中山大學 SUN YAT-SEN UNIVERSITY

实验三 ： 多周期CPU设计与实现

一. 实验目的

(1) 认识和掌握多周期数据通路原理及其设计方法；
(2) 掌握多周期 CPU 的实现方法，代码实现方法；
(3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
(4) 掌握多周期 CPU 的测试方法;
(5) 掌握多周期 CPU 的实现方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：（**说明：操作码按照以下规定使用，都给每类指令预留扩展空间，后续实验相同。**）

==>算术运算指令

（1）add rd, rs, rt

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

功能：rd<－rs + rt

（2）sub rd, rs, rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

完成功能：rd<－rs － rt

（3）addi rt, rs, **immediate**

000010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt<－rs + (sign-extend)**immediate**

==>逻辑运算指令

（4）or rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs | rt

（5）and rd, rs, rt

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

功能：rd<－rs & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

功能：rt<－rs | (zero-extend)**immediate**

==>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

100110	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

（9） sltiu rt, rs,immediate 不带符号

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

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

==>存储器读写指令

（10）sw rt, **immediate**(rs)

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

功能：memory[rs+ (sign-extend)**immediate**]<－rt。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)

110001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt <－ memory[rs + (sign-extend)**immediate**]。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** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110100	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

(13) bltz rs,**immediate**

110110	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],2'b00}，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，剩下最高 4 位由 pc+4 最高 4 位拼接上。

（15）jr rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（16）jal addr

111010	addr[27:2]

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],2'b00}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（17）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变 pc 的值，pc 保持不变。

三. 实验原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段：

(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。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

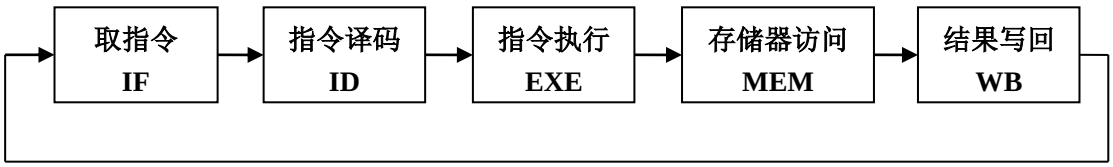
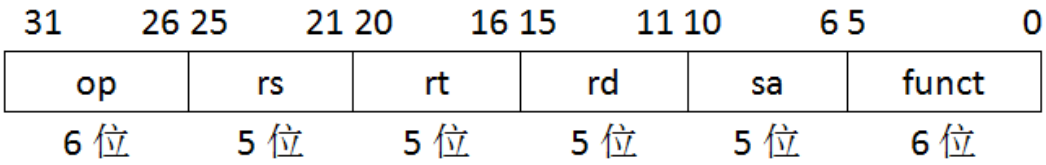


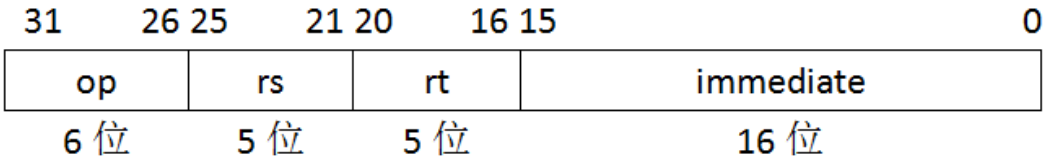
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式：

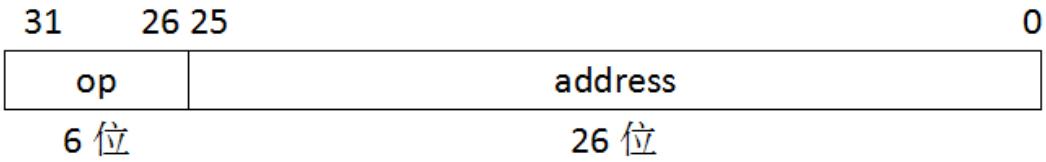
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

rt：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：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；

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。

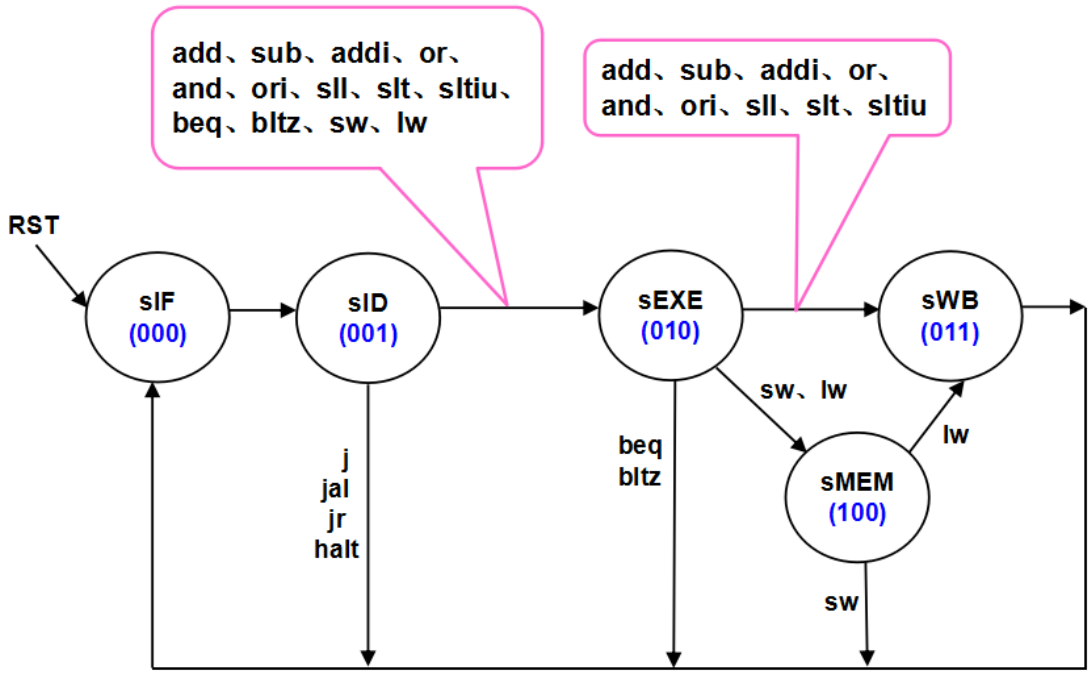


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 sIF 状态转移到 sID 就是无条件的；有些是有条件的，例如 sEXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

