中山大學
SUN YAT-SEN UNIVERSITY

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

（3）subi rt, rs, **immediate**

000011	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

==> 逻辑运算指令

（5）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。

（6）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。

（7）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

（8）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

（9）slti　rt, rs,immediate　　带符号

011011	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

==> 存储器读/写指令

（10）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。

（11）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

（12）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”。

（13）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。

==>跳转指令

（14）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],2{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

简述实验原理和方法，**必须有数据通路图及相关图**。

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。

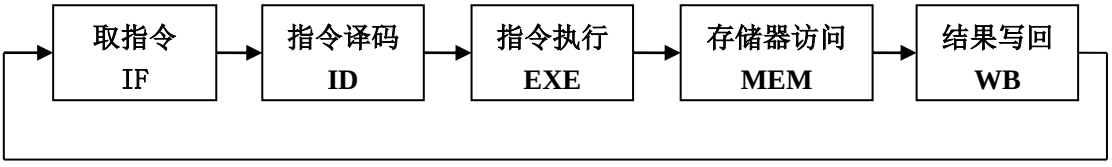
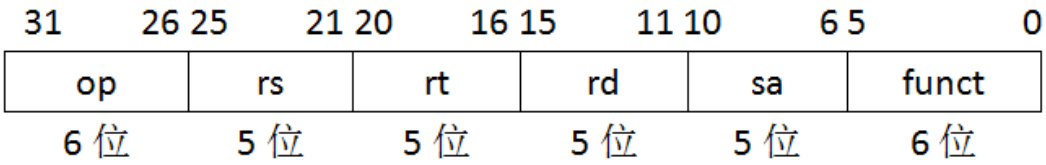


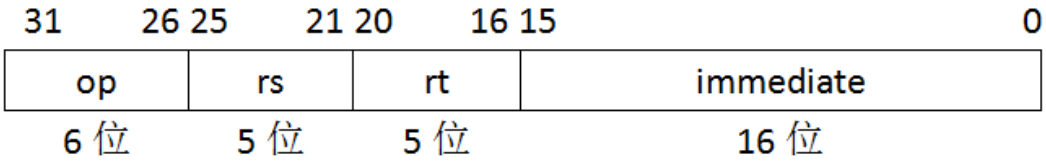
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

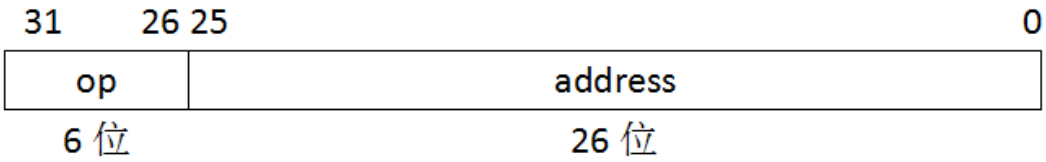
R 类型：



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。

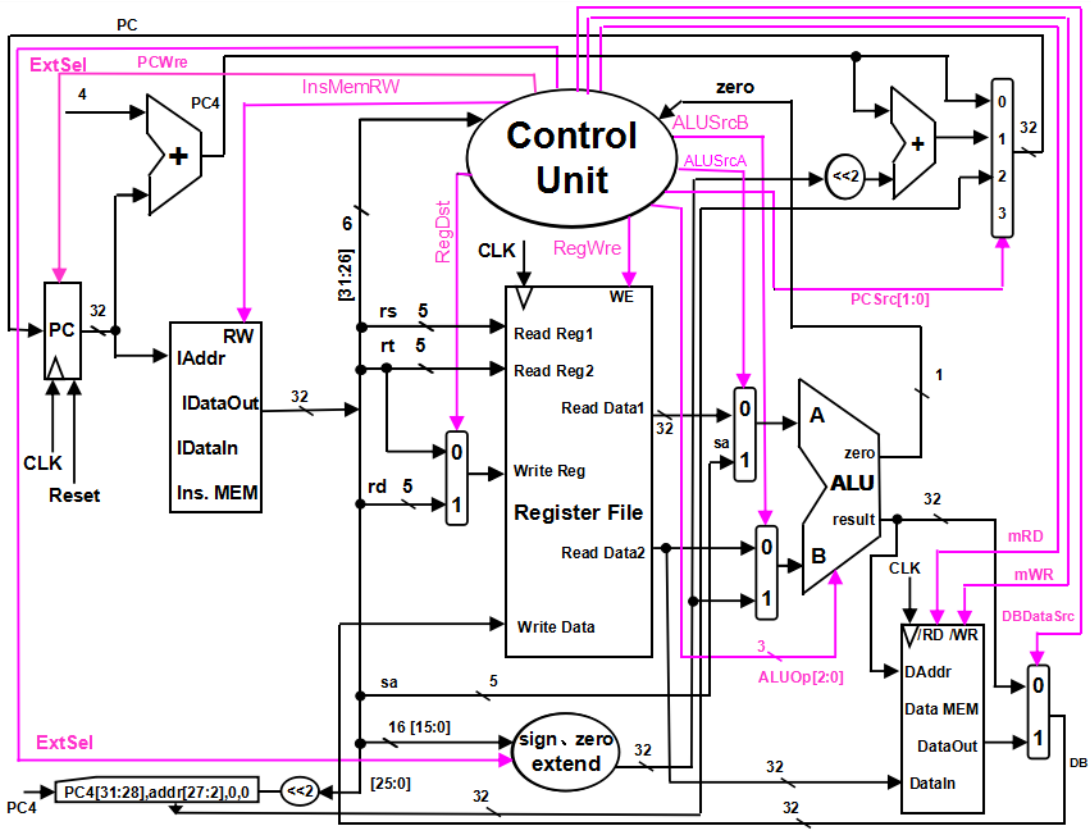


图 2　单周期 CPU 数据通路和控制线路图

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、slti、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、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、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、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slti、sll、

		lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw、slti	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、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、slti、sw、lw、beq、bne
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)； 10：pc<－{(pc+4)[31:28],addr[27:2],2{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

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	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	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（留给学生完成），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。

四. 实验器材

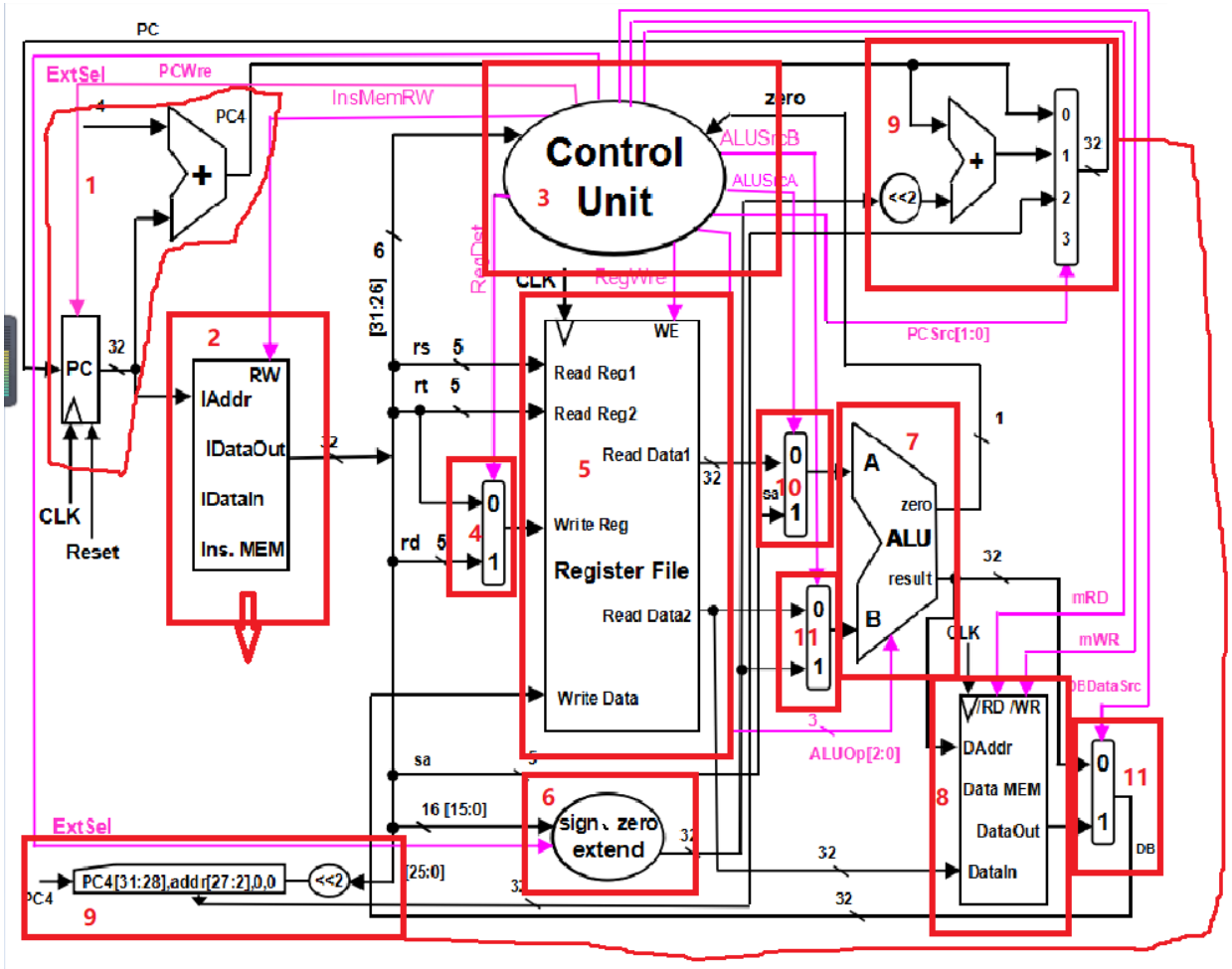
PC 机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

五. 实验过程与结果

实验设计：

首先分析CPU数据通路图，首先在平常学习中，我们已经知道CPU可以分为一些主要部分了，如程序计数器PC,指令存储器，寄存器，算术逻辑运算单元，内存。这些模块都是比较熟悉的。所以编写时也可以用模块化的思想编写。通过对数据通路的学习，理解数据如何在各个模块之间传递，所以只要确定模块的功能，输入与输出信号，就可以进行模块的编写。最后编写一个顶层模块，将所有子模块联系起来，所有的信号也就连通了

