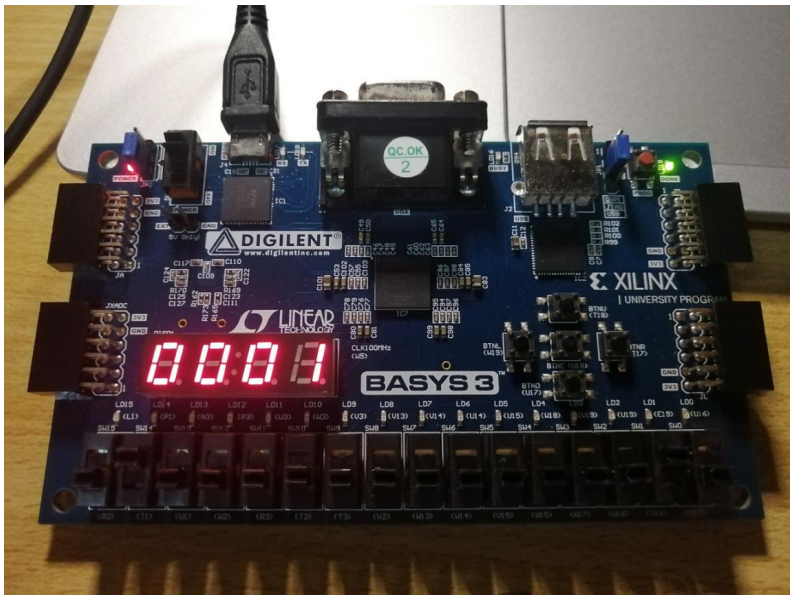


图 12

该图中 SW = 010

显示的是 aluRes 的低 16 位，其值为 1，对应表格中第一条指令，我们可以看到 $s1 = $s0 + 1，数据总线中 alu 的运算结果为 1，验证 CPU 运行正确。