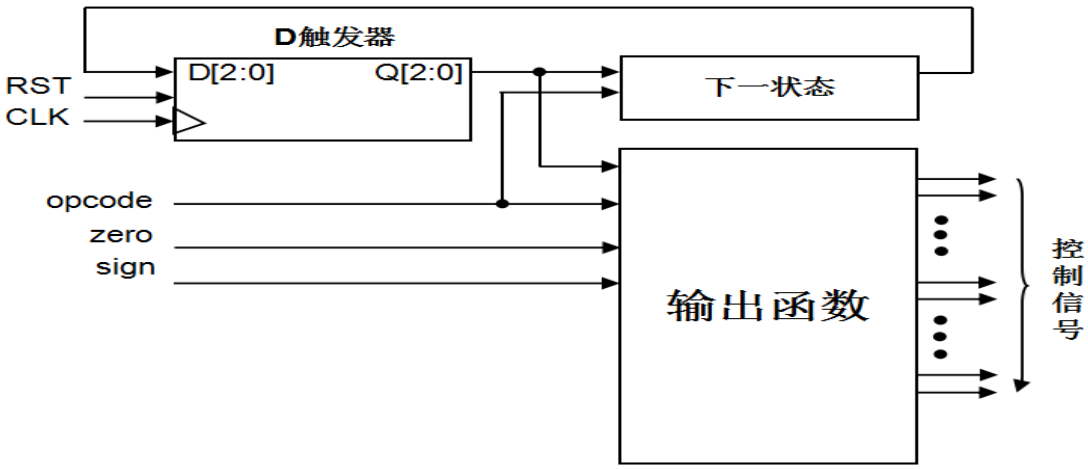


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

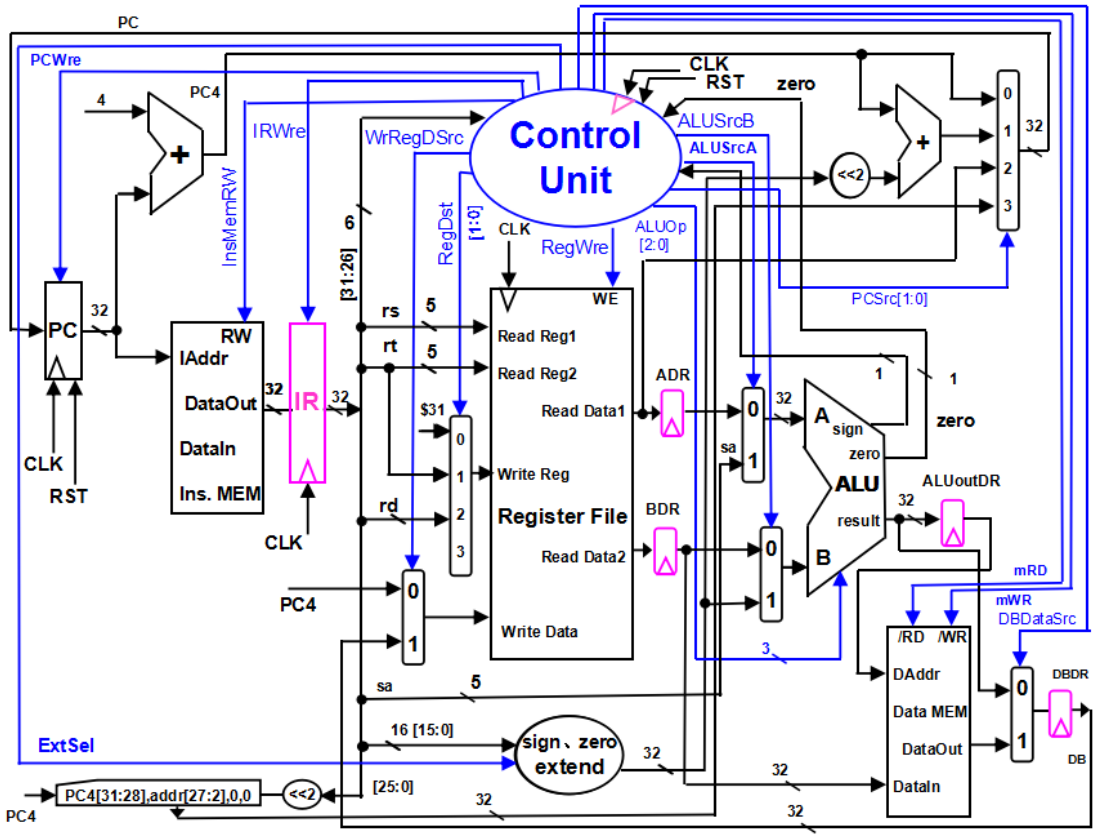


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 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，给出寄存器地址（编号），读操作时不需要时钟信号，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。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号，**其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1　控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bltz、slt、sltiu、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指	来自 sign 或 zero 扩展的立即数，相关

	令：add、sub、or、and、beq、bltz、slt、sll	指令：addi、ori、sltiu、lw、sw
DBDataSrc	来自 ALU 运算结果的输出,相关指令：add、sub、addi、or、and、ori、slt、sltiu、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、bltz、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、sltiu、sll、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4），相关指令：jal，写$31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据，相关指令：add、addi、sub、or、and、ori、slt、sltiu、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	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
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)**immediate**，相关指令：ori、sltiu；	(sign-extend)**immediate**，相关指令：addi、lw、sw、beq、bltz；
PCSrc[1:0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、sltiu、sll、sw、lw、beq(zero=0)、bltz(sign=0，或 zero=1); 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bltz(sign=1，zero=0); 10：pc<－rs，相关指令：jr; 11：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j、jal;	
RegDst[1:0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addi、ori、sltiu、lw; 10：rd 字段，相关指令：add、sub、or、and、slt、sll; 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，指令地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

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，数据地址输入端口

DataIn，存储器数据输入端口

DataOut，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，其地址输入端口（rt、rd）
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

IR：　指令寄存器，用于存放正在执行的指令代码

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= (A<B) ?1: 0	比较 A 与 B 不带符号
011	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
100	Y = B<<A	B 左移 A 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
111	Y = A ⊕ B	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

首先分析CPU数据通路图，通过单周期CPU的实验，我们已经知道可以采用模块化的思想。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，对比与单周期CPU的不同之处，理解数据如何在各个模块之间传递，确定模块的功能，输入与输出信号，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

首先明确需要编写的模块:

基础模块:

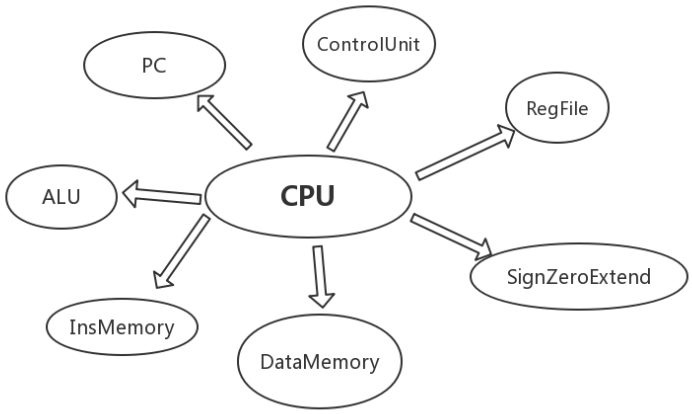
模块说明:

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	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

补充模块:

编号	名称	说明
1.	DataReg 数据寄存器	切分数据通路
2.	MuxFor32Bits二选一	针对32位数据的二选一
3	MUX4 四选一	计算下一条指令的地址
4.	MuxALUa 二选一	选择ALU操作数a
5	WriteDataSrc	选择写入寄存器的数据
6	WriteRegSrc	选择写入数据的寄存器号
7	DataReg	切分数据通路模块
8	DBDReg	下降沿触发的DataReg

主要模块图：



具体模块设计：

控制单元

该模块是本次实验的关键部分，编写的时候分为两个部分，下一状态，控制信号。而单周期中只有控制信号变化

下一状态

用于产生下一个阶段的状态，下个状态取决于指令操作码和当前状态，时钟下降沿触发，或者rst等于0时状态变为0。使用case与语句，分析当前状态，结合指令操作码，从CPU状态转移图中可以看到，不同的指令操作码的状态变化不同，如jal与lw，因此需要进行处理。

```
case(o_state)
          3'b000:begin
            jal = 0;
            o_state=3'b001;
          end
          3'b001:begin
            case(op[5:3])
              3'b111:o_state=3'b000;// 指令 j jal jar halt
              default:o_state=3'b010;
            endcase
          end
```

控制信号

产生每个阶段的控制信号，控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。控制信号比单周期 CPU 多，需要注意。并且要结合当前状态。如 PCWre 只有在状态为 0 且不是停机指令时有效。

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,slt,subi,halt,sltiu,b
ltz,jal,jr;
```

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 = 1;
        6'b000001:
          sub = 1;
        6'b000010:
          addi= 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  PCWre = !halt && (o_state==3'b000);
assign  ALUSrcA = sll;
assign  ALUSrcB = addi || ori||sltiu||slti||sw||lw||subi;
```

PcSrc 和 ALUop 采用逐位赋值，根据ALU与指令的关系表。

```
 assign PcSrc[1] = j ||jal||jr;
 assign ALUOp[0] = sub||orx||ori||beq||slt||bne||slti||subi||bltz;
```

ALU算术单元

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。这些参照ALU运算功能表实现除此之外还有zero和sign信号。sign标志有符号比较。操作码对应的操作与单周期不同，需要注意。

```
assign zero = (result==0)?1:0;
assign sign = (ALUopcode==3'b011)?1:0;
always @( ALUopcode or rega or regb ) begin
    case (ALUopcode)
      3'b000 : result = rega + regb;
```

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
```

寄存器模块

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];
```

写数据需要时钟上升沿触发，要判断 RST、RegWre、WriteReg 是否有效

```
always @ (posedge 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
```

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};
```

MUX4 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1，选择立即数表示的地址，对应的指令有 bne,若为2，则选择31号寄存器中存储的地址，对应指令为 jr,若为3 则为拓展后的地址，对应指令有 j。

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

WriteRegSrc 选择目的寄存器

通过判断RegDst信号选择目的寄存器，若为0，则选择31号寄存器，若为2，选择rd，否则选择rt

```
assign WriteReg = RegDst==0?5'b11111:(RegDst==2?rd:rt);
```

WriteDataSrc 选择写入寄存器的数据

通过判断WrRegDSrc信号，选择写入的数据。

```
assign WriteData = (WrRegDSrc==1)?DataOut:PC4;
```

MuxFor32Bits 32位二选一

使用三目运算符直接选择，该模块可用来选择ALU操作数b和选择DBdata.。

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

DataReg 数据寄存器

模块用来切割数据通路，使用上升沿触发，可用作ADR,BDR部分