以下是具体模块的分割：

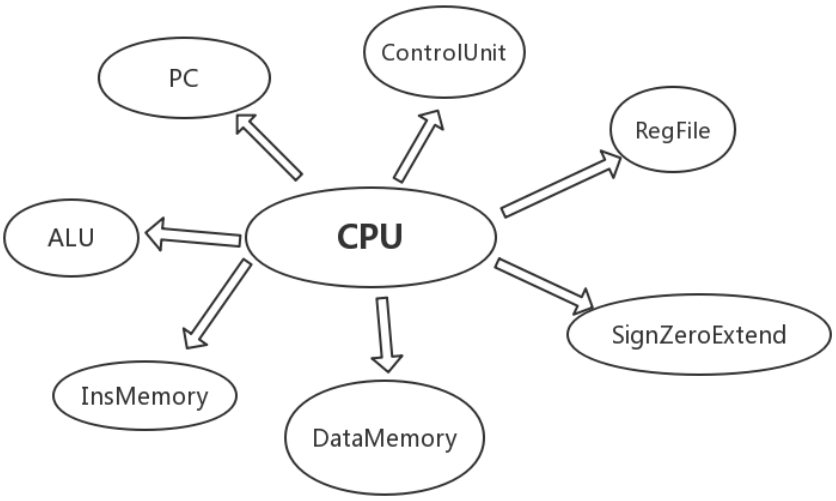


模块说明：

编号	名称	作用
1	PC 程序计数器	以时钟信号 clk、重置标志Reset、立即数以及PcIn和PCSrc两个信号控制为输入，输出当前 PC 地址和 PC+4
2	InsMemory指令存储单元	根据当前的PC地址得到对应的指令
3	ControlUnit 控制单元	接收一个6位的操作码（opCode）和一个标志符（zero）作为输入，输出PCWre、ALUSrcB等控制信号
4	SelectRtRd 二选一	根据控制信号RegDst选择数据源
5	RegFile 寄存器文件单元	接收 InsMemory 中的 rs，rt，rd作为输入，输出对应寄存器的数据，
6	SignZeroExtend 扩展单元	将一个16位的立即数扩展到32位

7	ALU 算术运算单元	接收寄存器的数据和控制信号作为输入，将结果输出
8	DataMemory 数据存储单元	根据地址得到内存中的数据，或将数据写入内存
9	Mux3 PC三选一	选择下一条指令的地址
10	MuxALUa 二选一	选择ALU操作数a
11	MuxFor32Bits 二选一	针对32位数据的二选一，可用于选择ALU操作数b，或选择要写入寄存器的数

主要模块图：



具体模块实现：

控制单元：ControlUnit

输入：op 操作码

Zero 零信号

输出：RegWre ALUSrcA ALUSrcB InsMemRw PcSrc ExtSel RegDst PCWre mRD mWR DBDataSrc ALUOp 控制信号，具体的解释在实验原理中。

原本想根据控制信号的真值表化简，但是变量太多，比较麻烦，因此直接分析各控制信号在哪一条指令下为1，使用逻辑或连接起来就可以了。

具体思路：

1. 针对每条指令定义一个变量，标志当前指令是否为变量对应的指令，即：

```
reg addi,add,sub,ori,andx,orx,sll,slti,sw,lw,beq,bne,j,halt;
```

因or,add 是关键字，因此使用 orx,addx 代替。

2.使用 swtich 语句，根据操作吗修改标志变量的值，如：

```
case(op)
        6'b000000: //add
          add = 1;
        6'b000001: //addi
          addi = 1;
        6'b000010://sub
          sub= 1;
        6'b010000://ori
          ori = 1;
```

注意每次判断前都要变量都要重置为0

3.编写各控制信号对应的逻辑表达式，如：

```
assign  ALUSrcA = sll;
assign  ALUSrcB = addi || ori||slti||sw||lw;
assign  DBDataSrc = lw;
assign  RegWre = add||addi||sub||ori||orx||andx||slti||sll||lw;
```

PcSrc 和 ALUop 采用逐位赋值，根据ALU与指令的关系表（实验原理中）

```
1. assign ALUOp[0] = sub||orx||ori||beq||bne||slti;
2. assign ALUOp[1] = sll ||orx||ori;
3. assign ALUOp[2] = andx||slti;
```

ALU 算术运算单元

输入：ALUopcode 选择ALU的功能 rega regb 两个操作数

输出：result，zero 零信号，标志结果是否为0，sign

思路:使用swtich 语句，根据ALUopcode 的值，对rega 和 regb 执行不同的运算，得到不同的reslut

关键代码：

```
case (ALUopcode)
      3'b000 : result = rega + regb;
      3'b001 : result = rega - regb;
      3'b010 : result = regb << rega;
```

PC 程序计数器

输入：clk 时钟信号，reset 重置信号、pcIn 、PCWre

输出：pcOut 当前指令的地址、PC4 下一条指令的地址

初始化pcOut 为 0，PC4 为4

需要时钟上升沿触发，同时需要判断 reset 和 PCWre 是否有效

```
always @(posedge clk ) begin
   if (reset == 0) begin
     pcOut = 0;
     PC4 = 4;
   end
   else if (PCWre==1) begin
     pcOut = pcIn;
     PC4 = pcOut+4;
   end
end
```

RegFile 寄存器模块

关于输入与输出在实验原理中

首先定义32位的数组存储寄存器的值，将数组内的数据都初始化为0，数组下标对应寄存器号。

读数据，若是取0号寄存器直接为0,否则取数组对应下标（即寄存器号）的值

```
assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
```

写数据需要时钟触发，要判断RST (reset)、RegWre、WriteReg是否有效

```
always @ (negedge CLK or negedge RST) begin
    if (RST==0) begin
      for(i=1;i<32;i=i+1)
      regFile[i] = 0;
    end
    else if(RegWre == 1 && WriteReg != 0) begin
      regFile[WriteReg] = WriteData;
      end
  end
```

SignZeroExtend 立即数拓展

先判断 ExtSel，确定是符号拓展还是零拓展，若是零拓展，直接补0，若是符号拓展，需要判断最高位的值，补0或者1

```
assign extendResult =
ExtSel?(immediate[15]==0?{16'h0000,immediate}:{16'hffff,immediate}):{16'
h0000,immediate};
```

MUX3 选择下一条指令的地址

首先得到jumpAddress的地址：将指令的[25:0]部分左移两位，之后PC+4的高四位作为 JumpAddress 的高四位

即：

```
assign extendAddr[27:2]= jumpAddress;
assign extendAddr[31:28] =PC4[31:28] ;
assign extendAddr[1:0] = 2'b00;
```

extendAddr 就是最终得到的 jumpAddress

使用三目运算符，通过判断 PCSrc 的值选择下一条指令的地址，若PCSrc为0，选择PC+4，即为顺序执行若为，选择立即数表示的地址，对应的指令有bne,若为2，则选择 JumpAddress，对应指令为 j

```
assign result = PCSrc==0?PC4:(PCSrc==1?(extendResult<<2)+PC4: extendAddr);
```

MuxALUa 选择 ALU 操作数a

直接根据ALUSrcA信号选择数据，为0，选择从寄存器中读取的ReadData1，否则选择指令中sa部分,注意sa是5位的，需要拓展为32位

```
assign ALUinputA = ALUSrcA==0?ReadData1:{0,sa};
```

SelectRtRd 选择目的寄存器

根据 RegDst 选择目的寄存器应该是 rt 还是 rd

```
assign WriteReg = RegDst==0?rt:rd;
```

MuxFor32Bits 32位二选一

使用三目运算符直接选择

```
assign res = control==0?v0:v1;
```

CPU 顶层模块

定义所有子模块中用到输入输出变量，注意模块间的输入输出变量是有联系的，名字不同的可能代表的是同一变量，比如指令用变量IDataOut表示，rt,rd,rs 都是IDataOut的一部分，不用再另外定义。

```
//ins memory
  output wire [31:0] IDataOut,
//register file
  output wire [4:0]WriteReg,
  output wire [31:0]readData1,readData2, writeData,
```

实例化每个模块，并且将对应的变量按照顺序转递进去，如：

```
InsMemory insMemory(pcOut,InsMemRw,IDataOut);
RegFile
regFile(clk,reset,RegWre,IDataOut[25:21],IDataOut[20:16],WriteReg,writeD
ata,readData1,readData2);
```

至此，CPU的设计就完成了。

实验测试：

首先编写测试文件，实例化一个CPU模块，并且设置好时钟和reset信号

```
always begin
   #5  clk =~ clk;
end

initial begin
  clk = 1;
  reset = 1;
end
```

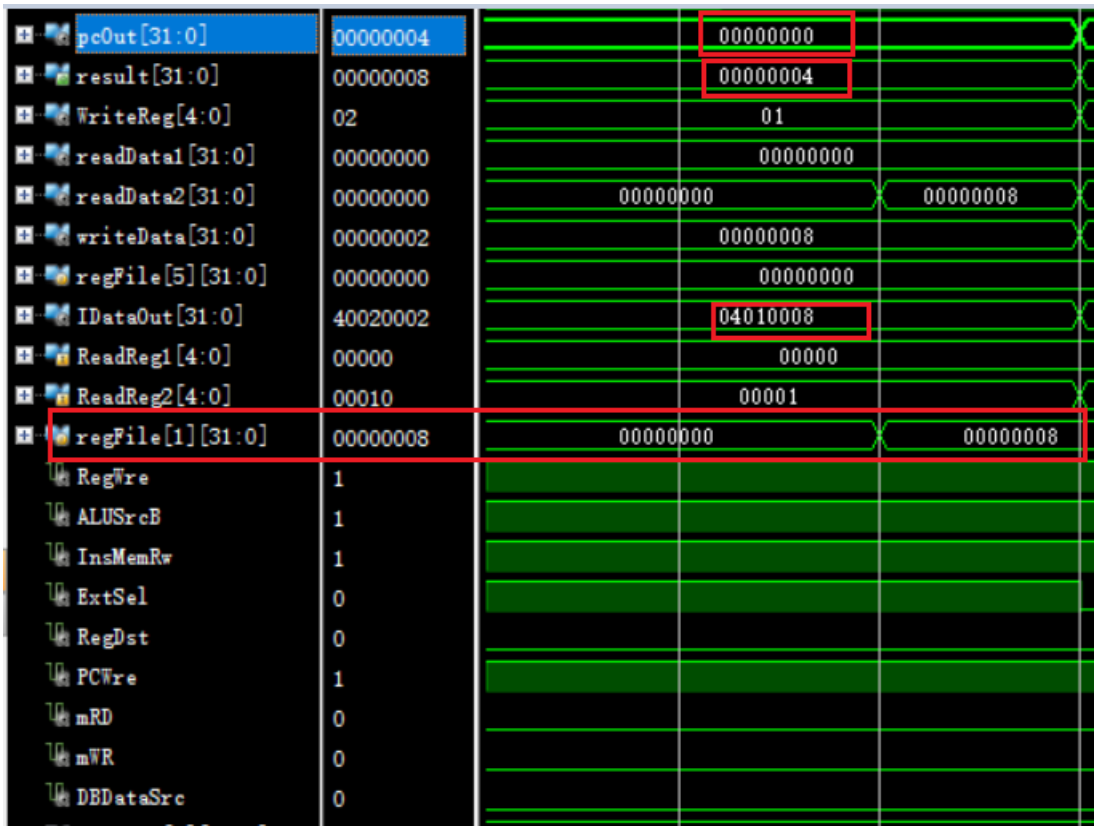
使用的测试程序段：

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	=	40020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000	=	411800
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000	=	8622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	=	44A22000
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000	=	48824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001C	bne $8,$1,-2 (≠,转18)	110000	01000	00001	1111 1111 1111 1110	=	C501FFFE
0x00000020	slti $6,$2,8	011011	00010	00110	0000 0000 0000 1000	=	6C460008
0x00000024	slti $7,$6,0	011011	00110	00111	0000 0000 0000 0000	=	6CC70000
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000	=	4E70008
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	=	C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100	=	98220004
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	=	9C290004
0x00000038	**j 0x00000040**	111000	00 000	00000	0000 0000 0001 0000	=	E0000010
0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010	=	40A000A
0x00000040	subi $1,$2,1	000011	00010	00001	0000 0000 0000 0001	=	C410001
0x00000044	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000

