中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 计算机类 4 班
学 生 姓 名 ：	杨志成
学　　号 ：	16337281
时　　间 ：	2017　年　11　月　20　日

成 绩 ：

实验二 ： 单周期CPU设计与实现

一. 实验目的

(1) 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
(2) 掌握单周期 CPU 的实现方法，代码实现方法；
(3) 认识和掌握指令与 CPU 的关系；
(4) 掌握测试单周期 CPU 的方法；
(5) 掌握单周期 CPU 的实现方法。

二. 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

==> 算术运算指令

（1）**add rd , rs, rt** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs + rt。**reserved 为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate；immediate** 做“0”扩展再参加“或”运算。

（5）and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（7）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

==>比较指令

（8） slt rd, rs, rt 带符号数

011100	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：if (rs<rt) rd =1 else rd=0，具体请看表 2 ALU 运算功能表，带符号

==> 存储器读/写指令

（9）sw rt ,**immediate**(rs) 写存储器

100110	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**]←rt；**immediate** 符号扩展再相加。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（10) lw rt , **immediate**(rs) 读存储器

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt ← memory[rs + (sign-extend)**immediate**]；**immediate** 符号扩展再相加。

即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==> 分支指令

（11）beq rs,rt,**immediate**

110000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs=rt) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。immediate** 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 **immediate** 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（12）bne rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

（13）bgtz rs,**immediate**

110010	rs(5 位)	00000	**immediate**

功能：if(rs>0) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4

==>跳转指令

（14）j addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],0,0}，无条件跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均

为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址了，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（15）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三. 实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（**如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。**）

CPU 在处理指令时，一般需要经过以下几个步骤：

(1) 取指令(**IF**)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

(2) 指令译码(**ID**)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(**EXE**)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(**MEM**)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(**WB**)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

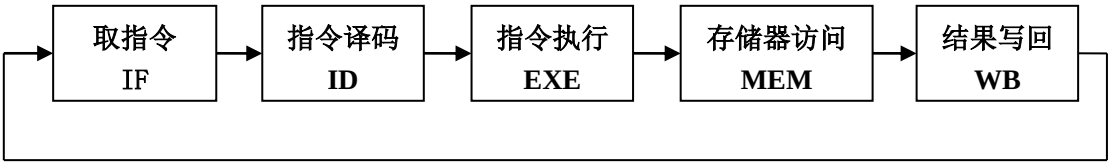
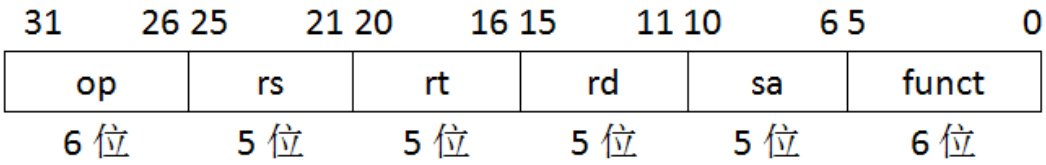


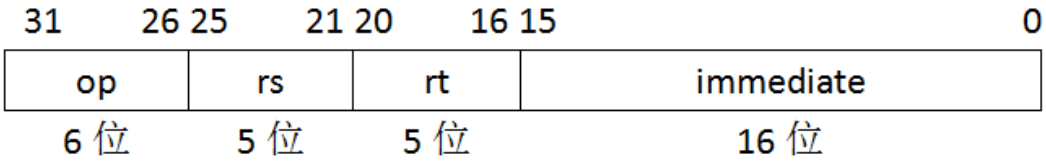
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

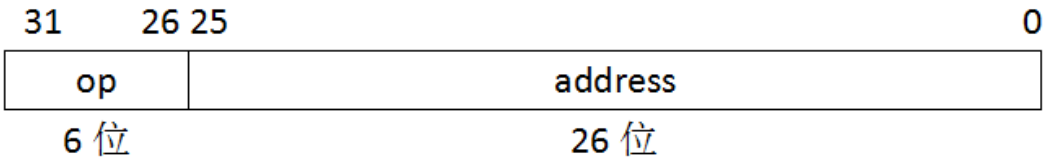
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

rs：只读。为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：可读可写。为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：只写。为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能与操作码配合使用；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Load）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

address：为地址。

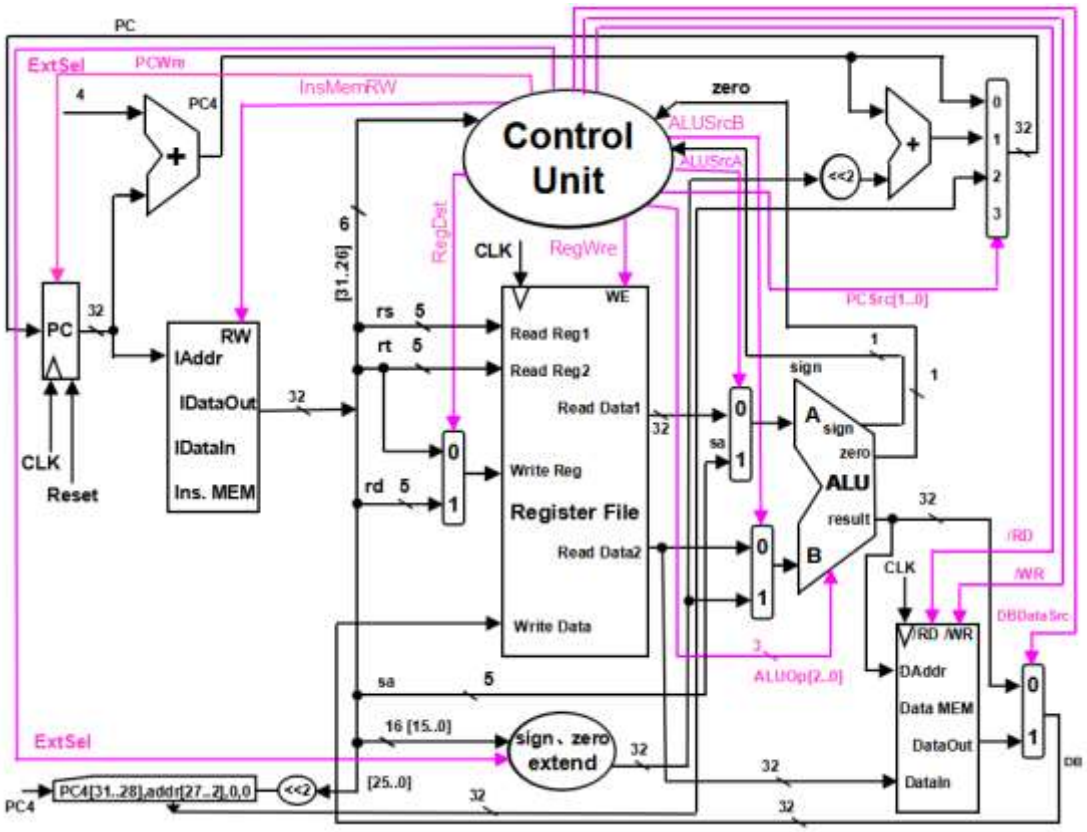


图 2　单周期 CPU 数据通路和控制线路图

图 2 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，即有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。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

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，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、bgtz、slt、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、slt、beq、bne、bgtz	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slt、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bgtz、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slt、sll、lw

InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器，相关指令：lw	输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、slt、sll
ExtSel	(zero-extend)**immediate(0 扩展)，**相关指令：ori	(sign-extend)**immediate (符号扩展)**，相关指令：addi、sw、lw、beq、bne、bgtz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1，或 zero=1); 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)、bgtz(sign=0，zero=0); 10：pc<－{(pc+4)[31:28],addr[27:2],0,0}，相关指令：j; 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：**指令存储器，**