```
  always @(posedge clk) begin
    outData = inData;
  end
```

DBDReg 数据通路数据寄存器

该模块与DataReg相似，只是改为下降沿触发。

```
  always @(negedge clk) begin
    outData = inData;
  end
```

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。

```
//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：

```
PC pc(clk,reset,pcIn,PCWre,pcOut,PC4);
MUX4 PCSelector(PC4,IDataOut[25:0],extendResult,readData1,PcSrc,pcIn);
```

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

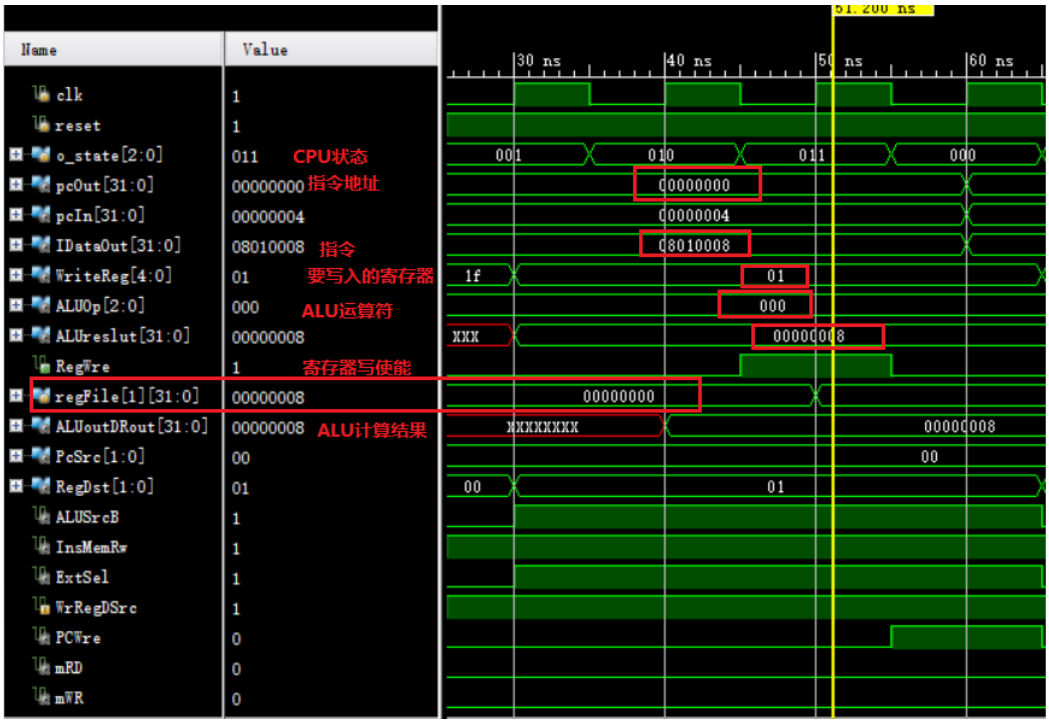
```
always begin
   #5  clk =~ clk;
end
```

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

使用测试程序段:

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	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	or $3,$2,$1	010000	00010	00001	0001 1000 0000 0000	=	40411800
0x0000000C	sub $4,$3,$1	000001	00011	00001	0010 0000 0000 0000	=	4612000
0x00000010	and $5,$4,$2	010001	00100	00010	0010 1000 0000 0000	=	44822800
0x00000014	sll $5,$5,2	011000	00000	00101	0010 1000 1000 0000	=	60052880
0x00000018	beq $5,$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000	=	E8000010
0x00000020	slt $8,$12,$1	100110	01100	00001	0100 0000 0000 0000	=	99814000
0x00000024	addi $13,$0,-2	000010	00000	01101	1111 1111 1111 1110	=	80DFFFE
0x00000028	slt $9,$8,$13	100110	01000	01101	0100 1000 0000 0000	=	990D4800
0x0000002C	sltiu $10,$9,2	100111	01001	01010	0000 0000 0000 0010	=	9D2A0002
0x00000030	sltiu $11,$10,0	100111	01010	01011	0000 0000 0000 0000	=	9D4B0000
0x00000034	addi $13,$13,1	000010	01101	01101	0000 0000 0000 0001	=	9AD0001
0x00000038	bltz $13,-2 (<0,转 34)	110110	01101	00000	1111 1111 1111 1110	=	D9A0FFFE
0x0000003C	j 0x000004C	111000	00000	00000	0000 0000 0001 0011	=	E0000013
0x00000040	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004
0x00000044	lw $12,4($1)	110001	00001	01100	0000 0000 0000 0100	=	C42C0004
0x00000048	jr $31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x0000004C	subi $1,$2,1	000011	00010	00001	0000 0000 0000 0001	=	C410001
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

指令：addi　$1,$0,8



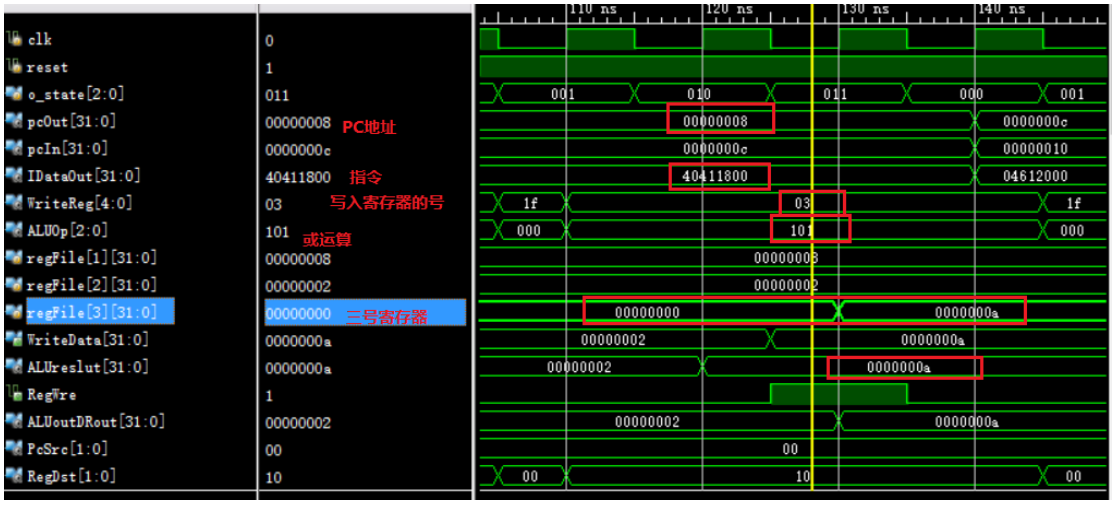
当前指令地址是0x00000000，对应指令08010008，该指令执行顺序：IF->ID->EXE->WB->IF，即 o_state 变化顺序 000->001->010->011->000,ALUop为000，代表加运算，运算结果 8，当 CPU 为 WB 时，RegWre 变为1，将结果写入1号寄存器，1 号寄存器数据从 0 变为 8，结果正确。

指令：ori　$2,$0,2

指令执行顺序IF->ID->EXE->WB->IF,当前PC地址为0x00000004，ALUop 为101，

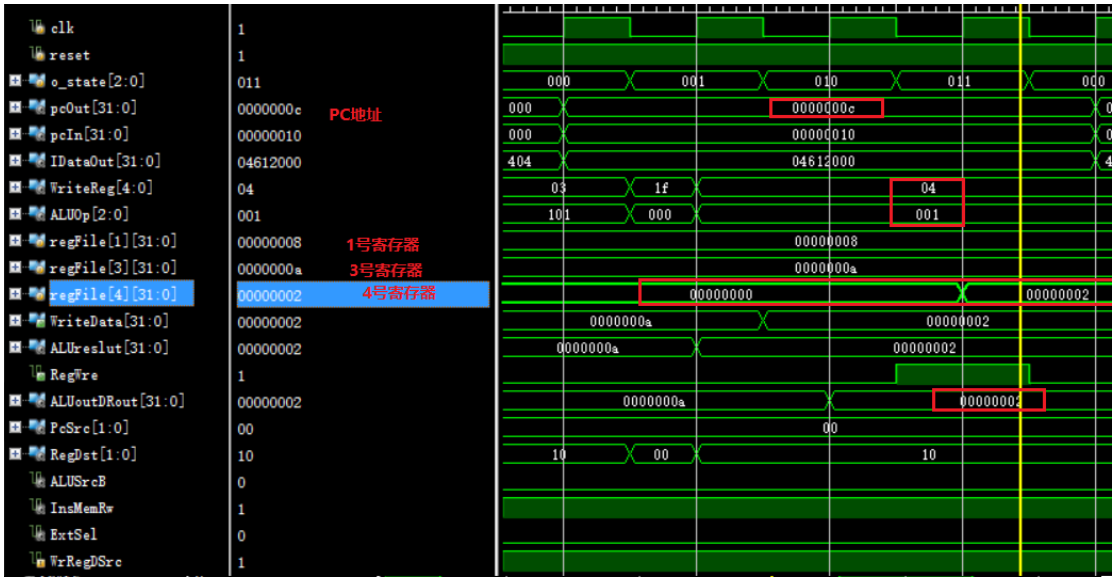
代表或运算，运算结果为2，要写入的寄存器为2，2号寄存器数据从 0 变为 2，指令执行正确。