指令：addi $1,$0,8

波形图：



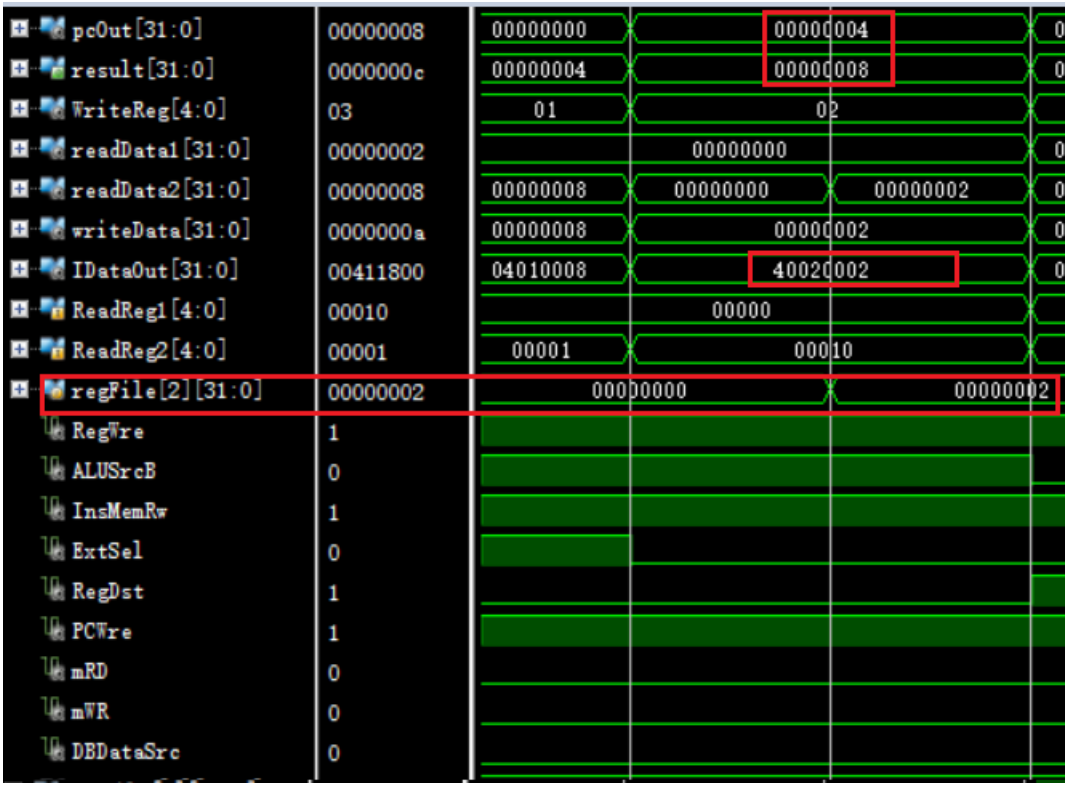
PcOut 为当前PC值，IDataOut 当前指令，reslut 为下一条指令地址（可能是顺序

执行地址，有可能是跳转的地址）

可以看到，IDataOut 为04010008，指令读取正确，寄存器1的值从0为8，结果正确。

指令：ori　$2,$0,2

波形图：

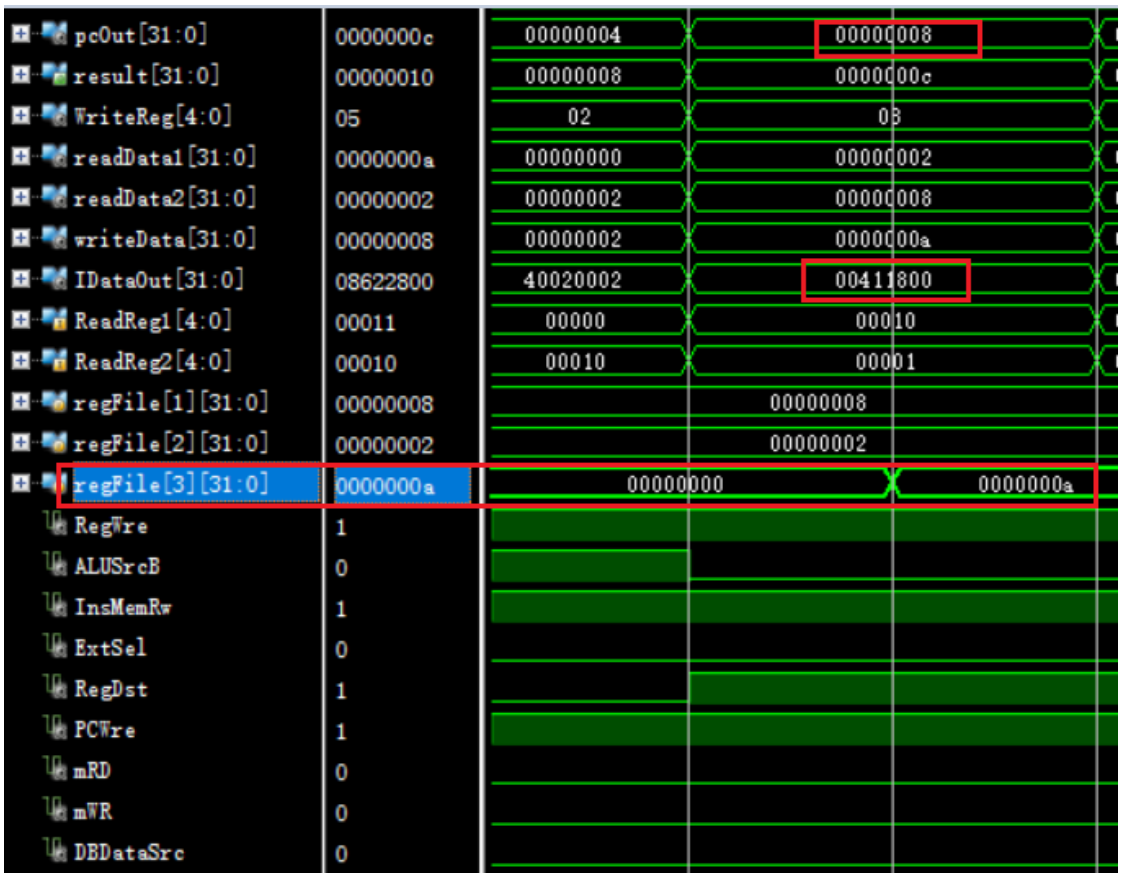


指令读取正确，寄存器2的值从0变为2，控制信号正确。

对应十六进制 40020002

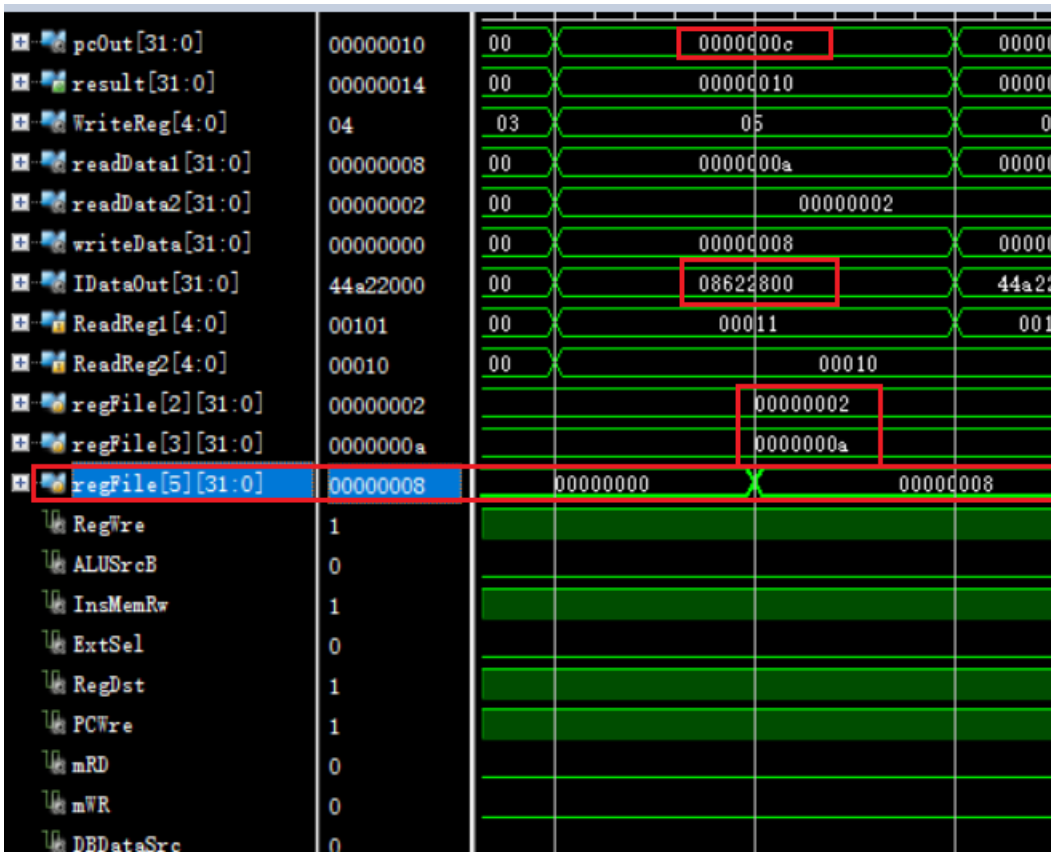
指令：add　$3,$2,$1

波形图：



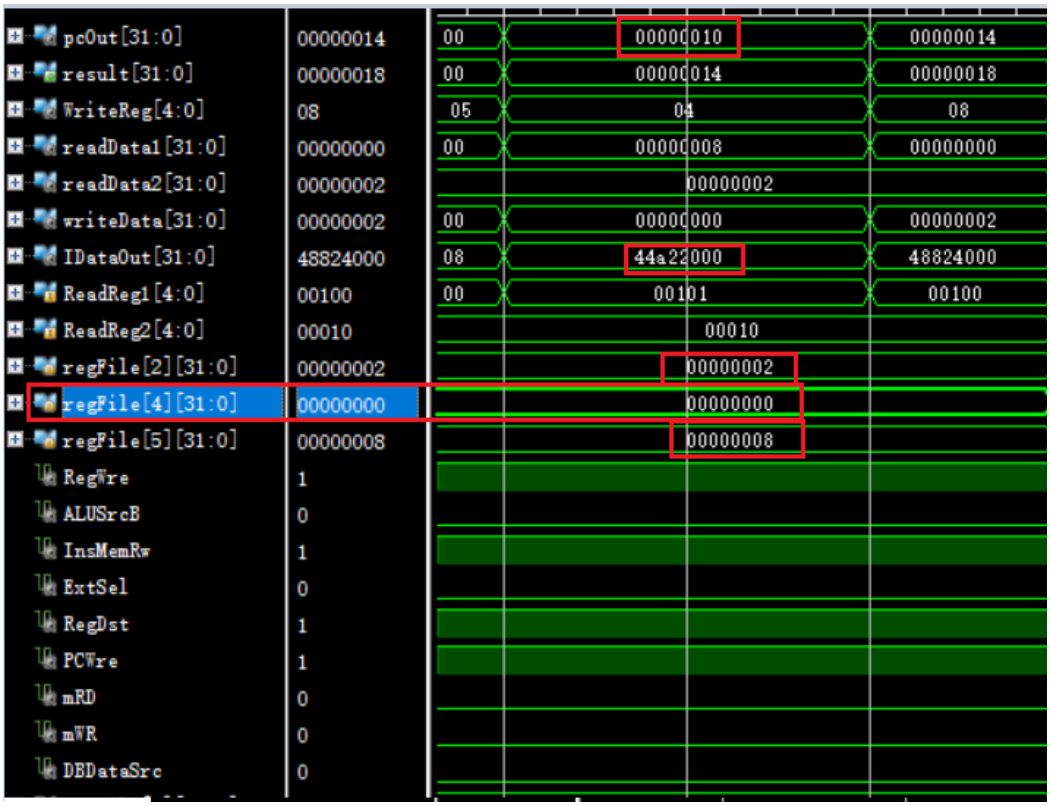
指令读取正确。寄存器3内容从0变为a ，即10.