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：**数据存储器，**

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

/RD，数据存储器读控制信号，为 0 读

/WR，数据存储器写控制信号，为 0 写

Register File：**寄存器组**

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU： 算术**逻辑单元**

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

sign，运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述

000	Y = A + B	加
001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号

110	if (A<B &&(A[31] == B[31]))
	Y = 1;
	else if (A[31] && !B[31] Y = 1;
	else Y = 0;
		比较 A 与 B
		带符号

111	Y = A ⊕ B	异或

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表，再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

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，PC 的改变是在时钟上升沿进行的，这样稳定性较好。

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设计思路:

由于我们要测试相应的指令，现在将相应的指令化成二进制，并给出每条指令对应的控制单元输出信号。

1、测试程序段

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	
0x00000000	addi $1,$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008
0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010		
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000		
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000		
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000		
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000		
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 000000		
0x0000001C	**bne $8,$1,-2 (≠,转 18)**	110001	01000	00001	**1111 1111 1111 1110**		
0x00000020	slt $6,$2,$1	011100	00010	00001	00110 0000 000000		
0x00000024	slt $7,$6,$0	011100	00110	00000	00111 00000 000000		
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000		
0x0000002C	**beq $7,$1,-2 (≠,转 28)**	110000	00111	00001	1111 1111 1111 1110		
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100		
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100		
0x00000038	**bgtz $9,1 (>0,转 40)**	110010	01001	00000	0000 0000 0000 0001		
0x0000003C	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000
0x00000040	**addi $9,$0,-1**	000001	00000	01001	1111 1111 1111 1111		
0x00000044	**j 0x00000038**	111000	00000	00000	0000 0000 0000 1110		
0x00000048							

2、控制信号与指令关系

Op		PCWre	ALUSrcA	ALUSrcB	RegWre	RegDst	IsMemRW	ExtSel	ALUOp	PCSrc	/RD	/WR	DBData
000001	Addi	1	0	1	1	0	1	1	000	00	1	1	0
010000	ori	1	0	1	1	0	1	0	011	00	1	1	0
000000	Add	1	0	0	1	1	1	0	000	00	1	1	0
000010	Sub	1	0	0	1	1	1	0	001	00	1	1	0
010001	And	1	0	0	1	1	1	0	100	00	1	1	0
010010	Or	1	0	0	1	1	1	0	011	00	1	1	0
011000	Sll	1	1	0	1	1	1	0	010	00	1	1	0
110001	Bne	1	0	1	0	x	1	1	001	00/01	1	1	x
011100	Slt	1	0	0	1	1	1	0	110	00	1	1	0
110000	Beq	1	0	1	0	x	1	1	001	00/01	1	1	x
100110	Sw	1	0	1	0	x	1	1	000	00	1	0	X
100111	Lw	1	0	1	1	0	1	1	000	00	0	1	1
110010	Bgtz	1	0	0	0	x	1	1	110	00/01	1	1	x
111111	Halt	0	x	x	0	x	1	X	x	X	1	1	x
111000	J	1	x	x	0	x	1	x	x	10	1	1	x

此外根据 图2.单周期CPU设计控制总线 来进行模块化设计，分成的模块关系如下：

其中ROM为指令寄存器，RAM为数据存储器，RegFile为数据寄存器,此外还有控制单元，PC，jump，选择器等模块，最后在Top顶层模块中连接起来。

几大主要模块的代码：

ROM：

```
`timescale 1ns/10ps
module ROM (idataout, addr, InsMemRW);
    input InsMemRW;
    input [31:0] addr; //2^5条指令地址 32条指令读取
    output reg [31:0] idataout;
    reg [7:0] mem [0:127]; //最大读取32条指令
    initial begin
        $readmemb("C:/Users/DELL/Desktop/mips_cpu_32/rom_data.coe", mem);
          idataout = 0;
     end
    always @( addr or InsMemRW)
        if (InsMemRW) begin
          idataout[31:24] = mem[addr];
          idataout[23:16] = mem[addr+1];
          idataout[15:8] = mem[addr+2];
          idataout[7:0] = mem[addr+3];
        end
endmodule
```

注：ROM由control_unit单元的控制信号InsMemRW决定是读还是写

数据输出按8位赋值一次

ALU:

```
module ALU(A, B, ALUOp, zero,sign, result);
  input [31:0] A, B;
  input [2:0] ALUOp;
  output zero;
  output sign;
  output reg [31:0] result;
  initial begin
       result = 0;
  end
  assign zero = (result? 0 : 1);
  assign sign = (result>0)?1:0;
  always @(A or B or ALUOp) begin
    case(ALUOp)
      3'b000: result <= A + B;
      3'b001: result <= A - B;
      3'b010: result <= B << A;
      3'b011: result <= A | B;
      3'b100: result <= A & B;
      3'b101: result <=(A<B)?1:0;
      3'b110:  begin // 带符号比较
                if (A<B &&( A[31] == B[31]) )
                    result <= 1;
                else if ( A[31] == 1 && B[31]==0)
                     result <= 1;
                else result <= 0;
                end
      3'b111: result <= (A & ~B) | (~A & B);
      default: result <= 0;
    endcase
  end
endmodule
```

注：ALU的功能由ALUOp决定，该代码主要由case语句实现

ALU功能表在先前已经给出，ALUOp由控制单元生成

RAM:

```
`timescale 1ns /1ns
module RAM(
    input clk,
    input [31:0] address,
    input [31:0] writeData, // [31:24], [23:16], [15:8], [7:0]
    input nRD, // 为0，正常读；为1,输出高组态
    input nWR, // 为0，写；为1，无操作
    output [31:0] Dataout
    );
reg [7:0] ram [0:60]; // 存储器定义必须用reg类型
// 读
    assign Dataout[7:0] = (nRD==0)?ram[address + 3]:8'bz; // z 为高阻态
    assign Dataout[15:8] = (nRD==0)?ram[address + 2]:8'bz;
    assign Dataout[23:16] = (nRD==0)?ram[address + 1]:8'bz;
    assign Dataout[31:24] = (nRD==0)?ram[address ]:8'bz;
// 写
    always@( negedge clk ) begin // 用电平信号触发写存储器，个例
    if( nWR==0 ) begin
    ram[address] = writeData[31:24];
    ram[address+1] = writeData[23:16];
    ram[address+2] = writeData[15:8];
    ram[address+3] = writeData[7:0];
    end
    end
endmodule
```

注：数据寄存器在这次试验中，只有lw指令输出了寄存器的内容，其余情况全是高阻抗

2选1，数据选择：（32位和5位实现思路类似）

```
module dataSelect_5_Bit(A, B, Ctrl, S);
  input [4:0] A, B; //两个数据
  input Ctrl; //控制信号
  output [4:0] S;
  assign S = (Ctrl == 1'b0 ? A : B);
endmodule
```

核心代码：assign S = (Ctrl == 1'b0 ? A : B);

该代码控制了数据选择端的输出

Control_unit控制单元：

```
always@(decode or zero or sign) begin // 解决跳转相等时候跳转的
        case( decode )
         6'b000000:
         begin   //以下都是控制单元产生的控制信号
           PCWre = 1;
           ALUSrcA = 0;
           ALUSrcB = 0;
           RegWre = 1;
           RegDst = 1;
           IsMemRW = 1;
           ExtSel = 0;
           ALUOp = 000;
           PCSrc = 00;
           nRD = 1;
           nWR = 1;
           DBData = 0;
         end
```

该代码的核心思路就是根据输入的指令前6位，用case语句分配给各个输出信号相应的值（由于代码太长，只贴出一部分即可,全部的代码可在附录中找到）

extern符号扩展单元：

```
module signExtend(i_num, extSel, o_num);
  input wire[15:0] i_num;
  input wire extSel;
  output reg[31:0] o_num;
  initial begin
    o_num = 0;
  end
  always @(i_num or extSel) begin
    if (extSel) begin
      o_num <= {{16{i_num[15]}}, i_num[15:0]};
    end
    else begin
        o_num[15:0]<=i_num[15:0];
    end
  end
endmodule
```