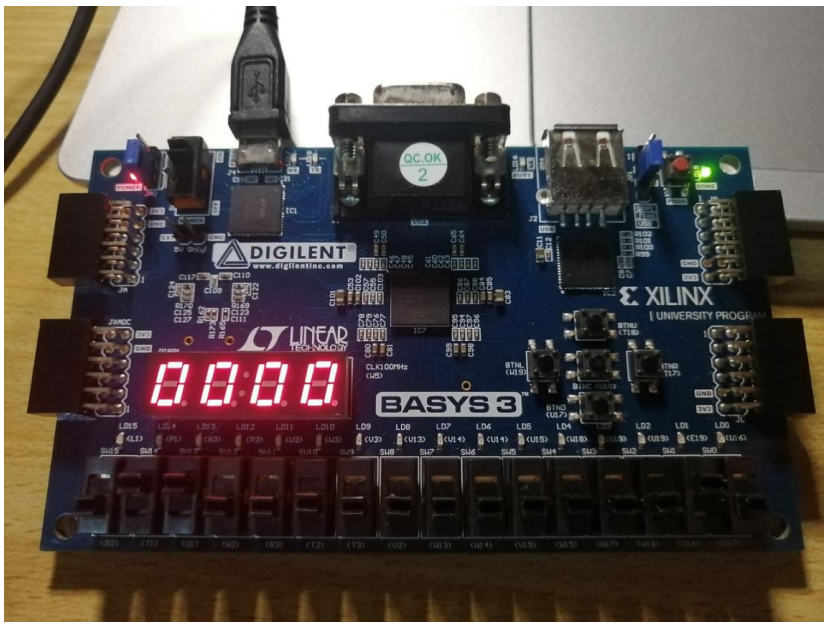


图 13

该图 SW = 011

输出显示为第一条指令对应 rt 寄存器的低 16 值，因为第一条指令为 I 型指令，其 rt 寄存器对应的位置为 0

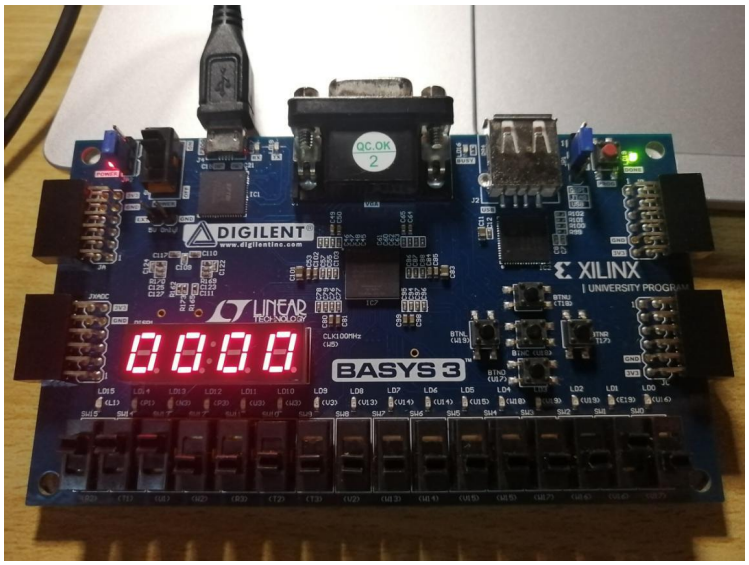


图 14

该图 SW = 111

输出显示为第一条指令对应 rt 寄存器的高 16 位值，因为第一条指令为 I 型指令，其 rt 寄存器中并没有数，默认为 0

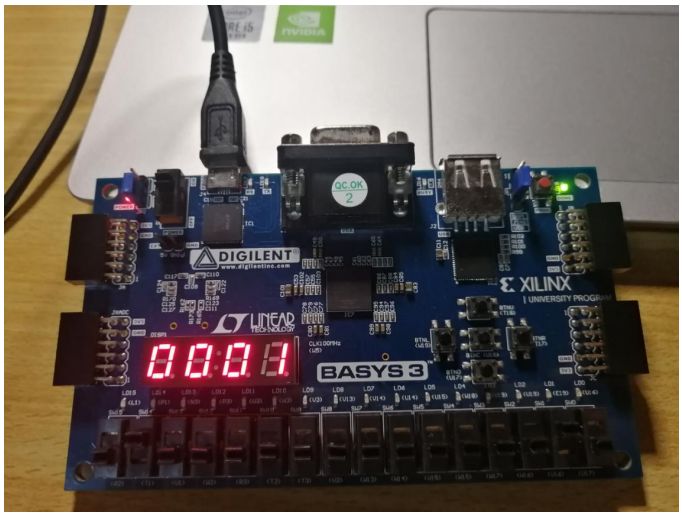


图 15

再次拨动开关，进入下一个周期，该图显示的是第二条指令的执行结果的低 16 位，依照表格 aluRes 仍为 1

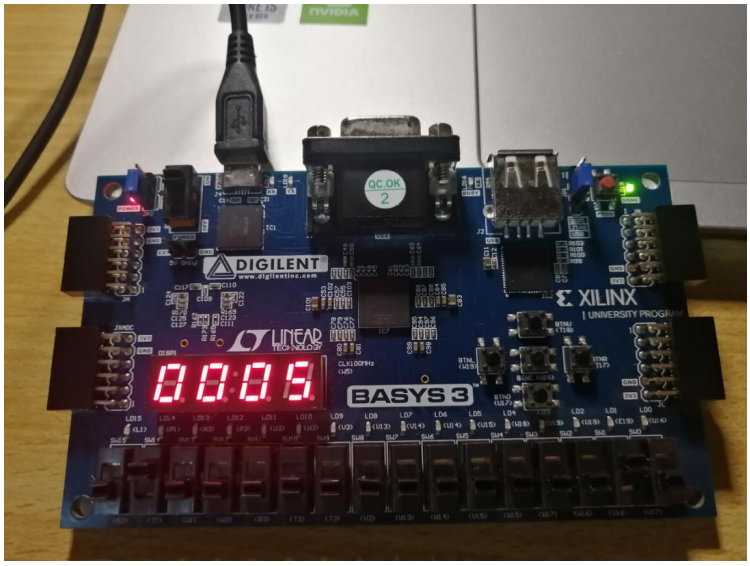


图 16

进入第四个周期，该图中 SW = 010，显示当下 aluRes 的结果，为 5，查看指令 or $t1,$s1,$s3，alu 运算结果应为 s1 和 s3 寄存器当中的值相或，即 4 和 1 相或，运算结果为 5，图示显示结果一直致，CPU 运行正确。