指令：or　$3,$2,$1



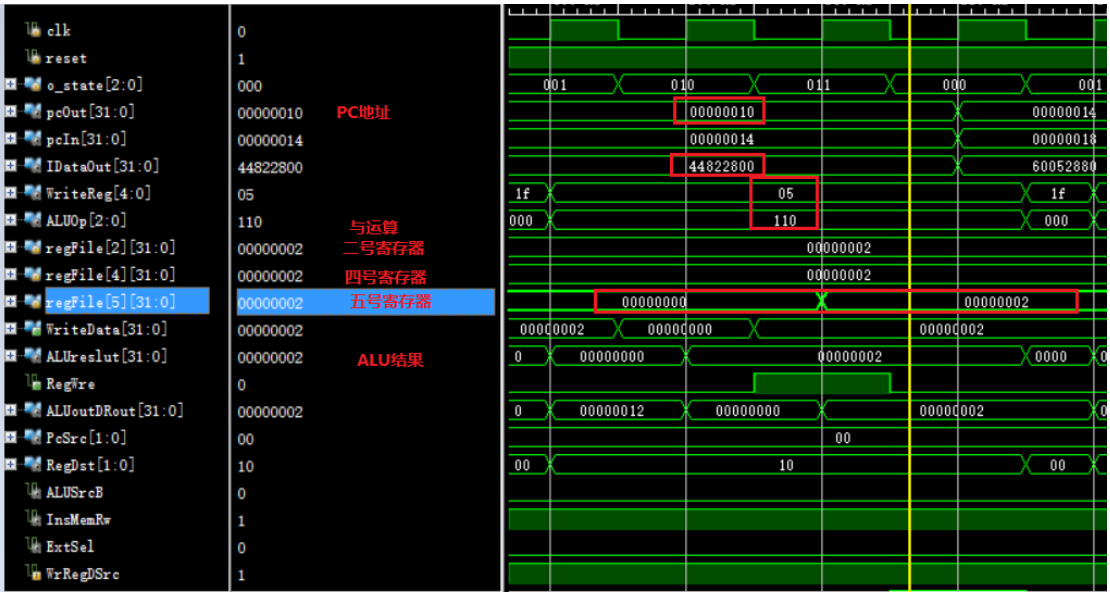
指令执行顺序IF->ID->EXE->WB->IF。1号寄存器的内容为 8，二号寄存器内容为2，做或运算，ALUop 为101，得到10，写入寄存器号为3号，3号寄存器内容从 0 变为 0xa，结果正确。

指令：sub　$4,$3,$1



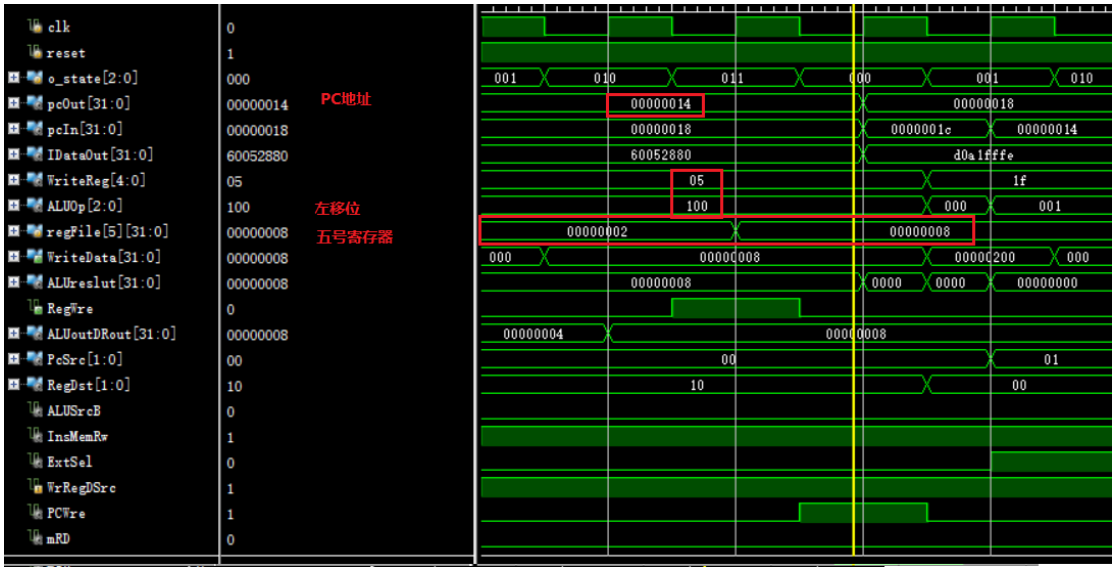
当前PC地址为0x0000000C，指令执行顺序IF->ID->EXE->WB->IF，1号寄存器内容为8，3号寄存器内容为10，ALUop 为001，执行减运算，结果为2，4号寄存器内容从0变为2，结果正确。

指令：and　$5,$4,$2



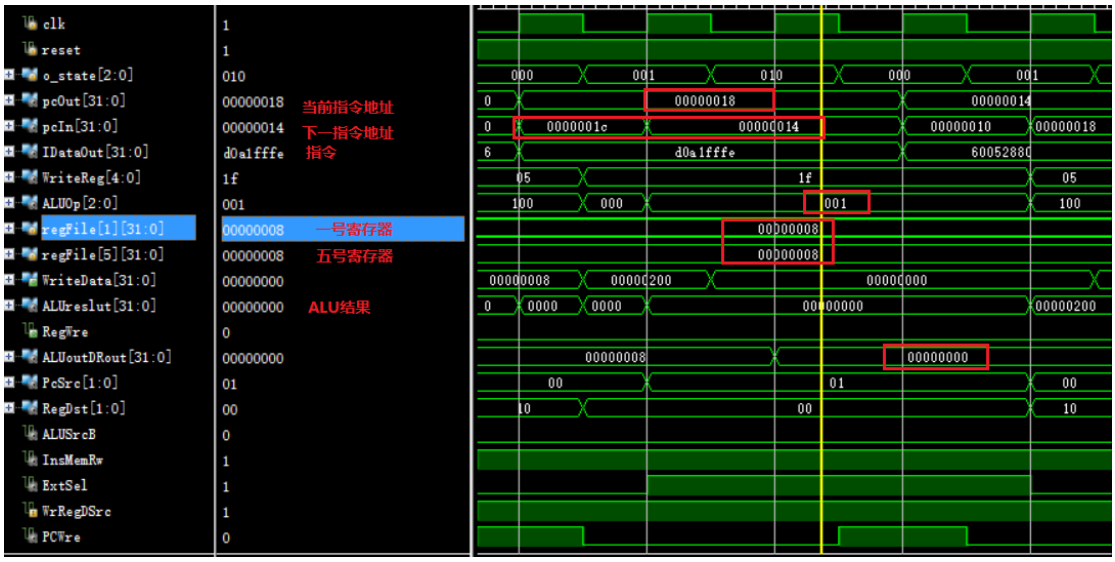
当前PC地址0x00000010，2号寄存器内容为 2，4 号寄存器内容为2，ALUop 为110，执行与运算，结果为2，写入5号寄存器，五号寄存器数据从 0 变为 2，结果正确。