注：扩展单元有两个功能:0扩展和1扩展

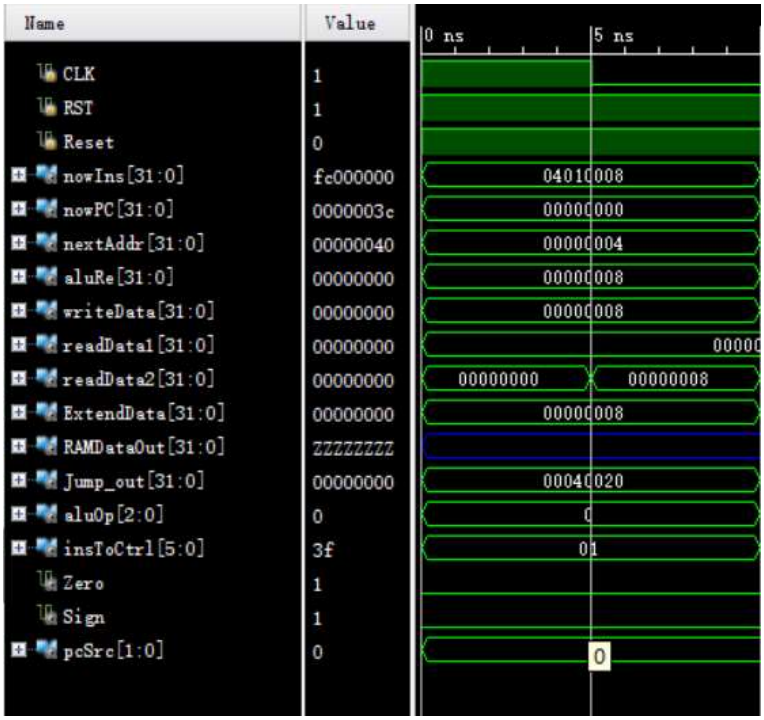
核心代码：o_num <= {{16{i_num[15]}}, i_num[15:0]};

o_num[15:0]<=i_num[15:0];

2.验证CPU正确性：

（1）addi　$1,$0,8

对应的波形图：



由图，alu运行结果为8 ，写入寄存器的值也为8.

这时CPU刚刚启动，数据寄存器中所有数据全为0，RST，Reset分别控制PC和REGfile的清零。由于cpu刚刚启动，所以pc为0H

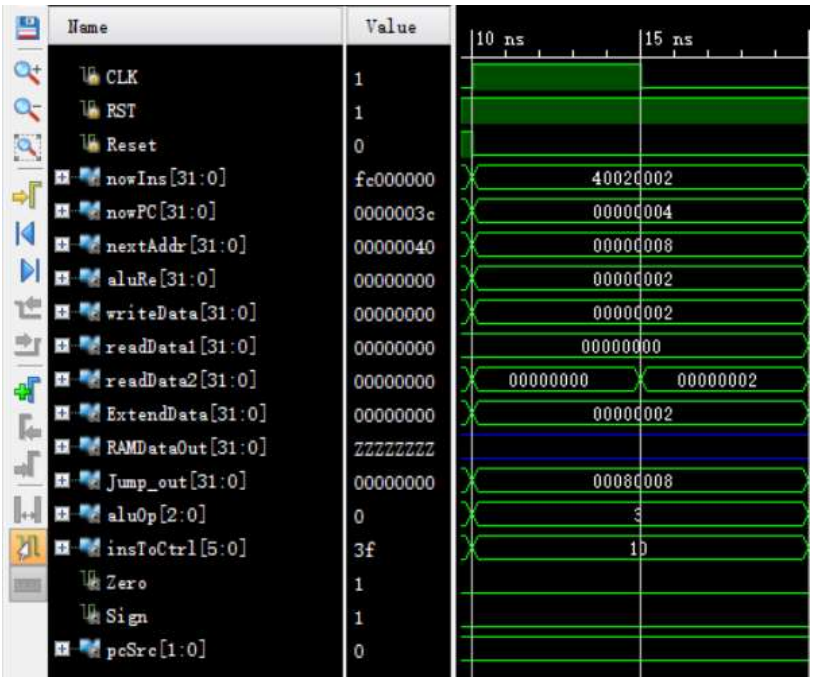
运行后寄存器数据：

$1 = 8

$0 = 0

（2）ori　$2,$0,2

波形图：



Ori 也为 R 型指令，运行结果正确

接下来我们把 R 型指令放在一起看他们的波形图

运行后寄存器数据：

$0 = 0;

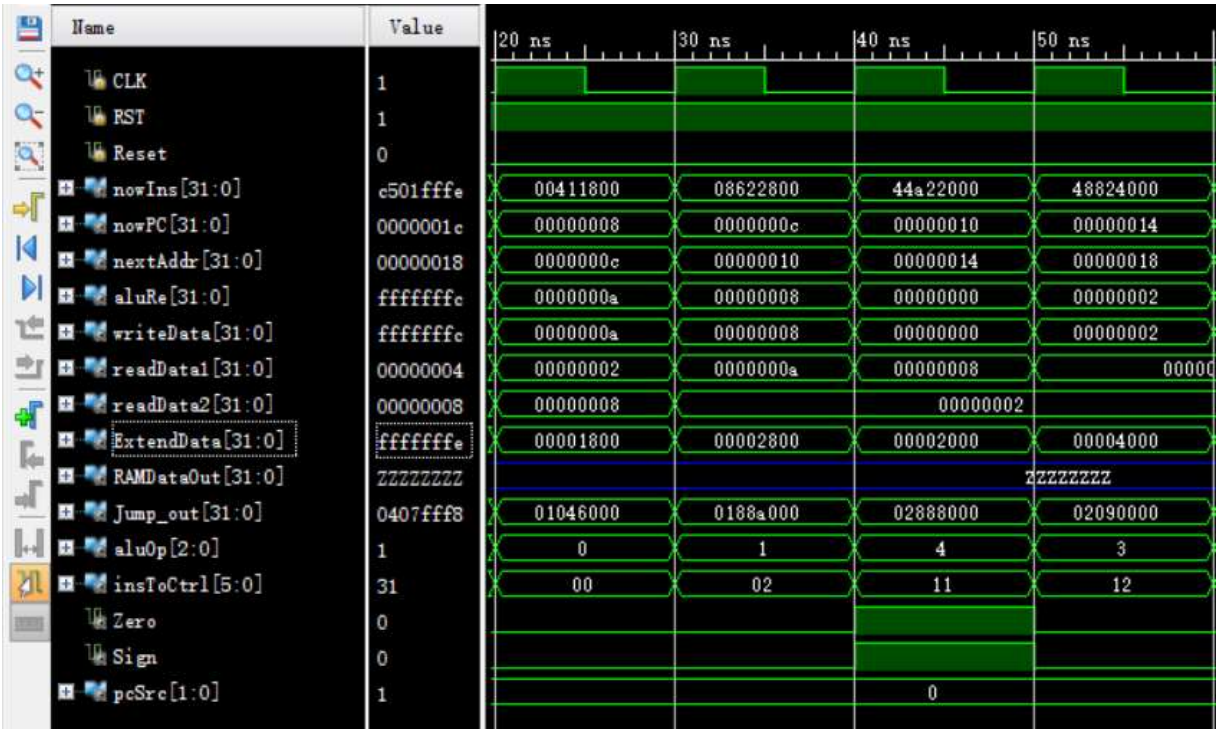
$1 = 8;

$2 = 2;