下面验证 beq 指令， 该指令对应 pc 值低位为 18，运行结果为零，要进行跳转，跳转至 next1 处，

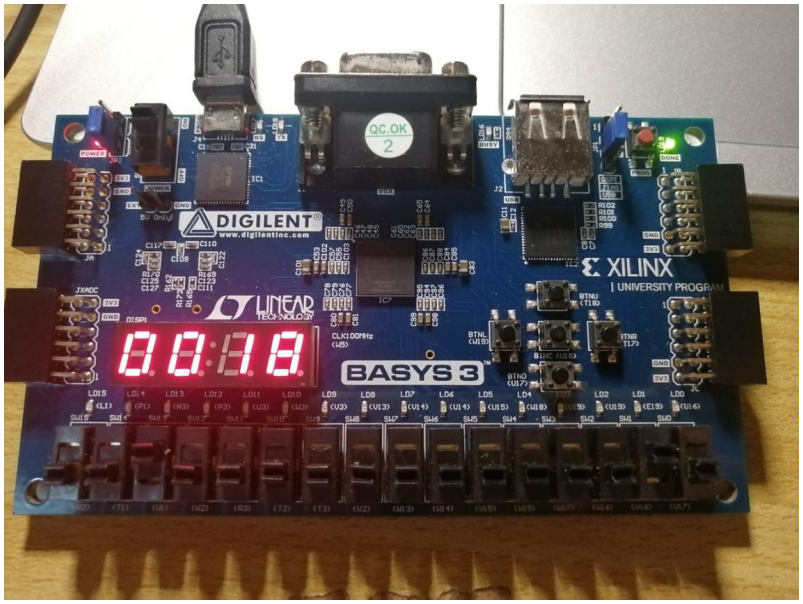


图 17

再拨动开关后

进行跳转，下一条指令对应的 pc 值为 20，原 pc 基础上加了 8

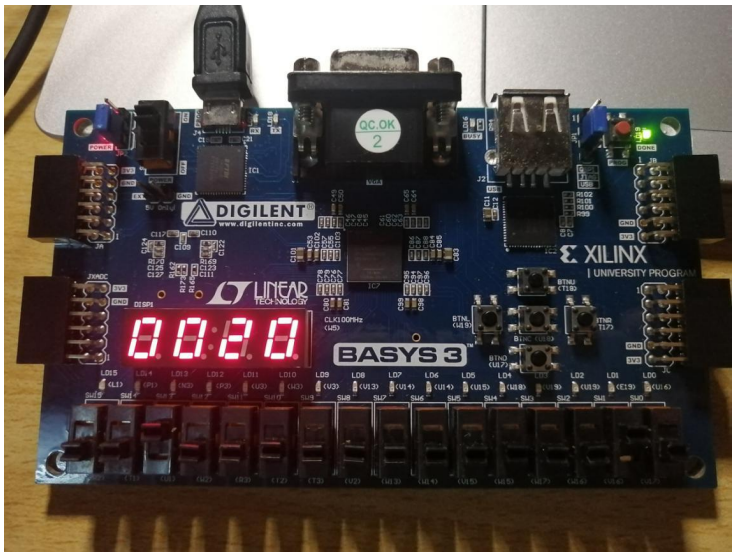


图 18

下面验证跳转指令 j

依照设计，j 指令对应 pc 为 24

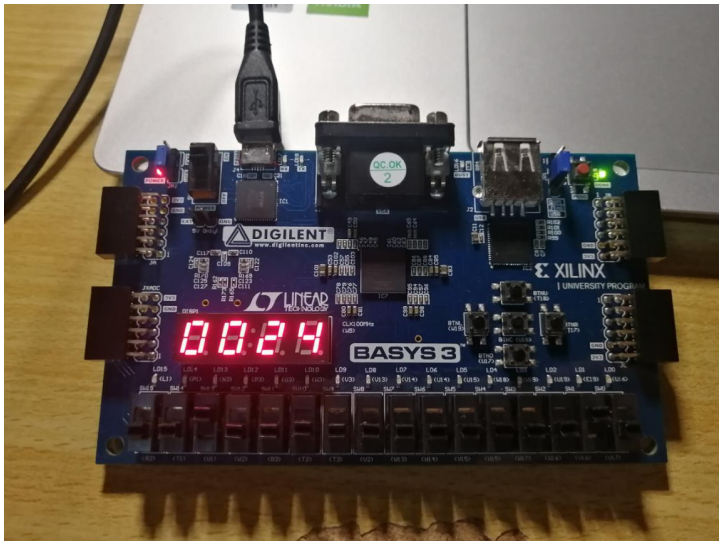


图 19

此时对应的指令十六进制的低 4 位为 000b