指令：sub $5,$3,$2



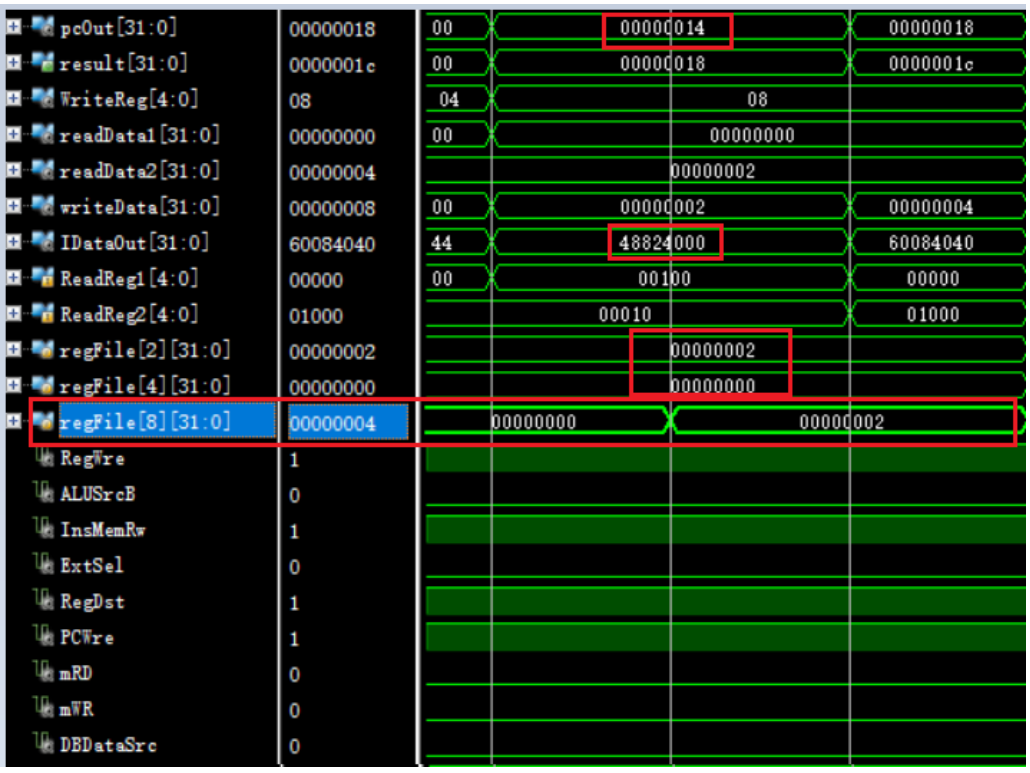
指令读取正确。寄存5的值从0变为8，等于寄存器3减去寄存器2的值，结果正确。

指令：and　$4,$5,$2



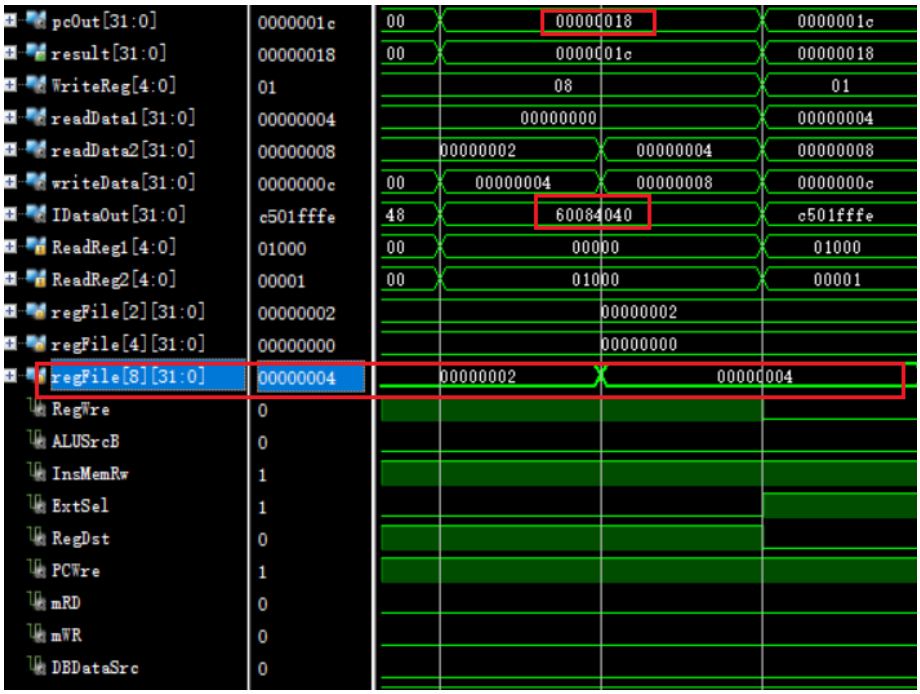
指令读取正确，寄存器2，寄存器5做与运算结果为0，寄存器4为0，结果正确。

指令：or　$8,$4,$2



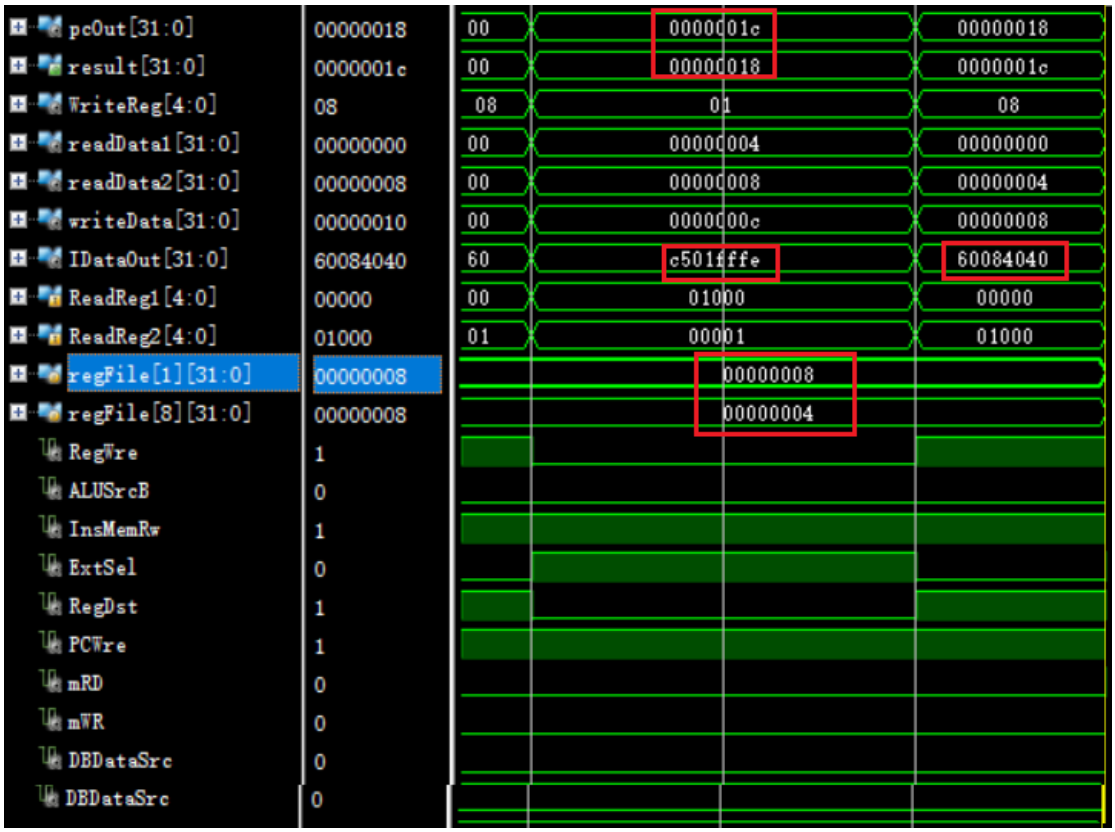
指令读取正确，寄存器8的值从0变为2，等于寄存器2与寄存器4做或运算的结果。

指令：sll　$8,$8,1



寄存器8的值从2左移一位，结果正确。

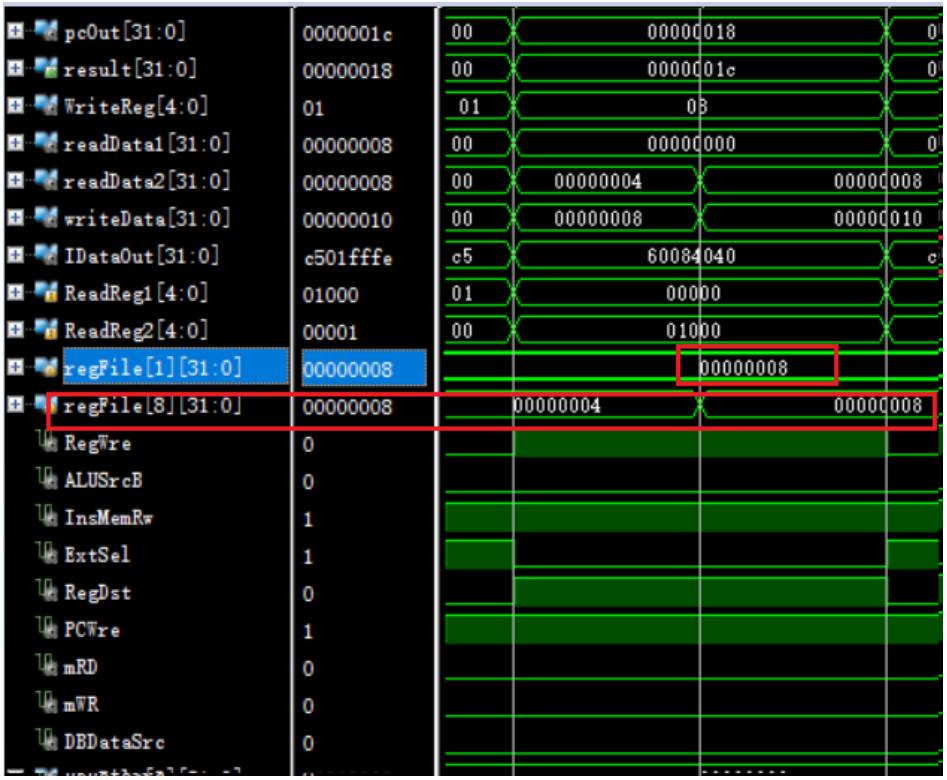
指令：bne　$8,$1,-2