（3）

add	$3,$2,$1
sub	$5,$3,$2
and	$4,$5,$2
or	$8,$4,$2

波形图：



该四条指令运行结果如上图，

这时，PC为14H，nextPC为18H，读取寄存器的结果也符合我们代码运行结果

说明到此为止我们仍然是正确的

运行后寄存器数据：

$0 = 0

$1 = 8

$2 = 2

$3 = 10

$4 = 0

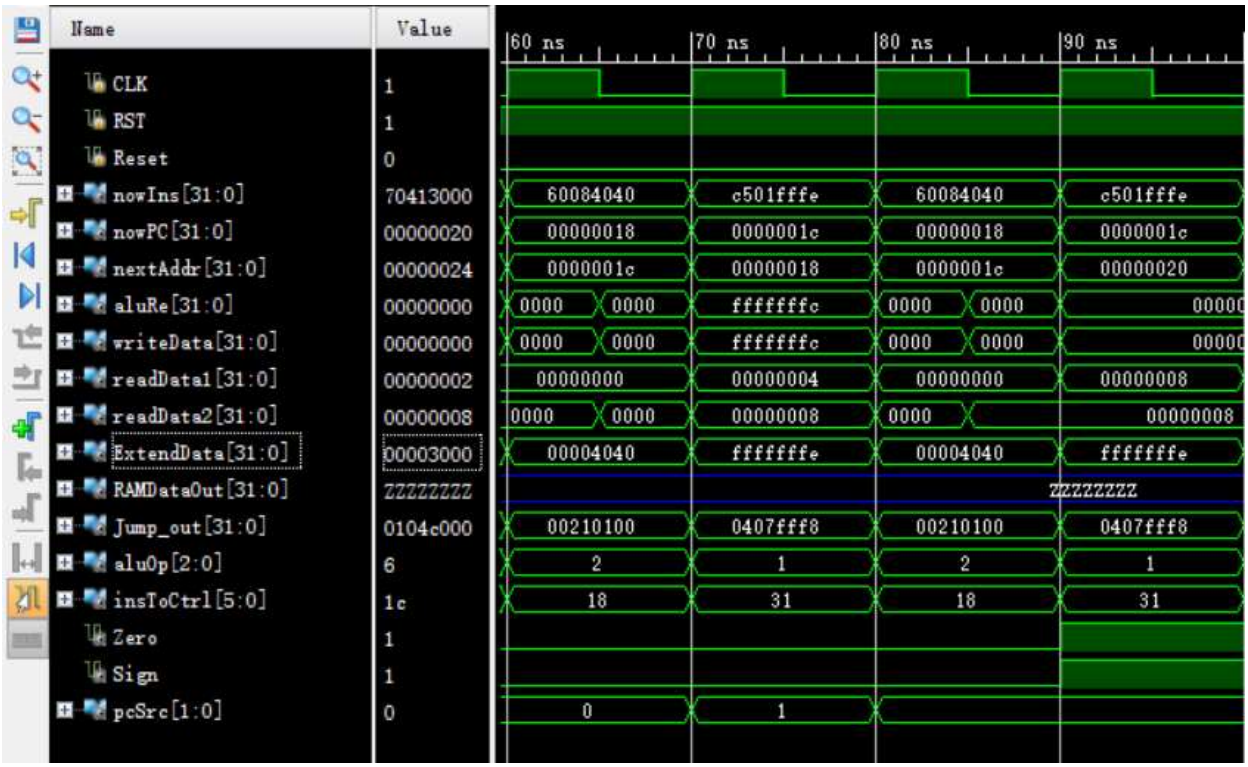
$5 = 8

$8 = 2

（4）

sll $8,$8,1
bne $8,$1,-2 (≠,转 18)

波形图：



指令执行前$8 = 2 ，$1 = 8

$8左移了两次才和$1相等，然后跳转

符合我们的预期结果，正确

运行后寄存器数据：

$0 = 0

$1 = 8

$2 = 2

$3 = 10

$4 = 0

$5 = 8

$8 = 8

(5)

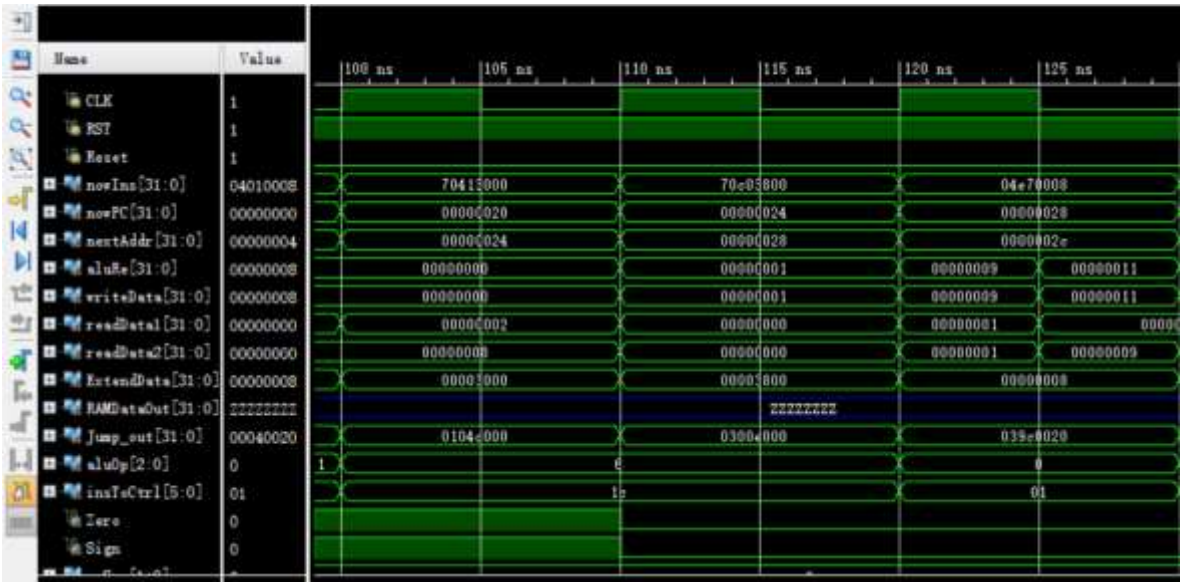
slt	$6,$2,$1
slt	$7,$6,$0
addi	$7,$7,8

波形图:

运行完这三条指令后，$7 = 0;

可见，我们的CPU到此处运行仍然正常。

指令运行后寄存器数据:



$0 = 0

$1 = 8

$2 = 2

$3 = 10

$4 = 0

$5 = 8

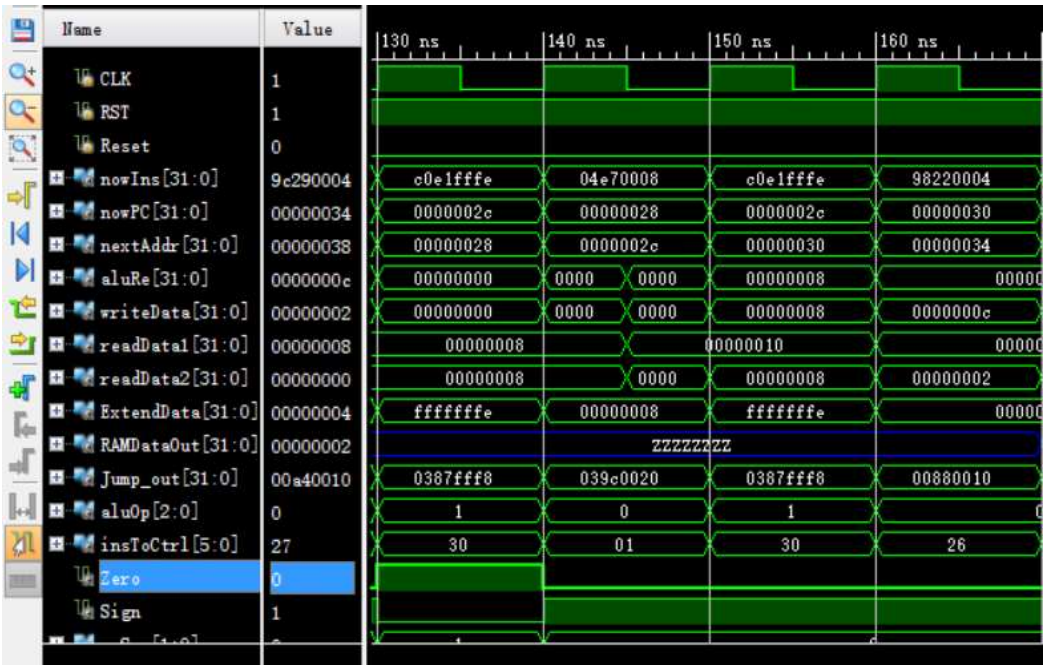
$6 = 0

$7 = 9;