指令：sll　$5,$5,2



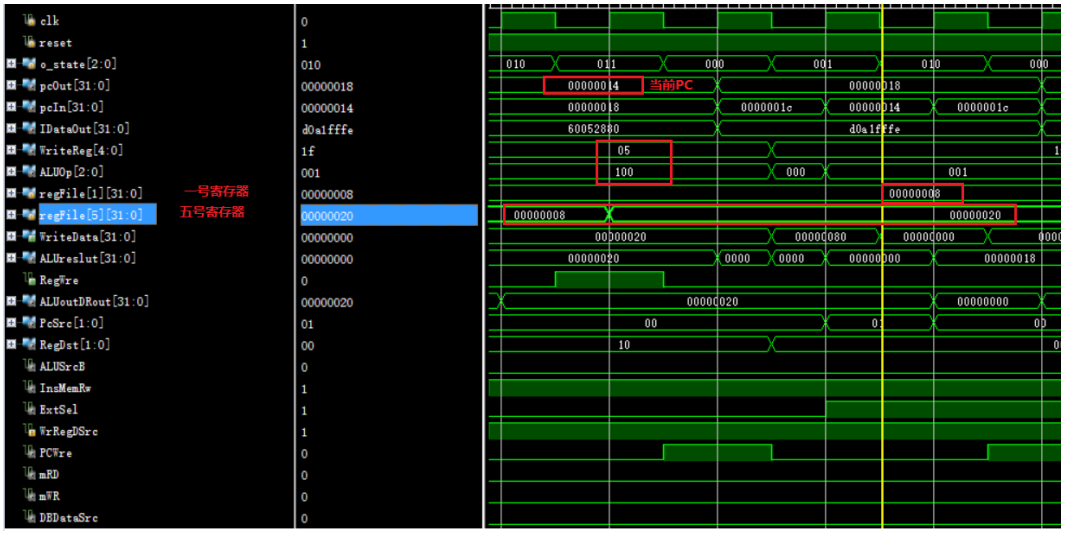
当前PC地址为0x00000014，五号寄存器内容为2，ALUop为100，表示左移位，ALU结果为8，五号寄存器结果变为8，指令执行正确。

指令：beq　$5,$1,-2(=,转14)



当前指令地址0x00000018，一号寄存器内容为 8，五号寄存器内容为8，比较五号寄存器和一号寄存器内容，ALUop为001，做减运算，因结果为0，代表两者相等，程序跳转当前地址-2处，下一指令地址为 0x00000014，结果正确。

再次执行指令：sll　$5,$5,2



当前PC地址为0x00000014，向左移2位，寄存器五号的内容从8变为16，结果正确。

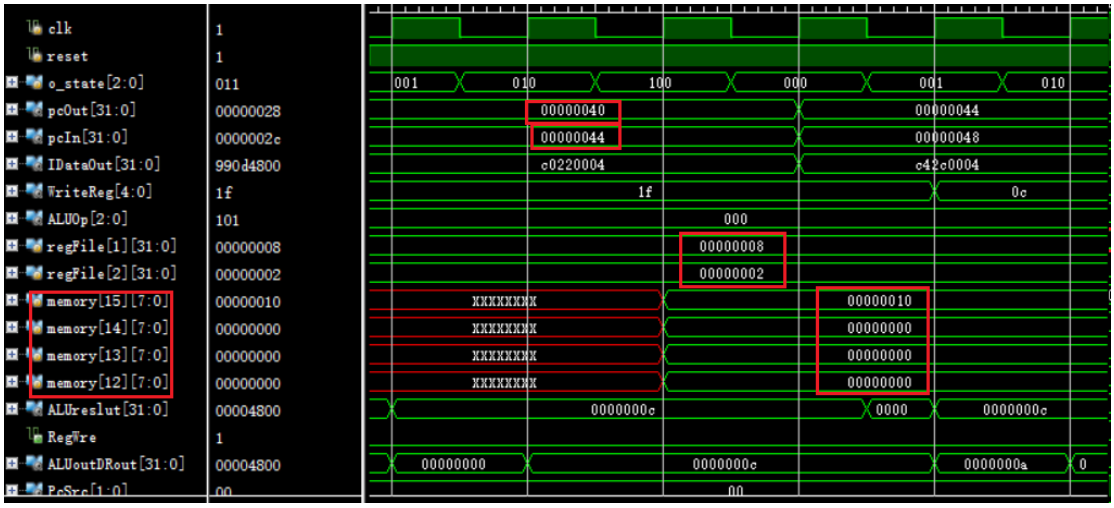
指令：beq　$5,$1,-2(=,转14)

当前PC地址为0x00000018，此时1号寄存器内容为8，5号寄存器内容为0x20，ALU执行减运算，两者不相等，结果为0x18,zero信号无效，因此下一PC地址为顺序执行的地址 0x0000001C。

指令：jal　0x0000040

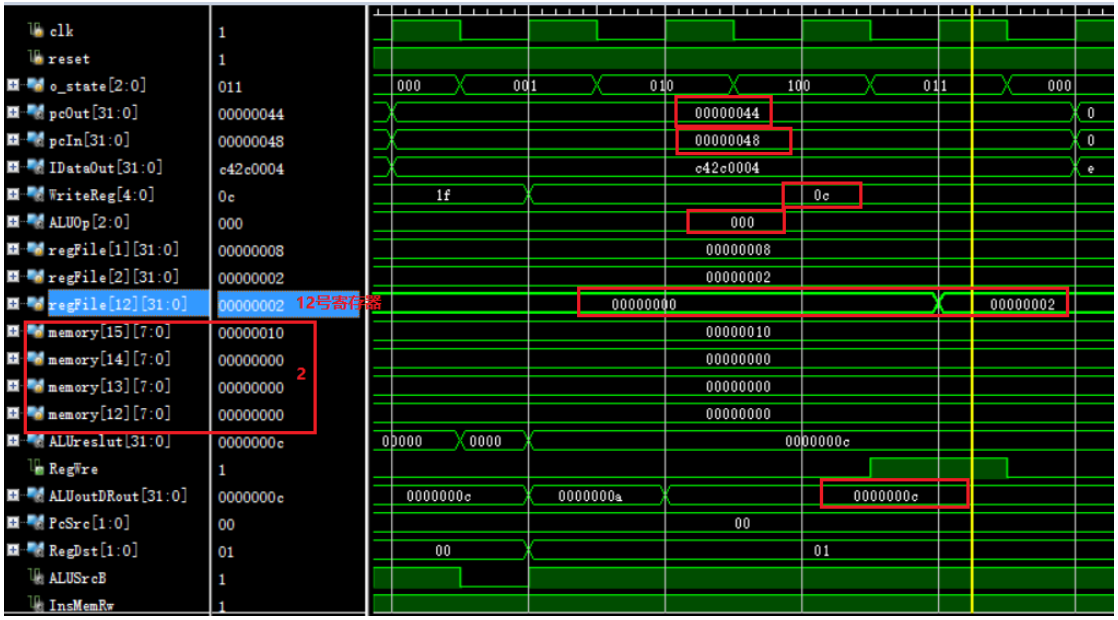
当前PC地址0x0000001C，跳转到0x40，因此下一PC地址为0x0000040，将PC+4保存到31号寄存器，31号寄存器内容从0变为0x20,结果正确。

指令：sw　$2,4($1)



当前PC地址0x00000040，下一PC地址0x00000044，指令执行顺序：IF->ID->EXE->MEM->IF,一号寄存器内容为8，二号寄存器为2，ALU运算结果为0xc，即将2写入存储器0xc的位置，memory[12]从0变为2。

指令：lw　$12,4($1)