比较寄存器1和寄存器8的值，不相等则跳转，可以看到两个值不相等，因此跳转到了

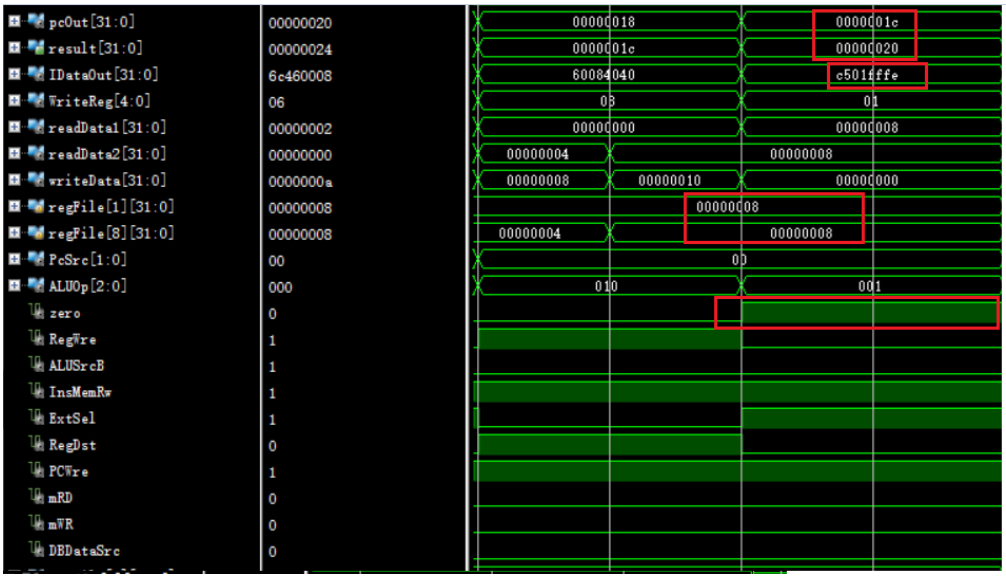
000018，结果正确。

因此程序再次执行指令：sll　$8,$8,1



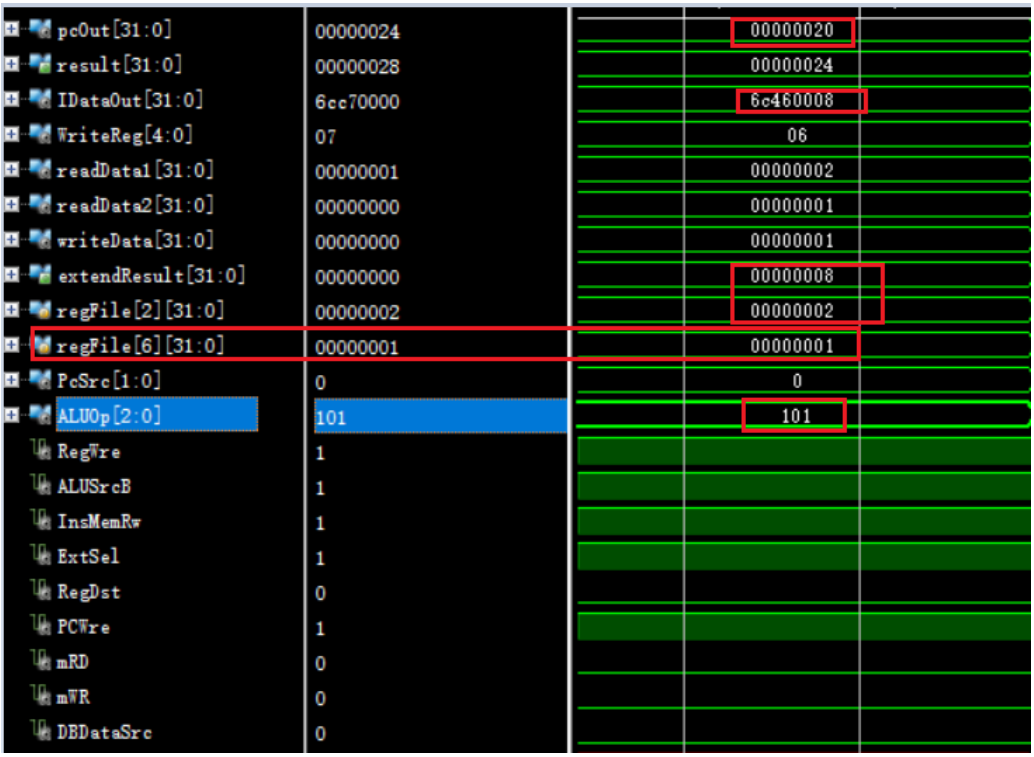
寄存器8的值左移一位，变为8.

再次执行指令：bne　$8,$1,-2



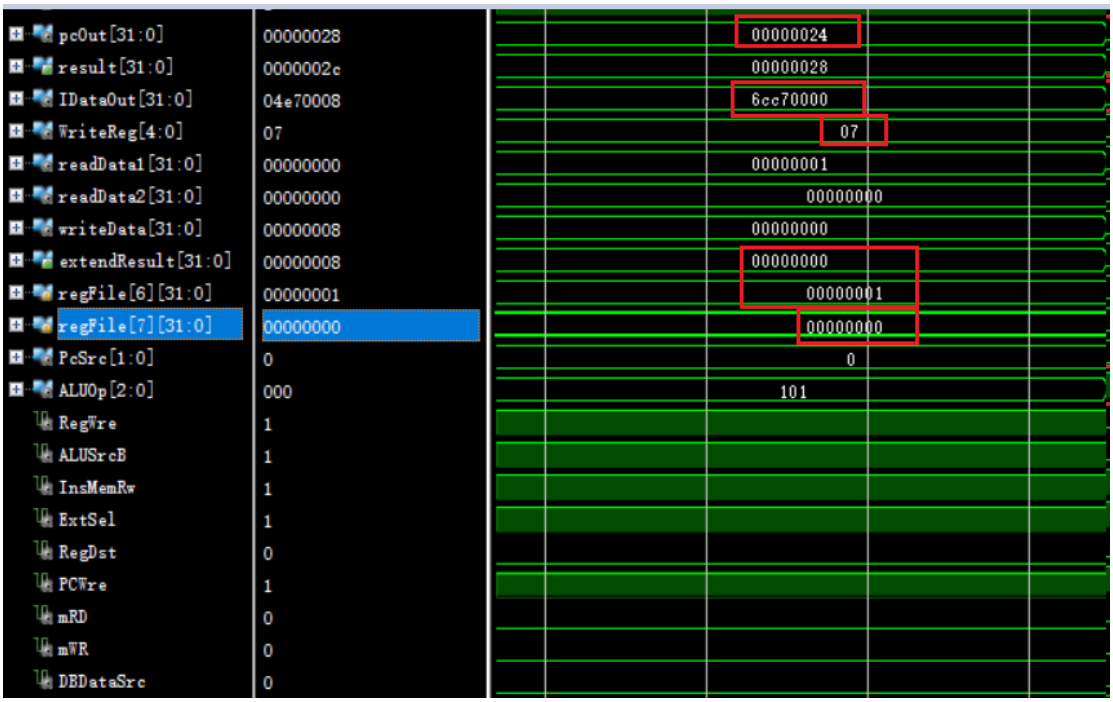
此时两者相等，zero 信号由0变为1，下一条指令的地址是 00000020

指令：slti　$6,$2,8



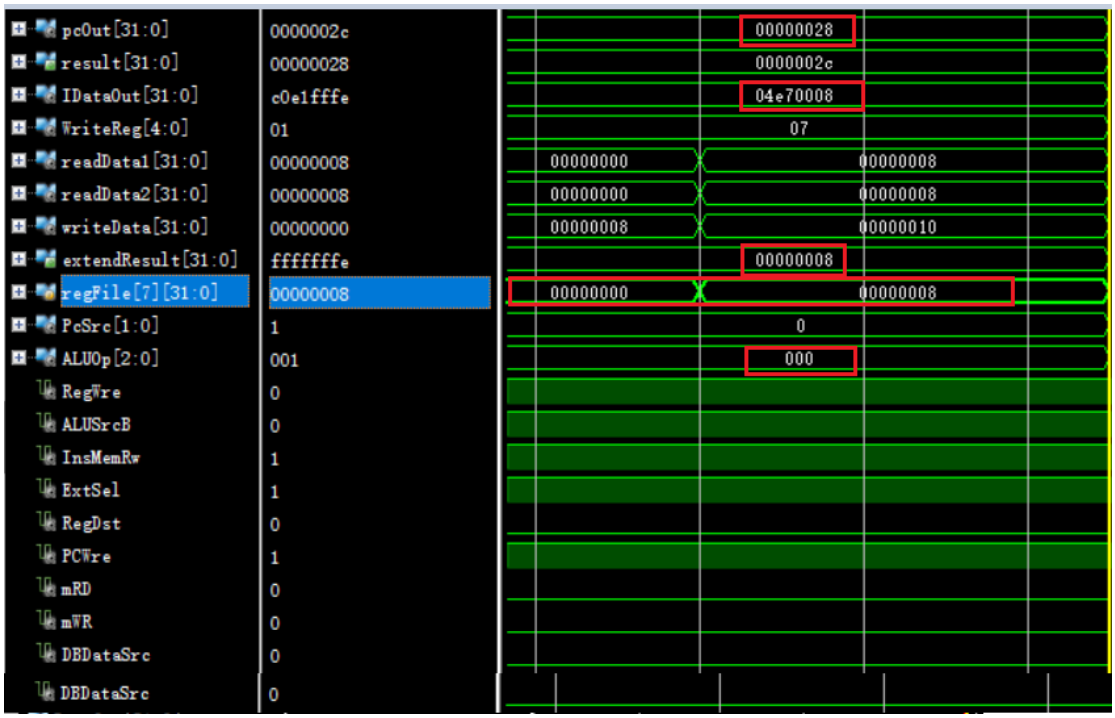
ExtendResult表示拓展后的立即数，寄存器2号值为2，小于立即数8，因此寄存器六号的值为1，结果正确。