$8 = 8

（6）**beq $7,$1,-2　(≠,转28)**

波形图：



Beq指令只有在两个寄存器值相同时才会跳转。

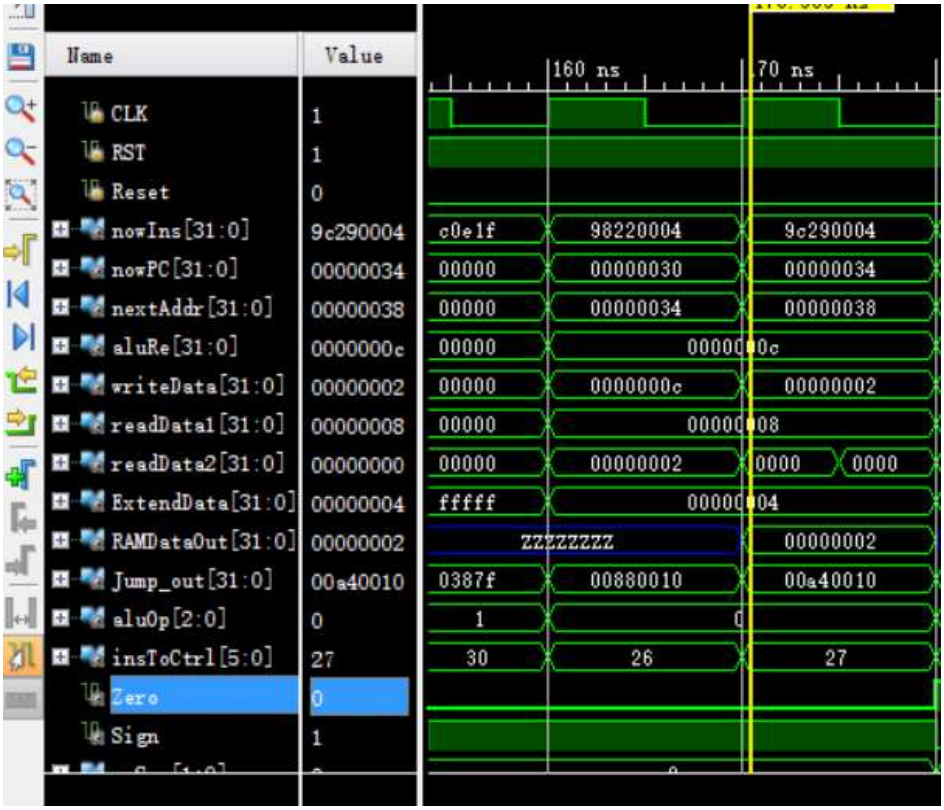
由于$7　=　8　与$1 = 8　相等，所以beq指令引起跳转

CPU运行正常

(7)

sw $2,4($1)
lw $9,4($1)

波形图：



注：由于只有lw指令运行时，数据存储器才输出了数据，所以除了这一次，其他情况下，数据存储器输出全为高阻抗。

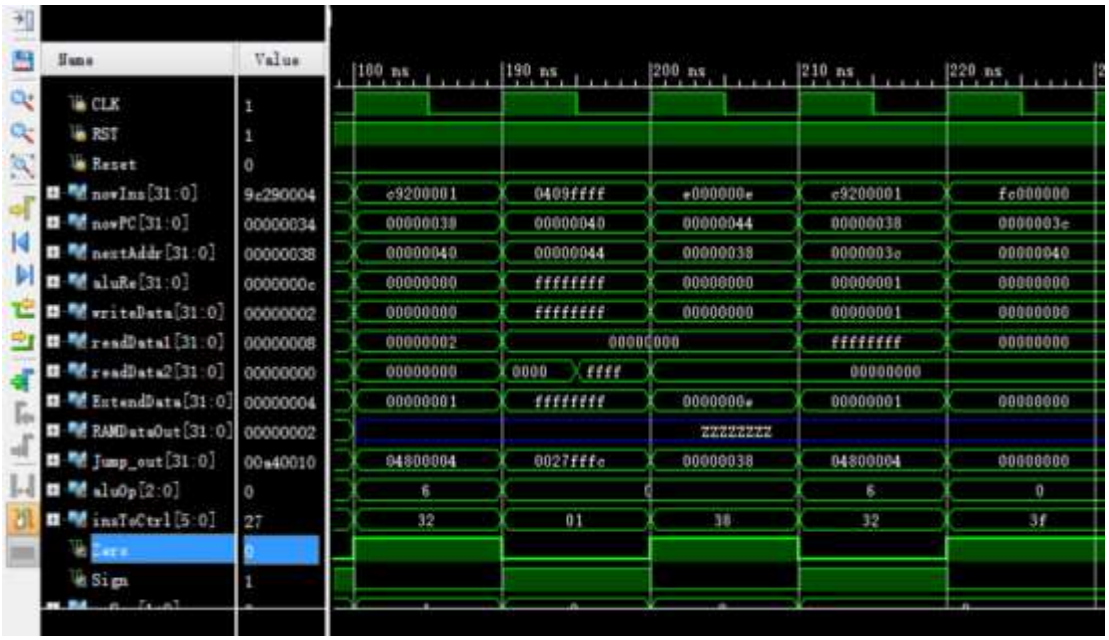
sw指令运行后，存储器12的值变为2

lw指令运行后 $9 = 2

(7)

bgtz $9,1 (>0,转40)
halt
addi $9,$0,-1
j 0x00000038

波形图:



指令运行前$9 = 2;
所以 PC 会在 j 指令和 bgtz 指令间跳转两次。
如图，pc 在跳转到 40H 后回到 38H,再经由 halt 指令停机
停机之后，pc 寄存器不再改变
综上所述，CPU 运行情况正确。
指令运行后，寄存器$9 = 0;

3. basys3 板的烧制

我给上述代码添加了一些新的模块，如按键消抖，数码管显示（具体代码请即见附录），其中用按键代替了时钟信号.具体实现情况如下：

开关说明：（以下数据都来自CPU）（SW15、SW14、SW0为Basys3板上开关名，BTNR为按键名）
开关SW_in (SW15、SW14)状态情况如下。显示格式：左边两位数码管BB ：右边两位数码管BB。以下是数码管的显示内容。
SW_in = 00：显示 当前 PC值:下条指令PC值
SW_in = 01：显示 RS寄存器地址:RS寄存器数据
SW_in = 10：显示 RT寄存器地址:RT寄存器数据
SW_in = 11：显示 ALU结果输出 :DB总线数据。
复位信号（reset）接开关SW0，按键（单脉冲）接按键BTNR。