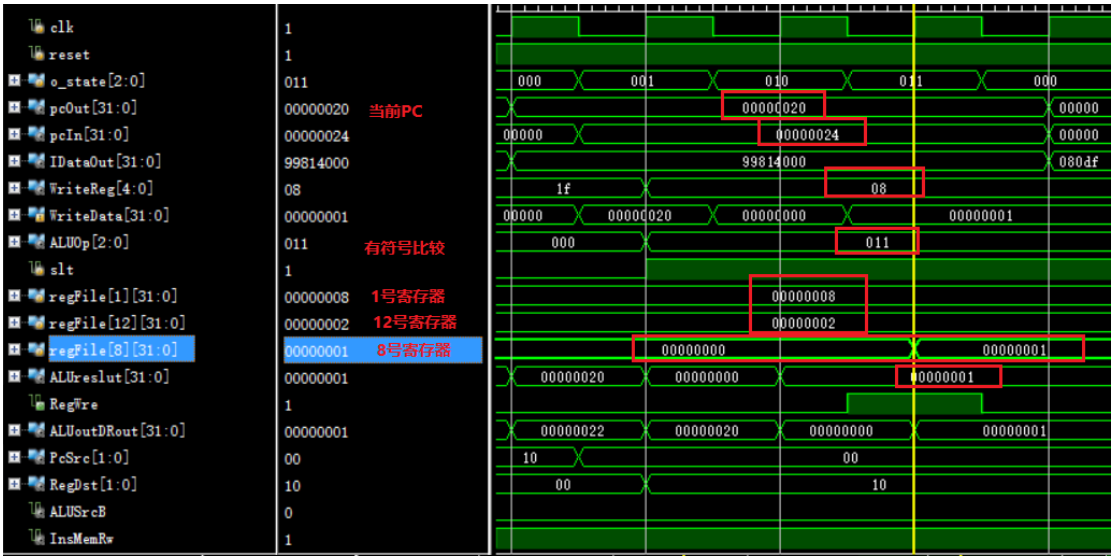


当前PC地址0x00000044，一号寄存器内容为8，ALU执行加运算，结果为0xc，因此把memory[12]内容写入12号寄存器，12号寄存器内容从变为2。

指令：jr　$31

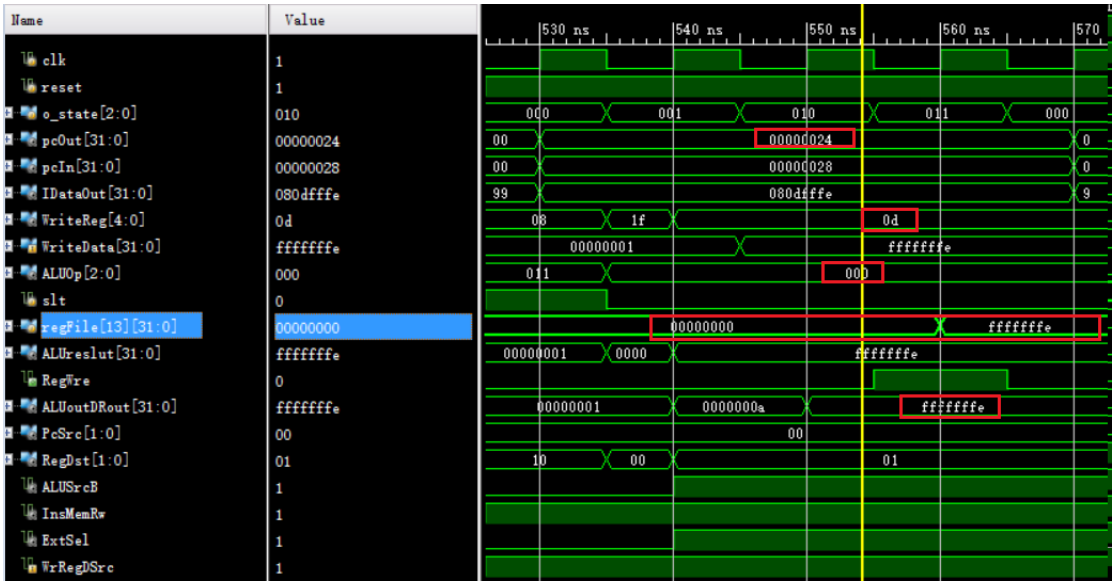
当前 0x00000048，31号寄存器内容为 0x20,下一PC地址为 0x20。指令执行顺序：IF->ID->IF

指令：slt　$8,$12,$1



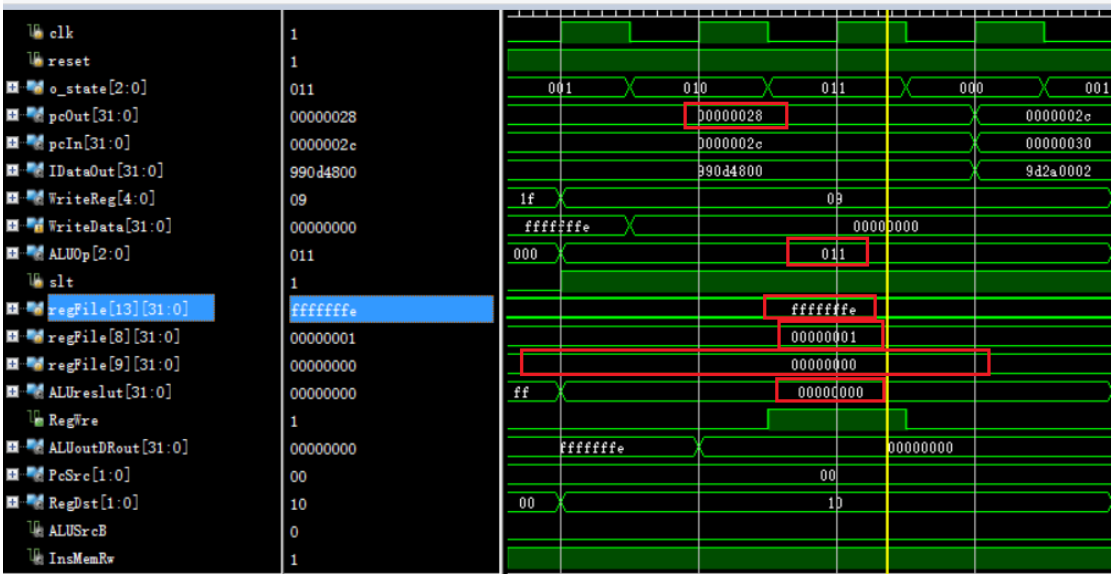
1号寄存器内容为8，12号寄存器内容为2，ALU进行无符号比较，2<8，因此结果为1，将1写入8号寄存器，8号寄存器内容从0变为1。

指令：addi　$13,$0,-2



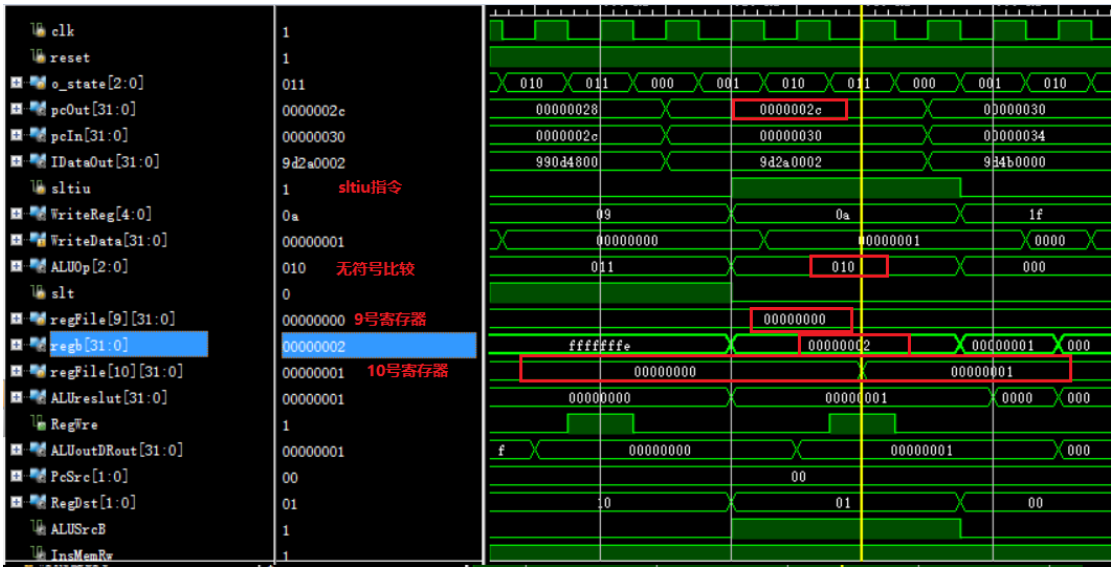
当前PC地址0x00000024，0号寄存器内容永远为0，ALUop 为000，执行加运算，结果为0xfffffffe,将结果写入13号寄存器，13号寄存器内容从0变为0xfffffffe

指令：slt　$9,$8,$13



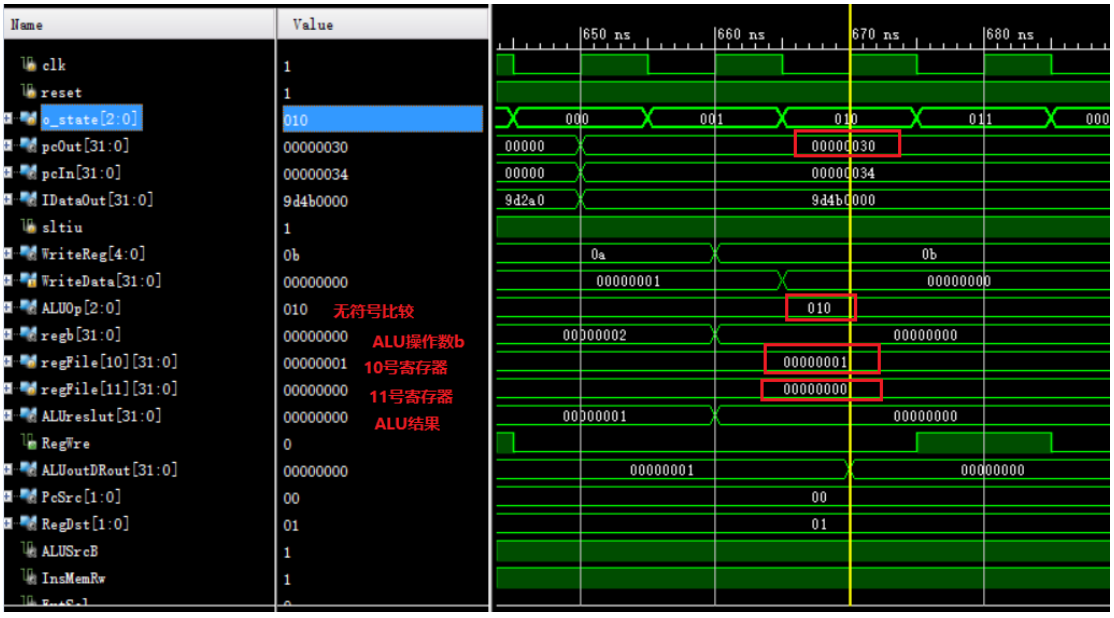
当前PC地址 0x00000028，ALUop 为011，进行有符号比较，8号寄存器内容为1，13号寄存器内容为0xfffffffe，是负数。8号寄存器数据大于13号寄存器数据。因此9号寄存器内容为 0。