指令：slti　$7,$6,0



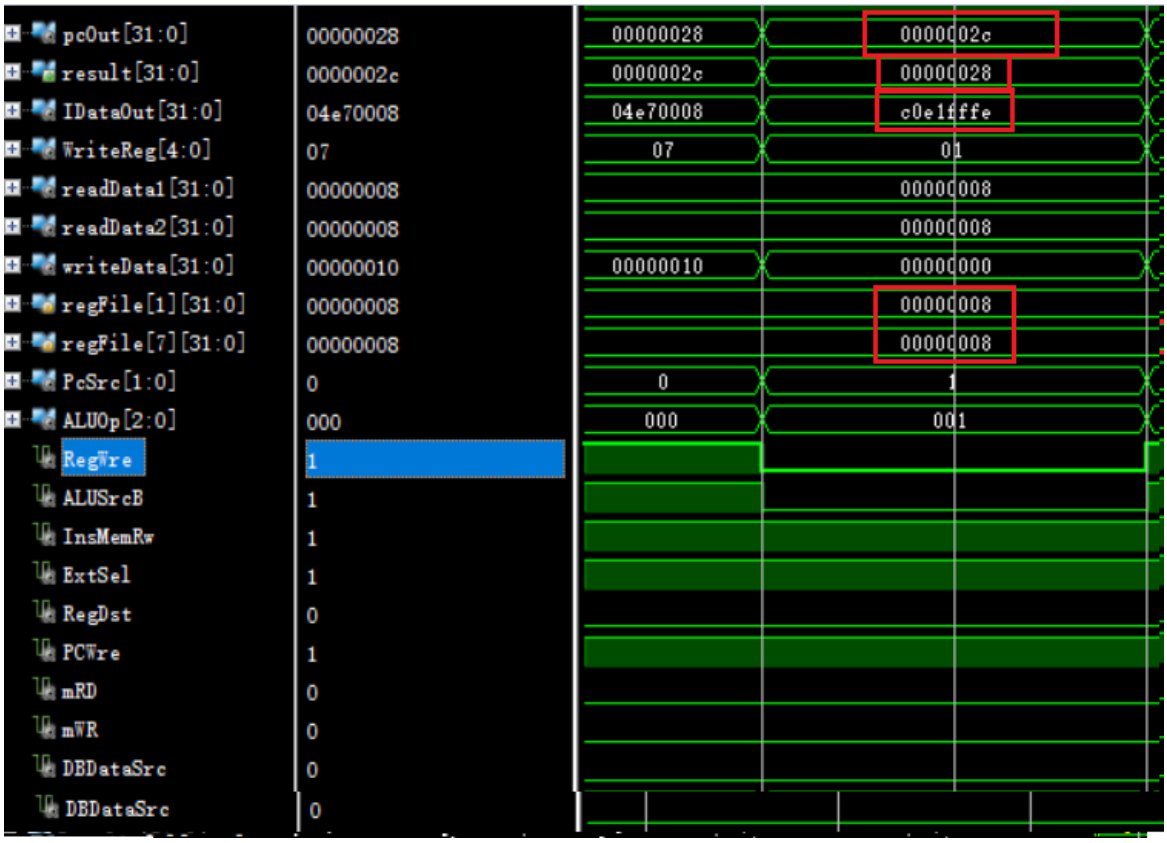
寄存6的内容是1，立即数为0，因此结果应该是0，寄存器7存放0。

指令：addi $7,$7,8



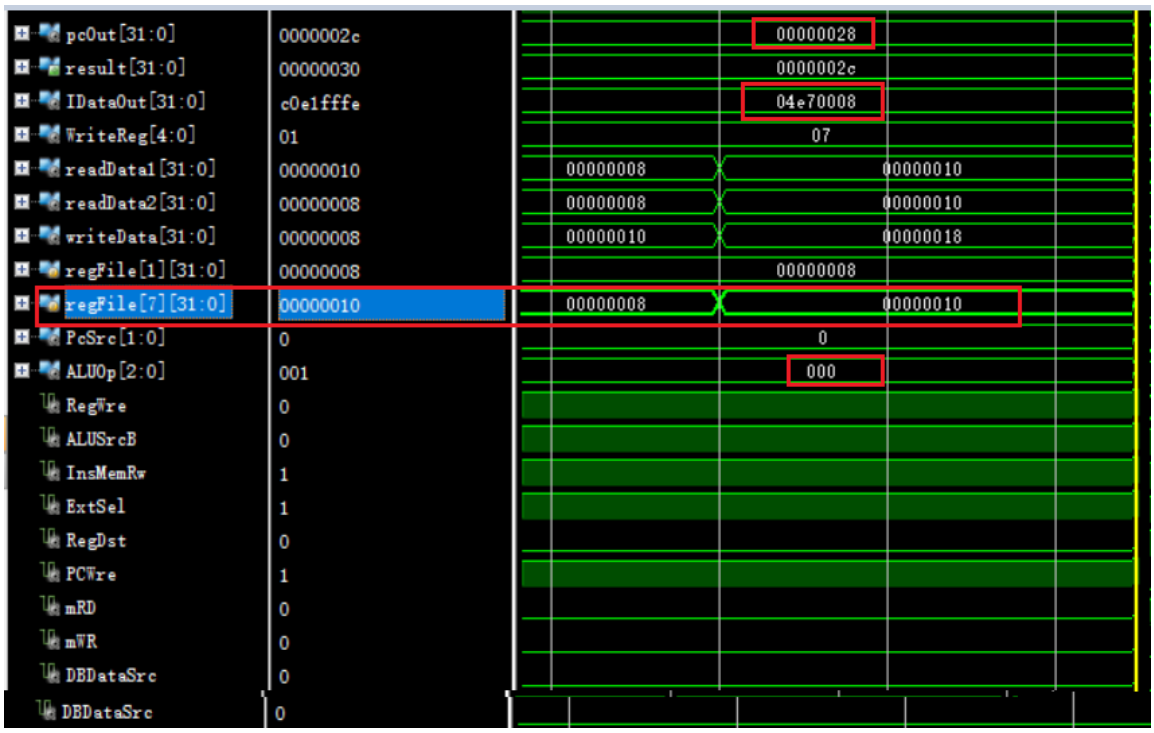
立即数为8，ALUop为000，代表加运算，寄存器7的内容从0变为8，结果正确。

beq $7,$1,-2：.



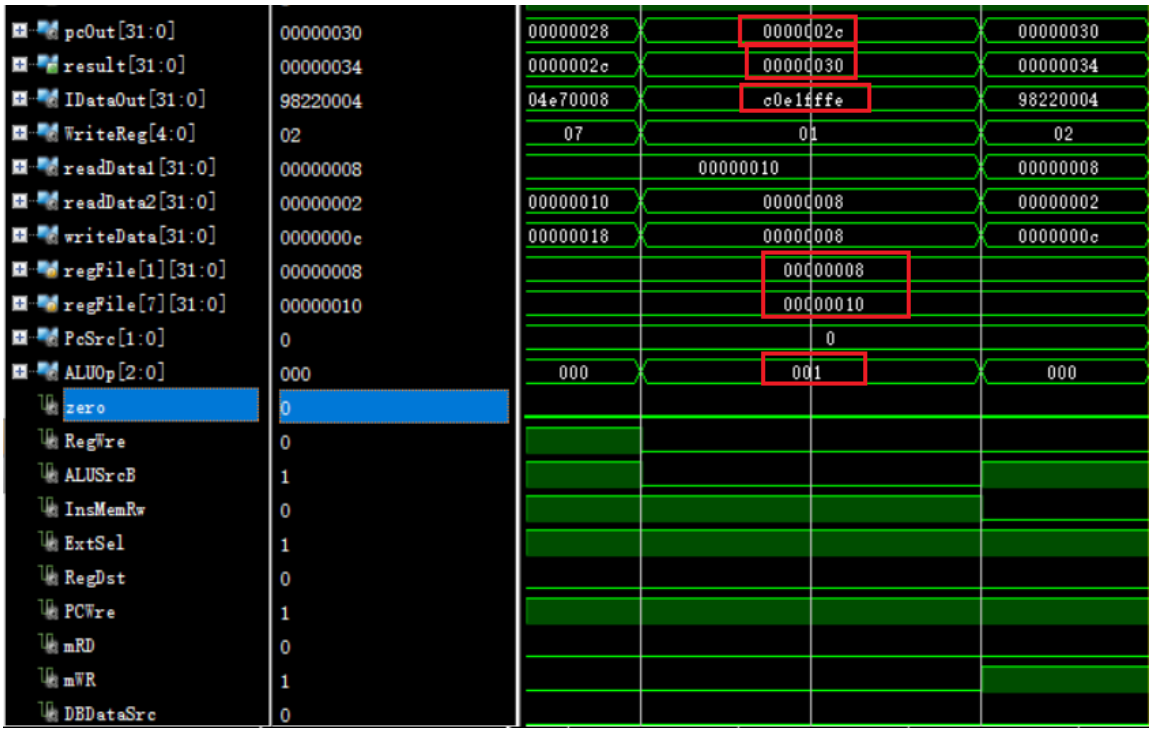
寄存器 1 和寄存器 7 内容相等，因此跳转到 00000028，result 显示的地址正确。