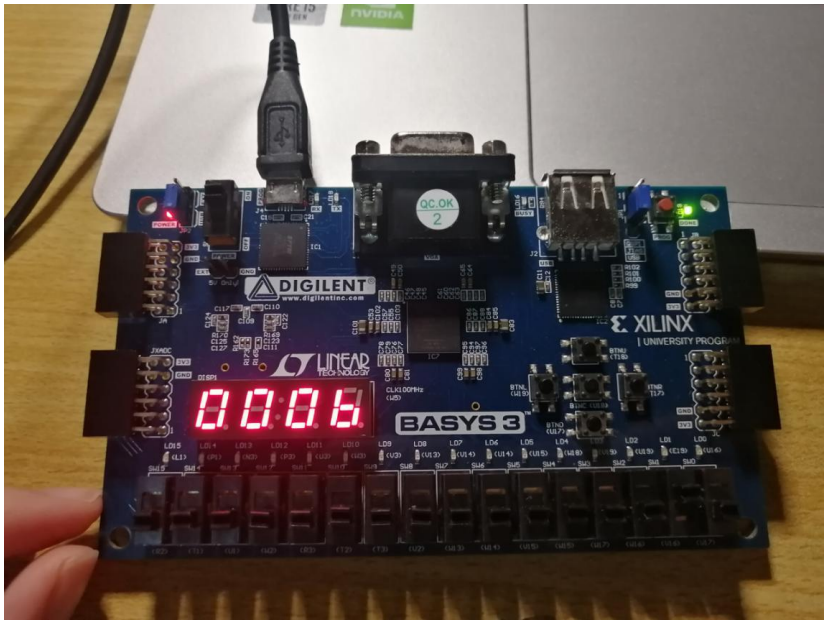


图 20

拨动开关，指令进行跳转，跳转至 next2 初，也即 pc 值为 2c 处，表格和烧板显示结果一致，证明 CPU 运行正确

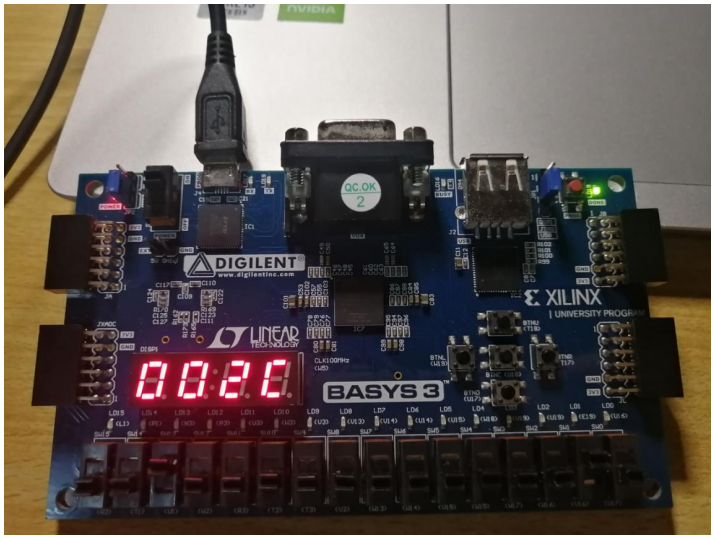


图 21

再查看此时的 aluRes ，其十六进制第四位为 22，对照表格验证其结果正确。

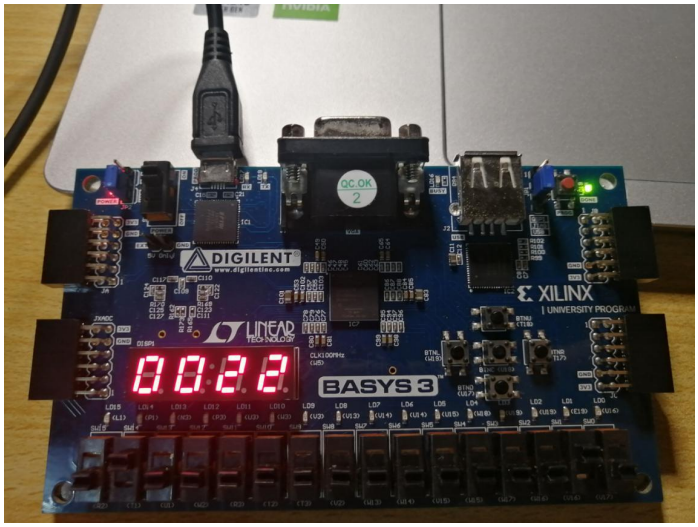
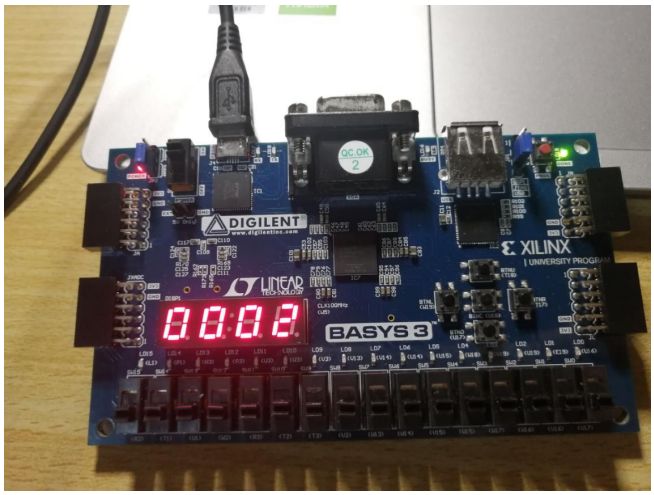