指令：sltiu　$10,$9,2



当前PC为0x0000002C，9号寄存器内容为0，与立即数进行无符号比较，小于成立，因此ALU结果为1，将1写入10号寄存器，10号寄存器内容从0变为1.

指令：sltiu　$11,$10,0

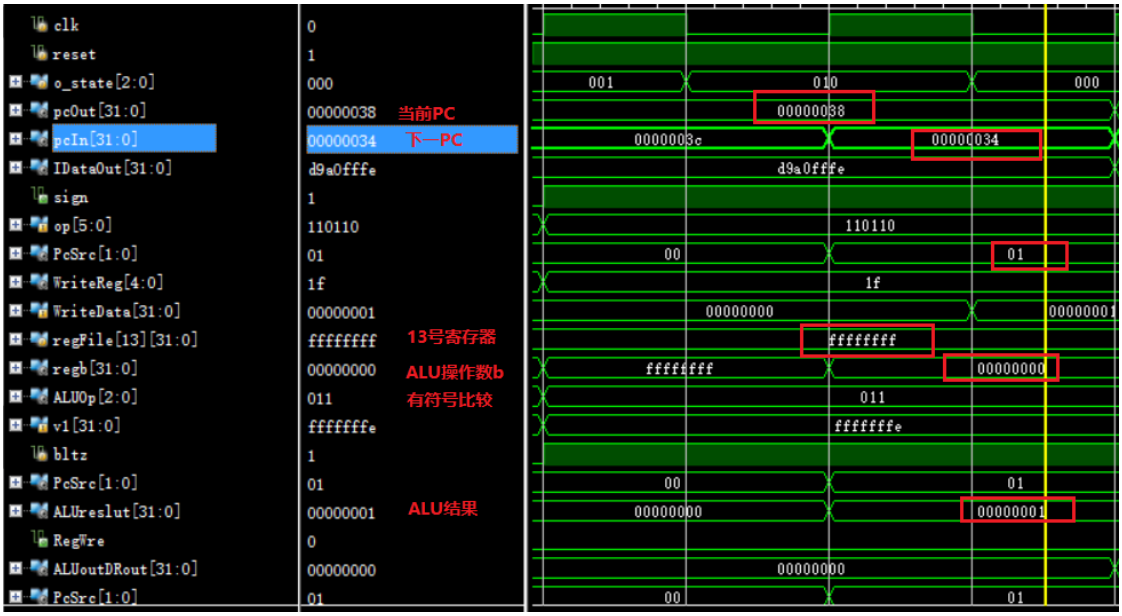


当前PC地址0x00000030，10号寄存器内容为1，与0进行无符号比较，大于0，因此ALU结果是0，将结果写入11号寄存器，11号寄存器内容仍为0.

指令：addi　$13,$13,1

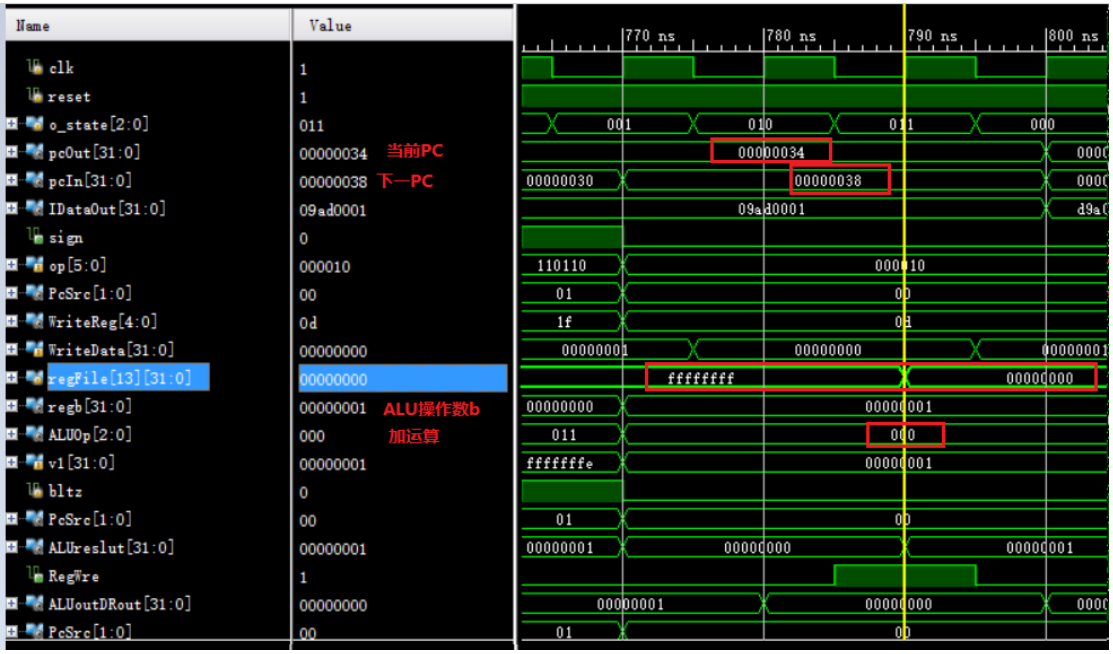
当前PC地址 0x00000034，13号寄存器内容为0xfffffffe,ALU进行加运算，结果为0xffffffff,写回13号寄存器。

指令：bltz $13,-2 (<0,转34)



当前PC地址0x00000038，13号寄存器内容为0xffffffff，小于0，因此ALU进行无符号比较时结果为1，因此PC将跳转到当前地址-2，即下一PC地址为0x00000034。