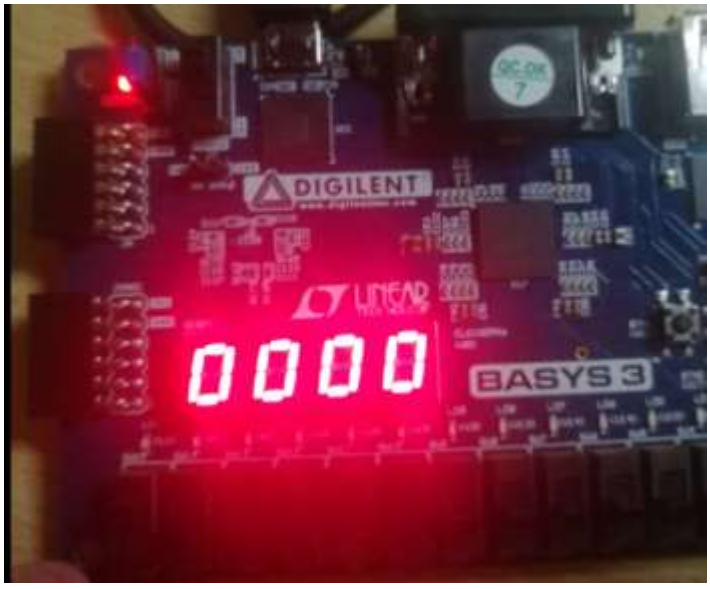
第一条指令（CPU 启动）

addi $1,$0,8

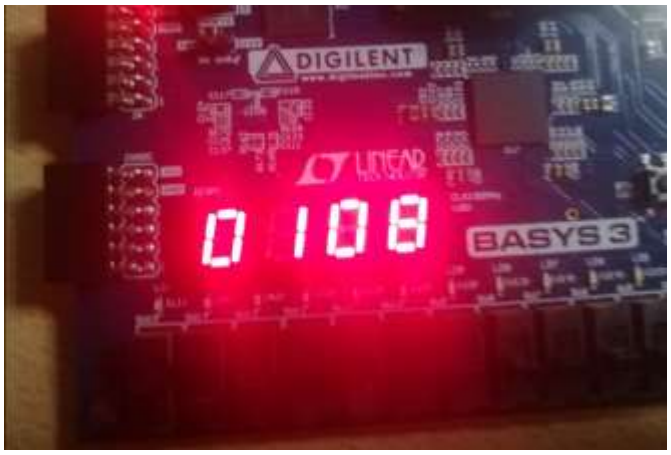
Sw_in = 00:

Sw_in = 01

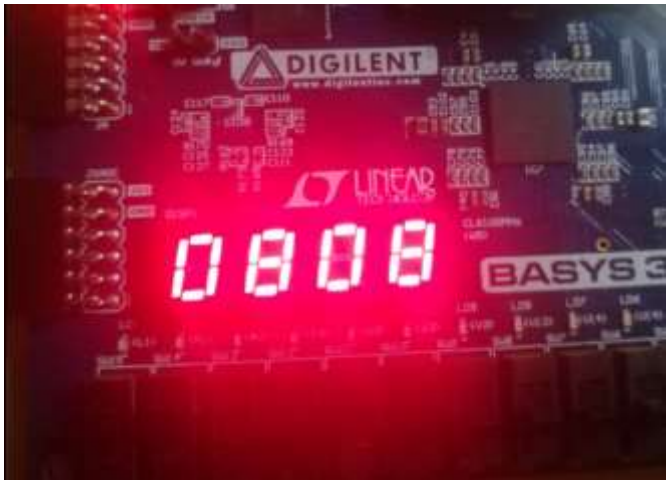
:



Sw_in = 10:



Sw_in = 11:



alu运行结果为8 ，写入寄存器的值也为8.

这时CPU刚刚启动，数据寄存器中所有数据全为0，RST，Reset分别控制PC和REGfile的清零。由于cpu刚刚启动，所以pc为0H

运行后寄存器数据：

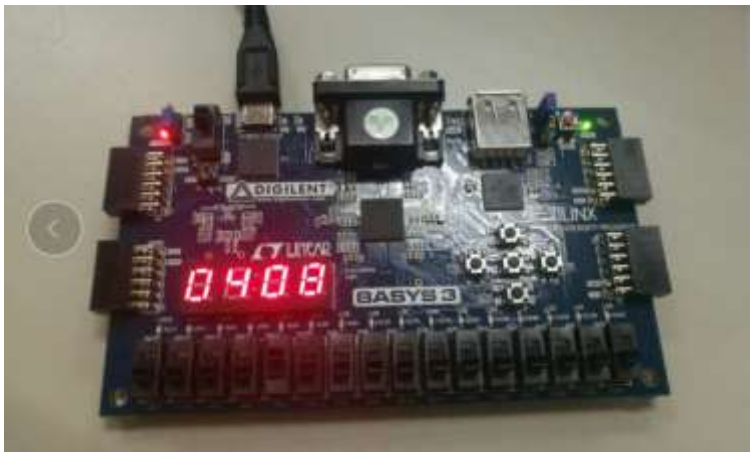
$1 = 8

$0 = 0

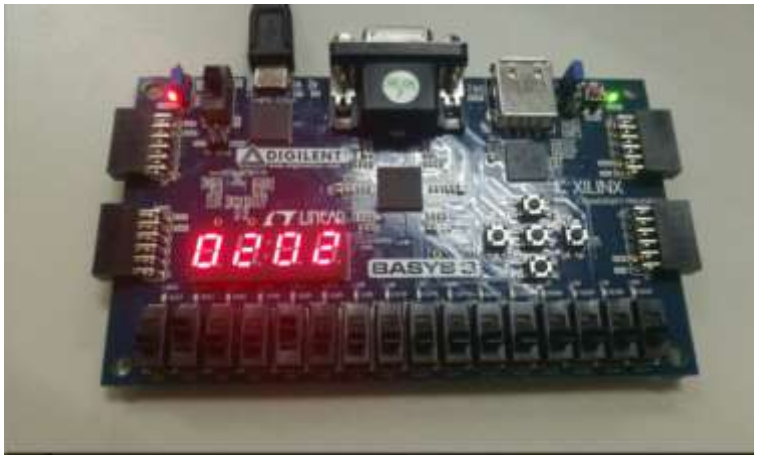
第二条指令：

ori　$2,$0,2

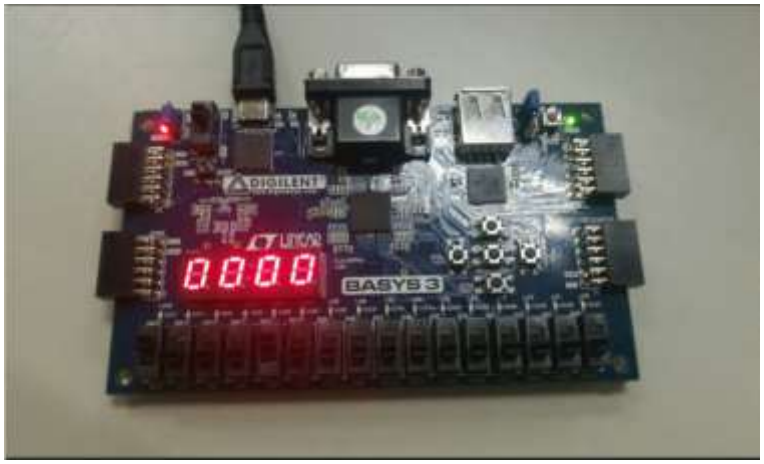
sw_00：



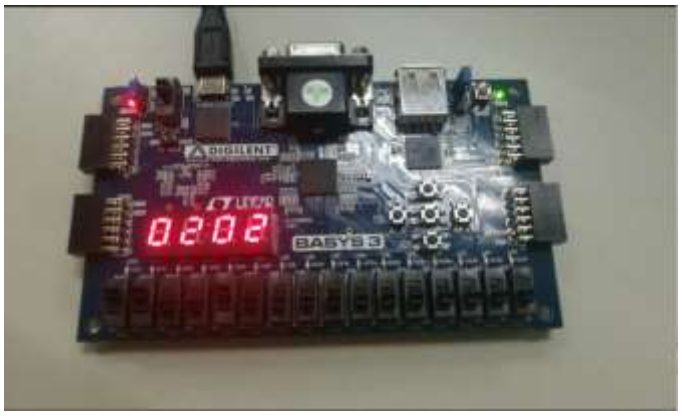
sw_01：



sw_10：



sw_11:



Ori 也为 R 型指令，运行结果正确

运行后寄存器数据:

$0 = 0;

$1 = 8;

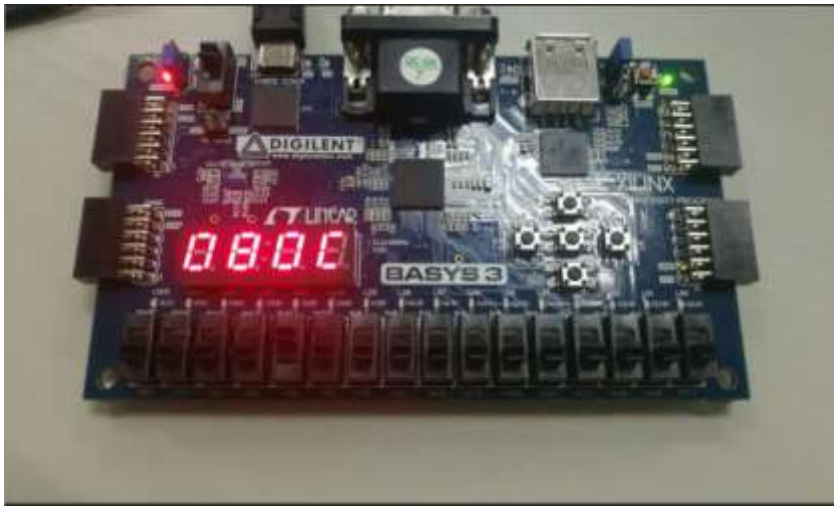
$2 = 2;

Basys 板正确

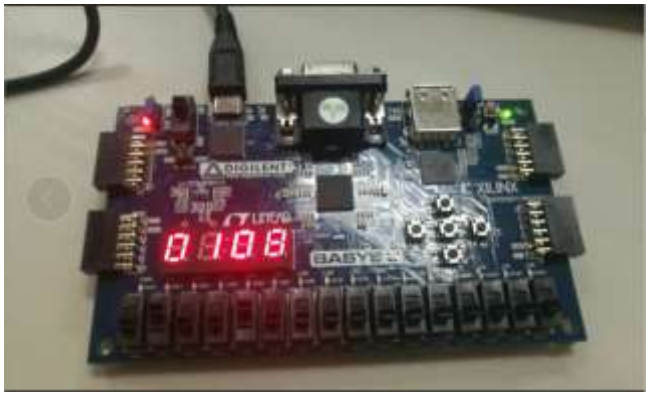
第三条指令：

add $3,$2,$1

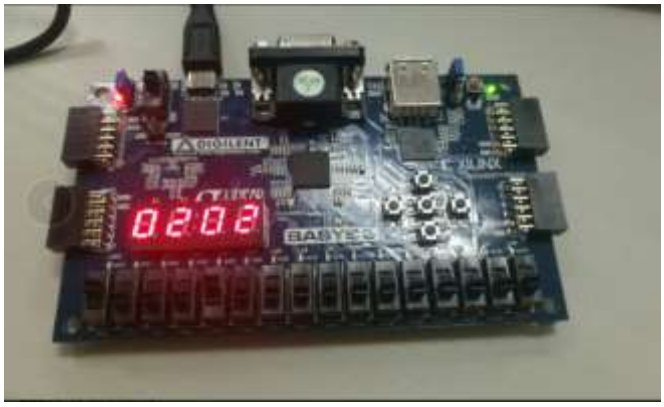
sw_00:



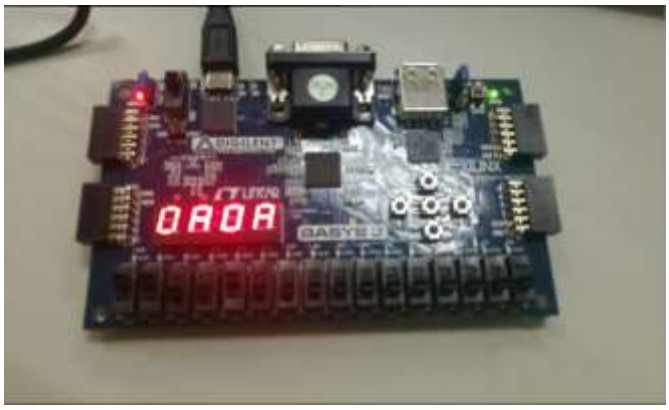
sw_01:



sw_10:



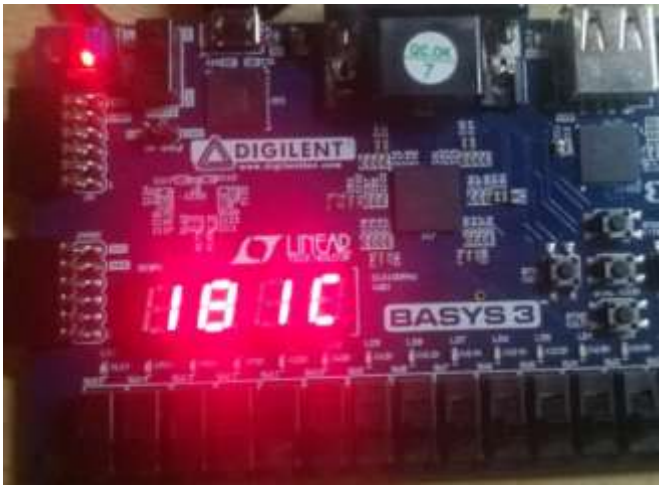
sw_11:



该指令也为 R 型指令，他将寄存器 1 和 2 的值相加赋给寄存器 3
则现在寄存器 3 的值为 10；
Basys 板显示正确
综上连续三条指令显示正确

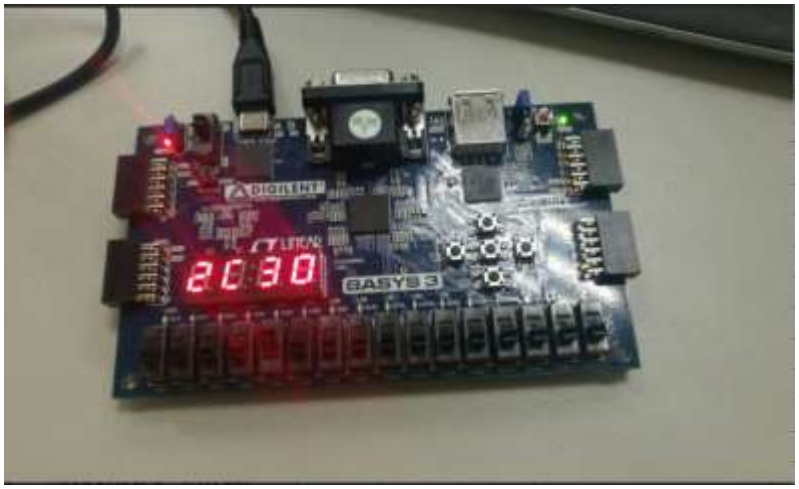
此外还有一些情况我也将它照了下来

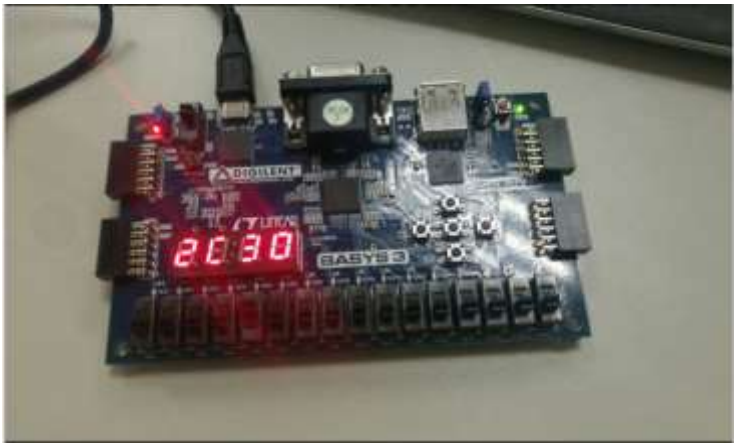
遇见跳转bne指令时：



可以看出他们在 18H 和 1cH 的地址跳转了一次。
这符合我们波形图的情况

在遇见 beq 指令时：





也有发生跳转的情况，符合我们的波形图
综上所述，basys 板烧制成功

六. 实验心得

通过本次实验，掌握了单周期cpu数据通路图的构成、原理及其设计方法，利用vivado将每一个模块实现了一遍，对于每一个模块的实现和运作有了更深的理解，掌握了对于cpu的检测方法，熟悉了从cpu的组合到测试到结果的每一个过程。