再次执行指令：addi $7,$7,8



寄存器7的值变为16

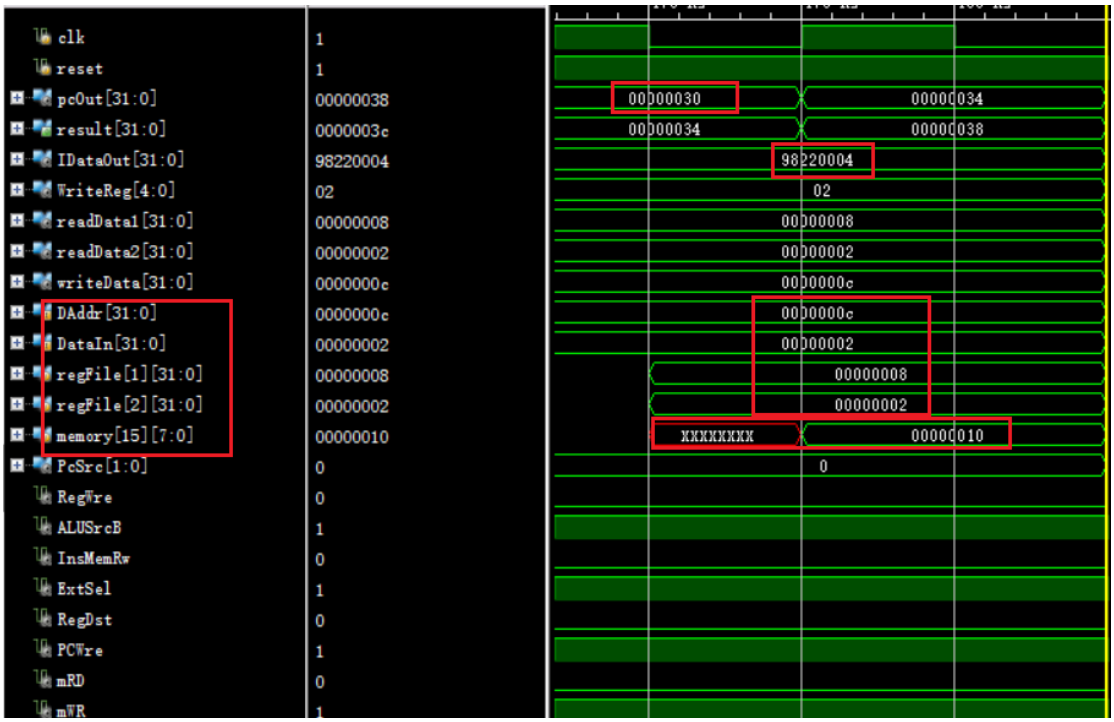
再次执行：beq $7,$1,-2



寄存器1的值为8，寄存器7的值为0x10，两者不一样，zero 为0，因此下一个指令的

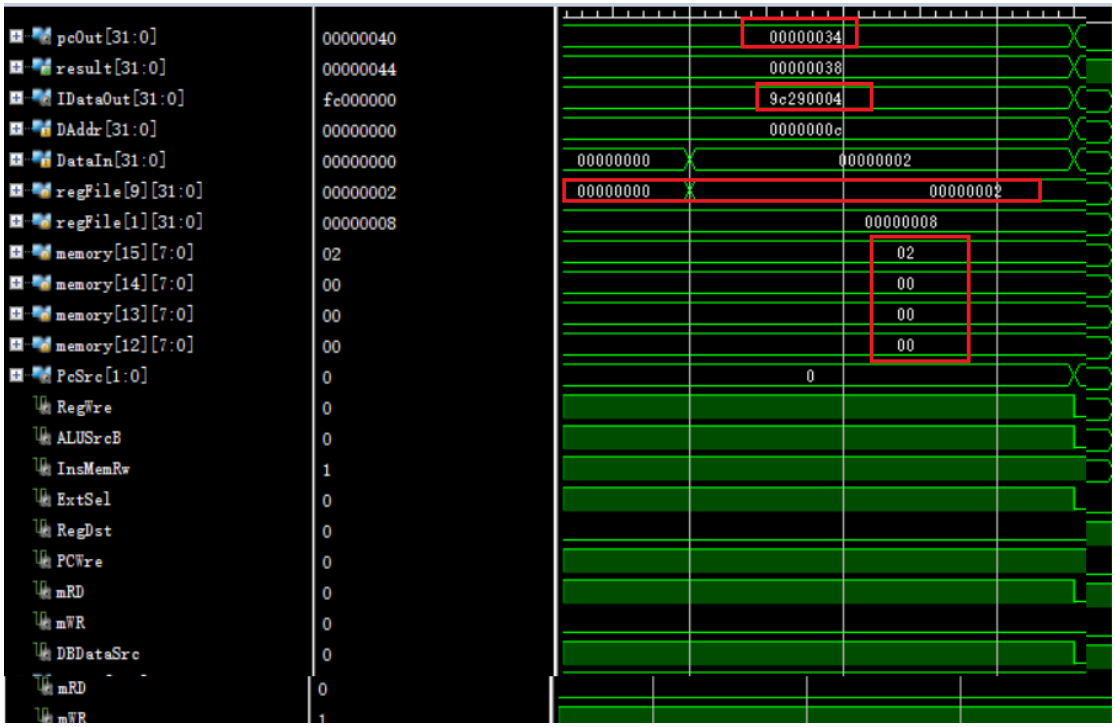
地址是0000030，reslut 显示的结果正确。

指令：sw　$2,4($1):



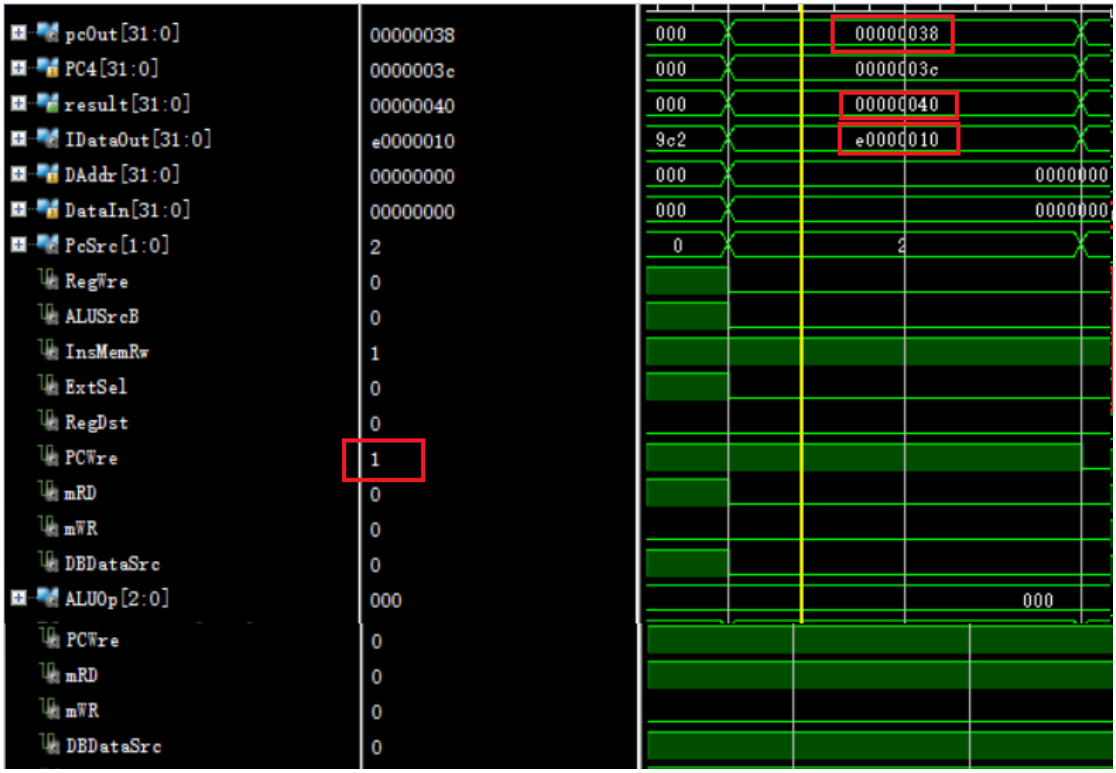
寄存器1内容为8,寄存器2内容为2,因此将2写进内存8+4的位置,又因为datamemory是8位存储的，因此是写2是写入 memory的位置15，可以看到，memory[15]变为00000010，结果正确。

指令：lw　$9,4($1)



寄存器一号内容为8，所以将存储器位置12（由memory[12]、memory[13]、memory[14]、memory[15]组成）中的数据写入寄存器9中，可以看到寄存器9存储的数据由0变为2，结果正确。

指令： j 0x00000040



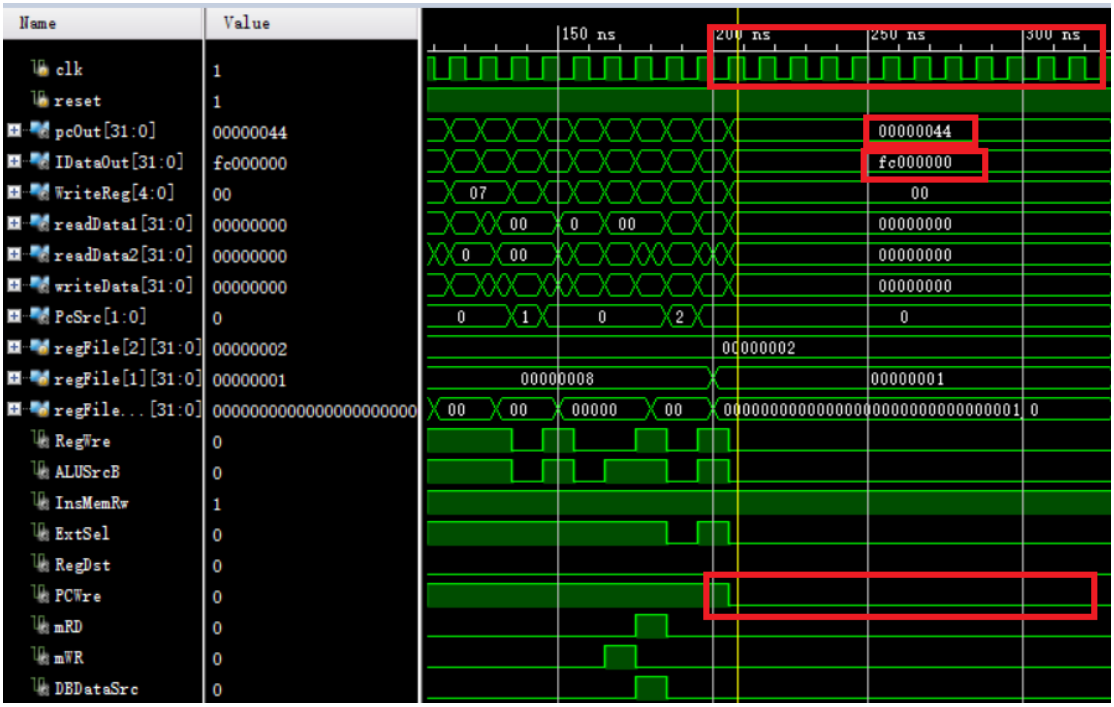
当前指令地址00000038，下条指令地址为00000040，跳过0000003c，结果正确。

指令：subi $1,$2,1

可以看到，二号寄存器内容为2，一号寄存器内容原来是8，将二号寄存器内容减去立

即数1，将结果存入一号寄存器，一号寄存器的值为1，结果正确。

指令：halt



可以看到，经过多个周期，程序结果都没变化，成功停止。

实现：

主要添加五个模块：

1.数码管扫描显示

根据时钟信号，改变控制显示的位置，达到扫描显示的效果

```
case(displayControl)
            4'b1110:
                displayControl <= 4'b1101;
            4'b1101:
                displayControl <= 4'b1011;
            4'b1011:
                displayControl <= 4'b0111;
            4'b0111:
                displayControl <= 4'b1110;
        endcase
```

四位分别对应数码管中的四位，低电平有限。

2.七段数码管译码

```
case (displayData)
        4'b0000 : dispcode = 8'b1100_0000;  //0; '0'-亮灯, '1'-熄灯
        4'b0001 : dispcode = 8'b1111_1001;  //1
        4'b0010 : dispcode = 8'b1010_0100;  //2
        4'b0011 : dispcode = 8'b1011_0000;  //3
```

根据要显示的数据得到数码管七段的信号

3.选择器，选择需要显示的信号，首先根据信号选择，再根据数码管此刻显示的位置选择具体的输入。

```
case(signal)
       2'b00:begin//当前 PC 值:下条指令 PC
           case(displayControl)
           4'b1011: //当前 PC 值的低 4 位
               displayData = PC[3:0];
           4'b0111://当前 PC 值的高四位
               displayData = PC[7:4];
           4'b1110:
               displayData = nextPC[3:0];
           4'b1101:
               displayData = nextPC[7:4];
           endcase
       end
```

4. 消抖模块

```
5. always @(posedge clk)
6.  if(count_high == SAMPLE_TIME)
7.   key_out_reg <= 1;
8.  else if(count_low == SAMPLE_TIME)
9.   key_out_reg <= 0;
```

SAMPLE_TIME 设为4，即延迟4个时钟周期

5.分频模块

```
always @(posedge clk or posedge rst) begin
    if (rst==0) begin
        clk_out <= 0;
        counter <= 0;
    end
    else if (counter == 99999) begin
        clk_out <= ~clk_out;
        counter <= 0;
    end
    else begin
          counter <= counter + 1;
    end
end
```