addi $13,$13,1

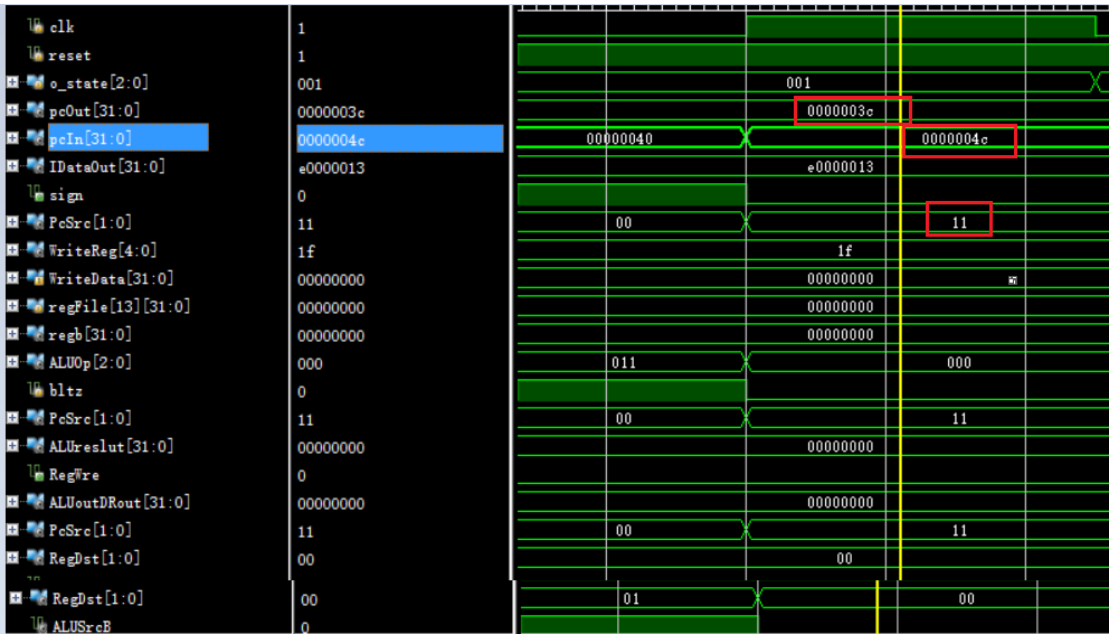


当前PC地址 0x00000034，13号寄存器内容与1进行加运算，结果为0，因此13号寄存器内容从 0xffffffff 变为0。

bltz $13,-2

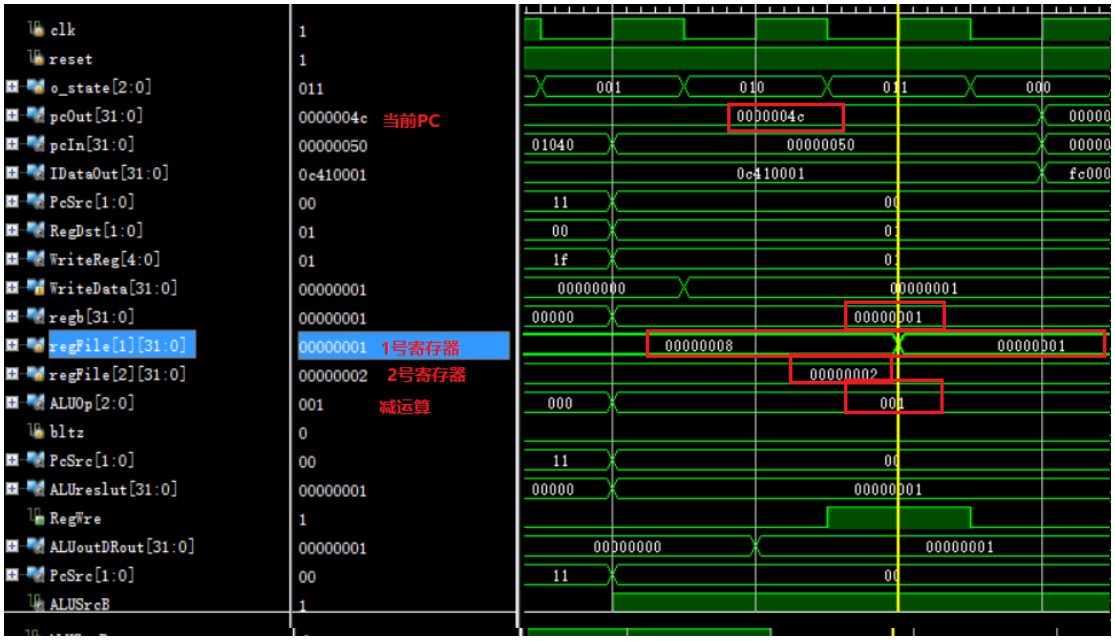
当前 PC 地址为 0x00000038，将 13 号寄存器内容与 0 进行比较，两者相等，因此 ALU 结果为 0，PcSrc 结果为 0，因此顺序执行，下一 PC 地址为 PC+4，即 0x0000003C

指令：j　0x000004C



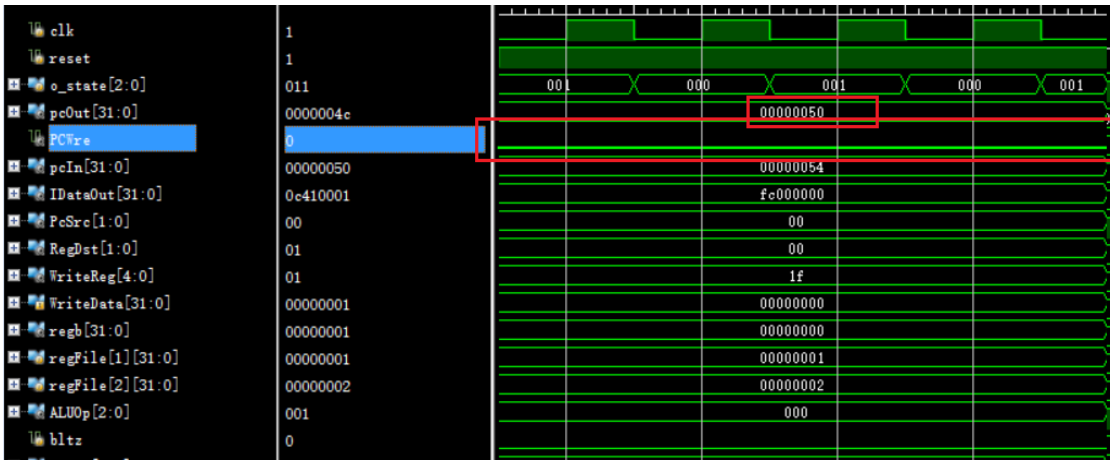
当前PC地址为0x0000003C，PcSrc 为3,因此下一PC地址为拓展后的地址，即0x00000040。

指令：subi　$1,$2,1



当前PC为0x0000004C，2号寄存器内容为2，与1进行减运算，ALU结果为1，将结果写入1号寄存器，1号寄存器内容

指令：halt



当前PC地址为0x00000050，PCWre为0并且不再变化，程序停止。

六. 实验心得

有了单周期的基础，本次实验容易上手，但是还是会遇到很多问题。

1.时钟触发问题

需要时钟触发的模块需要分清楚是上升触发还是下降沿触发，否则会导致不能正确读取指令，或者因为时钟延迟问题导致写入寄存器的数据 ALU 运算得到的新数据。

2.连线问题

在顶层模块中，如果给一个模块少传参，vivado 不会报错，甚至传入一个未定义的变量也不会报错。因此需要自己特别注意每个变量一一对应，否则就会导致信号出错。

3.跳转指令

执行 jal 0x0000040 指令（0x0000001C）时将 PC+4（0x00000020）保存到31寄存器，保存好了后，因为 PC 上升变化，state 下降沿变化，IR 上升沿触发，所以指令 jal 后，读取下一条指令，时钟下降沿先到来，state 先变化为 001，而IR需要等待上升沿，opcode 在经过IR后才会变化。因此当state为001时有一段时间opcode没有变化，仍然是jal，CPU得知是jal指令就会将 PC+4（0x00000044）保存到31号寄存器。在测试指令中，程序会不断在

0x00000040-0x00000048之间循环。类似的指令有j,jal,jr,halt，这四个指令经过ID就回到IF获取新的指令，而此时如果opcode没有及时变化，就会出错，控制单元中的op依然是前一指令的op。

所以在 state 为0时，将这四条指令全部置为0解决。这个问题也说明了需要设计好模块上升沿或下降沿触发的重要性。

4.执行连续执行两条相同（或操作码相同）的指令

在所给的测试指令中，有连续执行 sltiu 指令，虽然不影响指令执行的结果，但是如果细致地看每一个控制信号就会发现问题，因为在控制单元中是根据op改变而改变控制信号，而问题3中提到，会在 state 为0时将一些标志变量重置为0，这个时候因为op没有改变，对应的标志变量就不会变为 1，仍然是0，因此将控制单元改为PC改变触发控制信号的改变。而我也意识到单周期 CPU 同样这个问题…

除此之外， 还有很多细节的问题，比如控制信号的赋值，ALU 中移位的时候是 regb<<rega 等等，通过仔细分析波形图，结合数据通路都能找到问题所在处后解决。

本次实验一如既往地花时间，尤其是一条条指令分析波形的时候，略显枯燥。虽然每次改一个 bug 后十分开心，但是 bug 太多看不到尽头的时候便让人丝毫提不起精神。改代码->仿真->分析波形->找到错误->改代码。如此反复循环，最后终于完成了，回过头看好像也不是很难，只要细心一点就好了，单周期 CPU 也是这么想的。知道实验要细心，但是还是不可避免地犯错误。当然，经过了这么多，也有了很多收获，对于单周期和多周期的设计以及代码实现有了更清晰的了解，对于 verilog 语言和 vivado 更加熟悉，能更加熟练利用波形分析错误。

课程总结:

本门课程有三个大实验,汇编程序、单周期 CPU、多周期 CPU。不管是汇编还是 CPU，一开始都会出现很多小错误，汇编一步步查看每条指令的执行和寄存器的内容找错误，CPU 也类似，查看每一步的波形、变量的值分析错误。这方面还是非常相似的。不同的部分，CPU 比汇编更加复杂，涉及到更多的变量、更复杂的逻辑，运用到模块化的思想。虽然汇编中也会将代码分成几部分，但是不像CPU的要求更高，规模更大。

本门课程提高了我的动手能力，加深了对与计算机组成原理尤其是CPU 的理解，并且熟练使用 vivado 和 编写 verilog 代码。