在实验过程中，遇到了在always@（）中括号里的值的问题，数据冒险。对于这个问题，认真的查找了网上的资料，并通过不断的更改其中的值做尝试，解决了问题。

虽然中途出现了许多bug，但是在自己的努力检查和同学的帮助下，所有的bug都顺利的被解决了。CPU的实现是一个稍微繁杂点的工程，需要我们用耐心去实现。

在这次的作业中学到了很多东西，在每次遇到不懂的问题时，都会各种百度，然后和同学一起研究，在解决问题的过程中对指令，和CPU结构有了更进一步的了解，同时，也熟练了汇编语言的相关操作。作为一名初学者，有不懂的东西就要多琢磨，多研究，多百度，只有这样才能学好这门课程，才能真正掌握计算机组成原理的相关知识。

从veriolog代码中获得的经验：

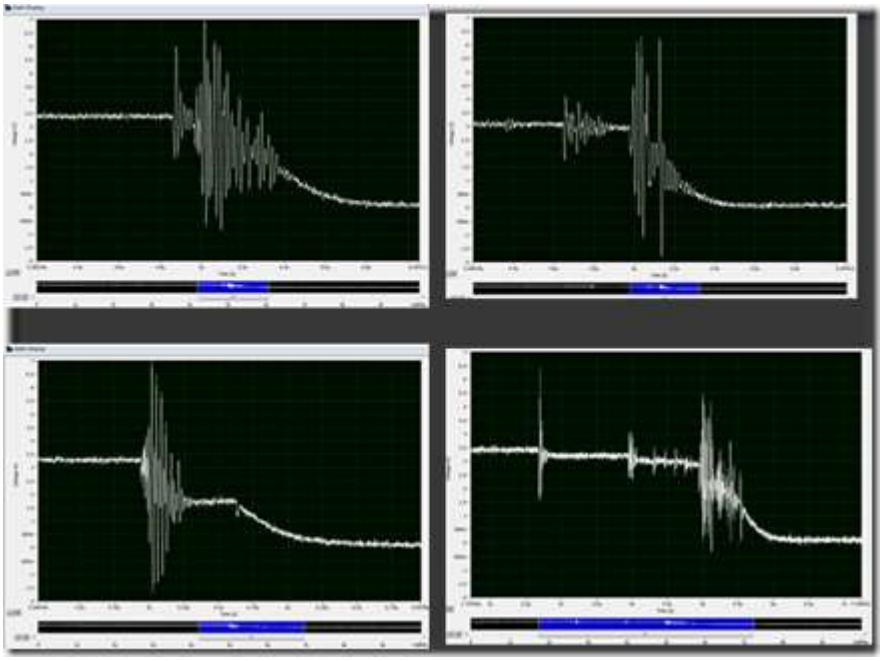
就拿按键消抖来说，按键开关是各种电子设备不可或缺的人机接口。在实际应用中，很大一部分的按键是机械按键。在机械按键的触点闭合和断开时，都会产生抖动，为了保证系

统能正确识别按键的开关，就必须对按键的抖动进行处理。

下面的四张图都是按键在闭合的时候抓到的波形。可以看到两个明显的趋势：

1．按键在几个us之内就可以达到稳定状态，从高电平转换到底电平；

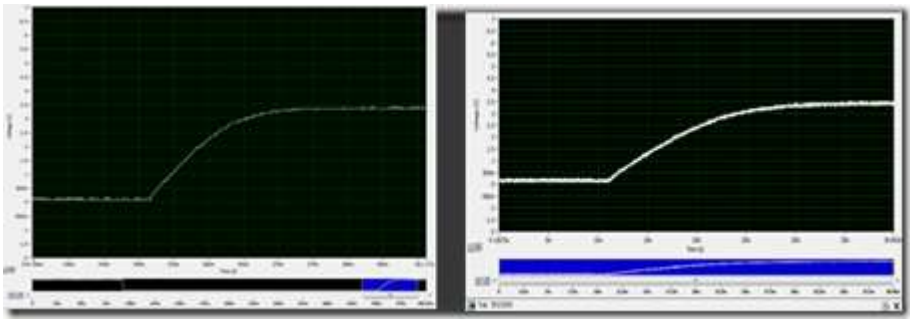
2．在高电平转换到低电平的过程中，触点有非常明显的抖动。



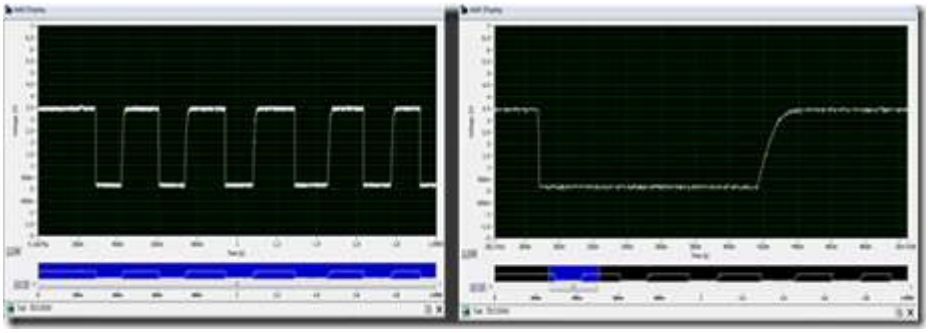
下面的两张图是按键在断开的时候抓到的波形。也可以看到两个明显的趋势：

1．按键的变化趋势比较缓慢，从低电平变为高电平需要大概10~20ms的时间；

2．按键断开时没有闭合时那么大的抖动



下面两张图是用手迅速闭合按键然后就断开时，按键的输出波形。



最后用代码实现：

```
1  `timescale 1ns / 1ps
2  module Light( in_key, out_key, clk);
3
4   input in_key, clk;
5   output out_key;
6   reg delay1, delay2, delay3;
7   always@( posedge clk)//CLK 50M
8      begin
9          delay1  <= in_key;
10         delay2  <= delay1;
11         delay3  <= delay2;
12     end
13 assign out_key = delay1&delay2&delay3;
14 endmodule
```

因为只有当连续三个时钟周期按键电平都未改变的情况下，才可以认为此时按键稳定，这一个思想比较巧妙。

收获：对于vivado及verilog语言的使用更加熟悉，对于单周期cpu设计，原理等都比只在理论课上听来的更加清晰明白。

感想：CPU十分精巧复杂，任何一个环节出错，带来的结果是灾难性的，这次试验当中我遇到了不少的问题，都是一些细微的错误，比如在顶层模块中一不小心把线连错，等等。错误看似不小，但是对结果所造成的影响，确很大。这次试验更教会我，在今后的学习生活中，我应该学会更加仔细，减少错误的发生，避免不必要的麻烦。