图 22

再看此时的十六进制的指令第四位，对应 next2 后的第一句指令 354a0002 的后四位，结果输出正确



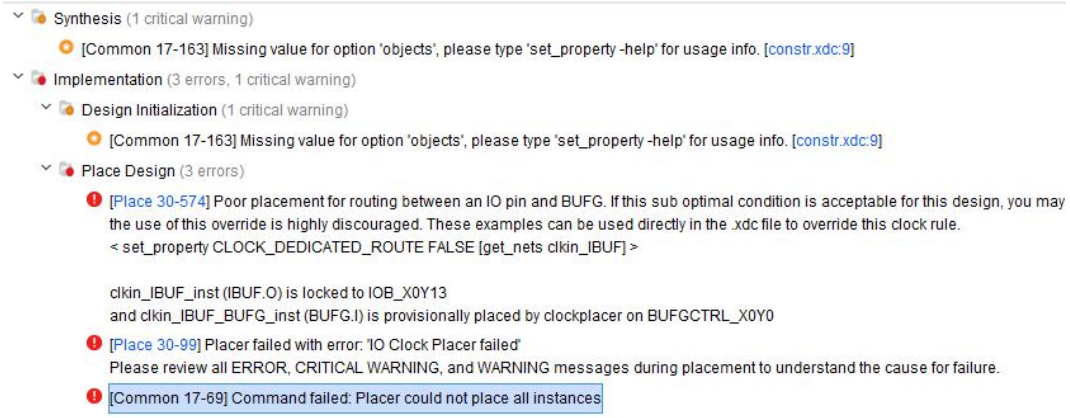
六. 实验心得

通过本次实验，我对单周期CPU的运行过程以及其各个模块的主要功能有了更深入的了解，掌握了不同类型指令的实现方法，如R型、I型以及跳转指令j、分支指令beq、bne等。

在实验过程中，我花费了大量的时间在处理“pc值在何时变换更新以及它更新的值是什么？”这一块的问题，最后理清楚了其变化规律，用了除pc之外的两个参数PC4（表示当前pc值+4的结果）和npc（表示下一个pc的值）来进行分情况讨论赋值，解决了这个问题。

最后在烧板的时候也遇到了很多问题，诸如七段数码管全亮但是每段亮的强度不同，进行分频改进后又遇到了一次只亮一位灯的情况，对代码修正后显示终于正常，但又发现烤到板子上之后实际操作情况和仿真波形的显示结果不一样：第一条指令对应pc值为4，实际上应该是0，每次遇到跳转指令时，都会在进行跳转指令的下一条再跳转。询问同学之后，发现是因为指令延迟了1个时钟周期才被读到。解决方法是，在指令存储器中将output的指令改为wire型，这样指令直接传出，没有延迟，解决了最后一个问题。

当然还有许多小问题，诸如约束文件报错，其实它已经给出了解决办法



还有最终的顶层文件output定义太多，而没有给他们分配引脚形成的错误，这时把所有最终和板子连接的input和output定义在module后的括号里即可，其他放到外面，用wire或reg进行定义。

这次CPU实验花费了时间很长，出现了很多问题，每次总觉得离成功只有一步之遥的时候又出现了新的问题，但是通过不断地和同学讨论、向同学请教、以及上网搜索，终于成功实现，成就感还是蛮大的。