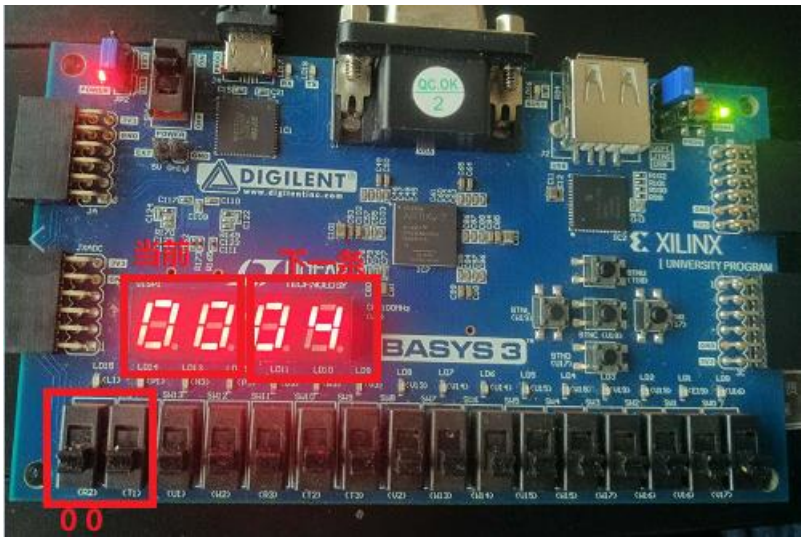
目标频率500hz，不分频的话，扫描显示数码管会　不稳定地显示 0000

将上面的模块实例化，添加到顶层模块中，注意，CPU 中地址，数据为32位，板子上的是8位。CPU中32位的数据取低8位。

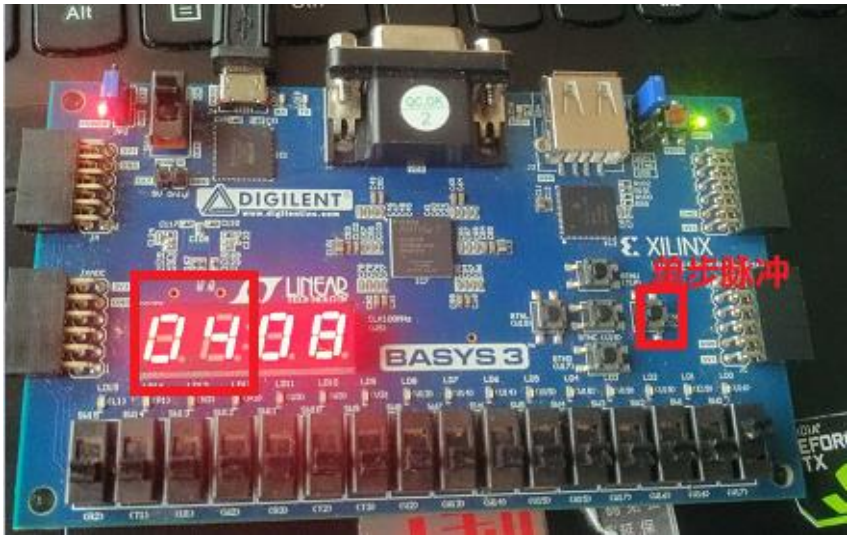
综合，实现，添加约束文件，将程序烧到板子中。

注意：在板子上测试时，首先需要在reset=0,即有效值的情况下，点击一次单次脉冲，否则第一条指令的值无法写进寄存器

00：显示 当前 PC值:下条指令PC值



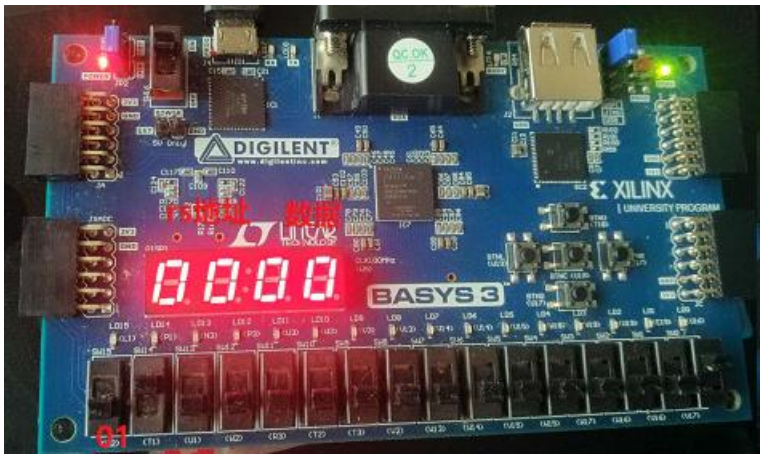
如图，当前PC值为00，下一PC值为04



当前PC值为4，下一PC值为8.

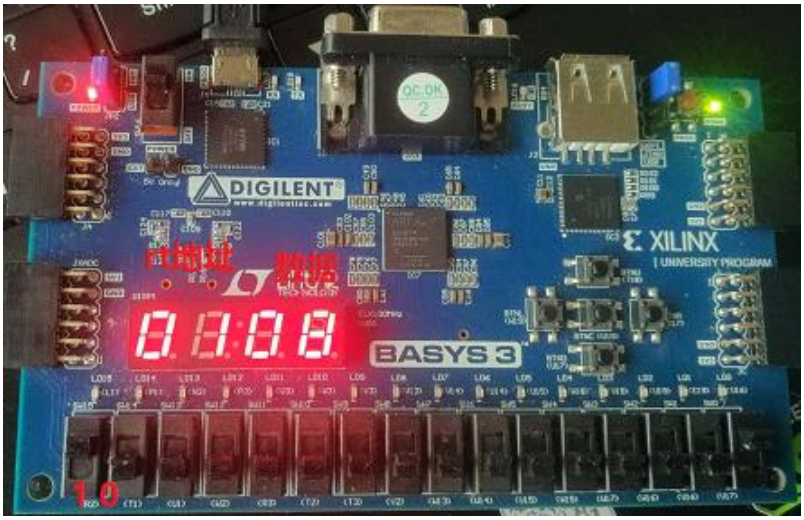
01：显示 RS寄存器地址:RS寄存器数据

当前指令:add　$3,$2,$1 ，rs 地址00000，对应数据为0



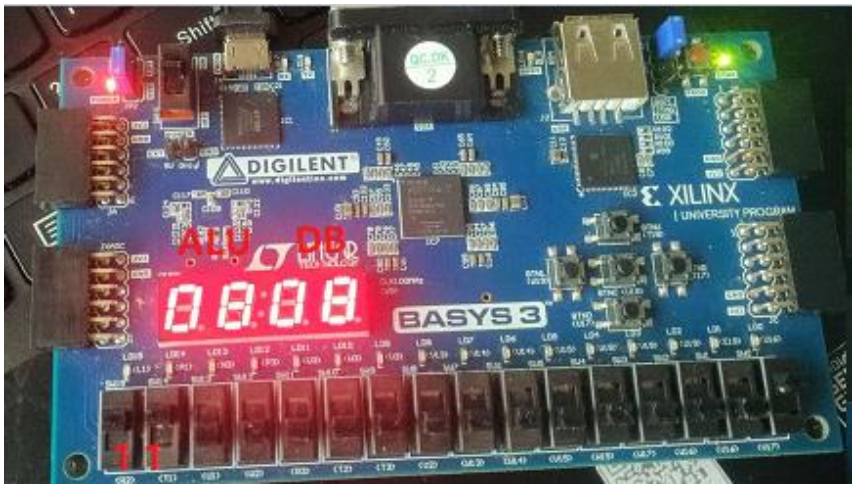
10：显示 RT寄存器地址:RT寄存器数据

当前指令 addi　$1,$0,8，因此 rt 为 00001 ，一号寄存器数据为 8，因此显示

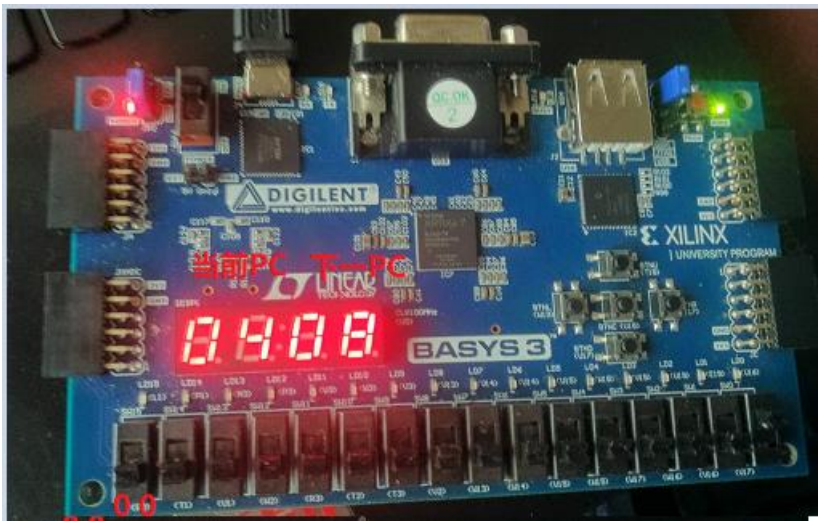


11 显示 ALU结果输出 :DB总线数据

指令addi $1,$0,8，ALU将0与8相加得到结果8，addi指令中，8需要写回到寄存器一号，因此DB数据也为8



按单步脉冲，显示当前PC和下一PC的地址，分别为4和8：



六. 实验心得

遇到的问题：

1. 提示文件权限错误

一开始以为是文件路径的问题，尝试过重新添加文件，修改文件路径等都不可以，后来发现目录cpu\cpu.sim\sim_1\behav 下的compile.log 文件，有详细的错误输出，比如变量未定义、变量名不匹配、位宽不匹配的问题，根据里面的输出修改就可以了。

2. 测试代码编写错误

程序执行后发现第一条指令的结果总是写不进寄存器,仔细查看波形图,分析得到代码,得到原因：寄存器写是时钟下降沿触发，程序计数器是时钟上升沿触发，并且在PC中初始化了第一条指令的地址，一开始clk=0,因此读取第一条指令后，上升沿先到来，直接读取了下一条指令，寄存器来得及写数据。解决方法：在测试文件中，先设置reset为有效，过了一个时钟周期后，再将reset 设为无效值，这样在第一次时钟上升沿到来时读取的依旧是第一条指令。

……

体会和建议：

本次实验较为复杂，一开始没有头绪，但是根据数据通路图，把实验分成各个模块，思考模块之间的联系，输入与输出信号，就很容易上手编写了，若想拓展指令，只要确定好对应的操作符，修改控制单元模块就可以了。编写完就开始 debug 之路了。

一部分是编译问题，如语法错误、 wire 和 reg 类型的使用，位宽需要不匹配等。根据报错就可以修改，或者简单搜索也可以解决。

还有一大部分是测试问题，经常得不到想要的结果，根据波形图和变量值大致就可以判断出错误在哪了。比如指令对应的控制信号输出是错的导致指令没有正确读取，跳转地址不对，数据没有写入存储器等，都是模块编写的逻辑错误。

烧板子的时候，扫描显示总是出问题，但是没有具体输出，不方便调试，每次改一行代码，就要重新综合、实现、写板。因为前面测试过CPU模块没有题，因此写板出现问题时也只需要关注新增的模块，没有循环显示时，查看分频有没有错误。新增模块整体逻辑还是比较简单的，就是每次烧板子都要耗费很多时间。

本次实验不算难，最重要的是细心和耐心，不管对内容多么熟，都不可避免地花费很多时间在上面。

通过本次实验，对于 MIPS 的指令结构，CPU 的指令执行有了深入的认识，同时也锻炼了动手能力和 debug 能力，通过编写实验报告，重新梳理了自己的思路，对实验整体的理解更加清晰了。