计算机组成原理实验2---单周期CPU

原创

MJ-GOD 于 2018-09-25 22:08:06 发布 7039 收藏 60
分类专栏： Vivado 文章标签： CPU Vivado

实验目的

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

掌握单周期CPU的实现方法。

实验内容

实验的具体内容与要求。

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）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

（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。

==>跳转指令

（13）j addr

111000	addr[27..2]

功能：pc <一{(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

（14）halt

111111	00000000000000000000000000(26位)

功能：停机；不改变PC的值，PC保持不变。

自己添加的指令：

==>逻辑指令

（15）xor rd , rs , rt

010011	rs(5位)	rt(5位)	rd(5位)	reserved

功能：rd←rs ^ rt；逻辑异或运算。跟前面的or操作是类似的，只不过因为没有调用异或的操作所以就添加了。

==>移位指令

（16）sllv rd, rt, rs

011001	rs(5位)	rt(5位)	rd(5位)	sa	reserved

功能：rd<－rt<<rs, 左移rs位，

说明：这个操作跟前面的sll是类似的，主要不同在于不是I型指令而是R型指令。

==>比较指令

（17）sltu rt, rs,rt 不带符号比较

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

功能：if (rs <rt) rd =1 else rd=0

说明：这个指令跟前面的带符号比较slti是类似的，因为老师要求的指令没有调用无符号比较，所以就多写了这一个指令。

==>寄存器写指令(左移操作)

（18）lui rt, immediate

011011	未利用rs（10000）	rt(5位)	**immediate**(16位)

功能：rt = {immediate , 16'b0000000000000000 };

说明：因为这一个指令是I型指令，但是这里的rs没有被利用，而这个写寄存器指令实际上是对立即数进行左移的操作。也就是把立即数放在寄存器的高16位。所以这里实现的方法就是利用左移操作实现的。

必须写一段测试用的汇编程序，而且必须包含所要求的所有指令，slti指令必须检查两种情况："小于"和"大于等于"；beq、bne："不等"和"等"。这段汇编程序必须尽量优化且出现在实验报告中，同时，给出每条指令在内存中的地址。检查实验时，必须提供。

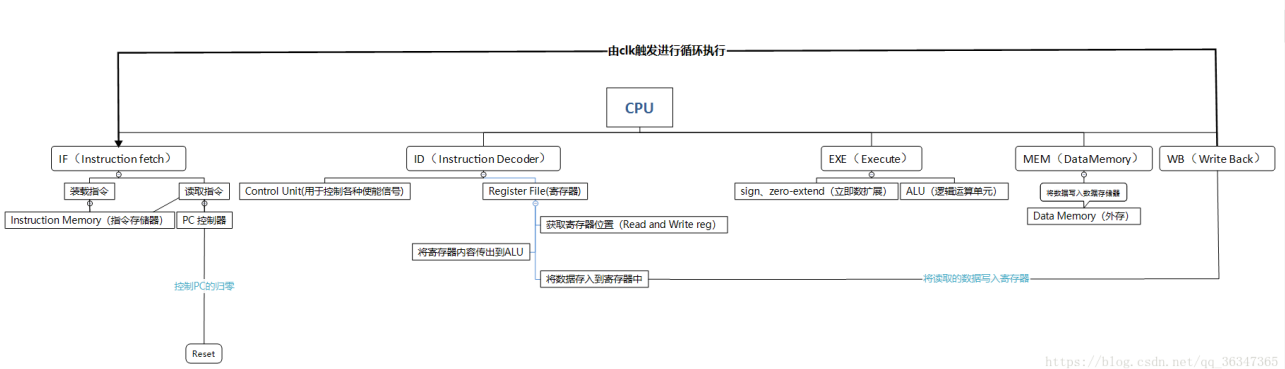
其他基本要求：

1. PC和寄存器组写状态使用时钟触发。
2. 指令存储器和数据存储器存储单元宽度一律使用8位，即一个字节存储单位。
3. 控制器部分要学会用控制信号真值表方法分析问题并写出逻辑表达式；或者用case语句方法逐个产生各指令控制信号。注意：控制信号的产生不能使用时钟触发！
4. Always@（...）的敏感信号表中，时序触发和电平触发不能同时出现，即不能混用。

简述实验原理和方法，必须有数据通路图及相关图。

实验原理：

1. 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。时钟信号是驱动CPU运行的条件以及动力。
2. CPU在处理指令的时候，大致分为IF->ID->EXE->MEM->WB这五个步骤的循环执行。根据老师所给的图片可以对每个模块根据这五个环节作为依据划分五个部分。



1. 如下：
2. 每一个步骤的解释以及相关的模块操作：

取指令(**IF**)跟PC计数器以及指令存储器有关：在程序刚开始执行的时候需要将Reset置为0从而使PC计数器重置为0。而后在将Reset置为1，接着指令存储器根据程序计数器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。

指令译码(**ID**)跟控制单元有关：ControlUnit会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，从而产生相应的控制后面操作的模块的操作控制信号，用于驱动执行状态中的各种操作。

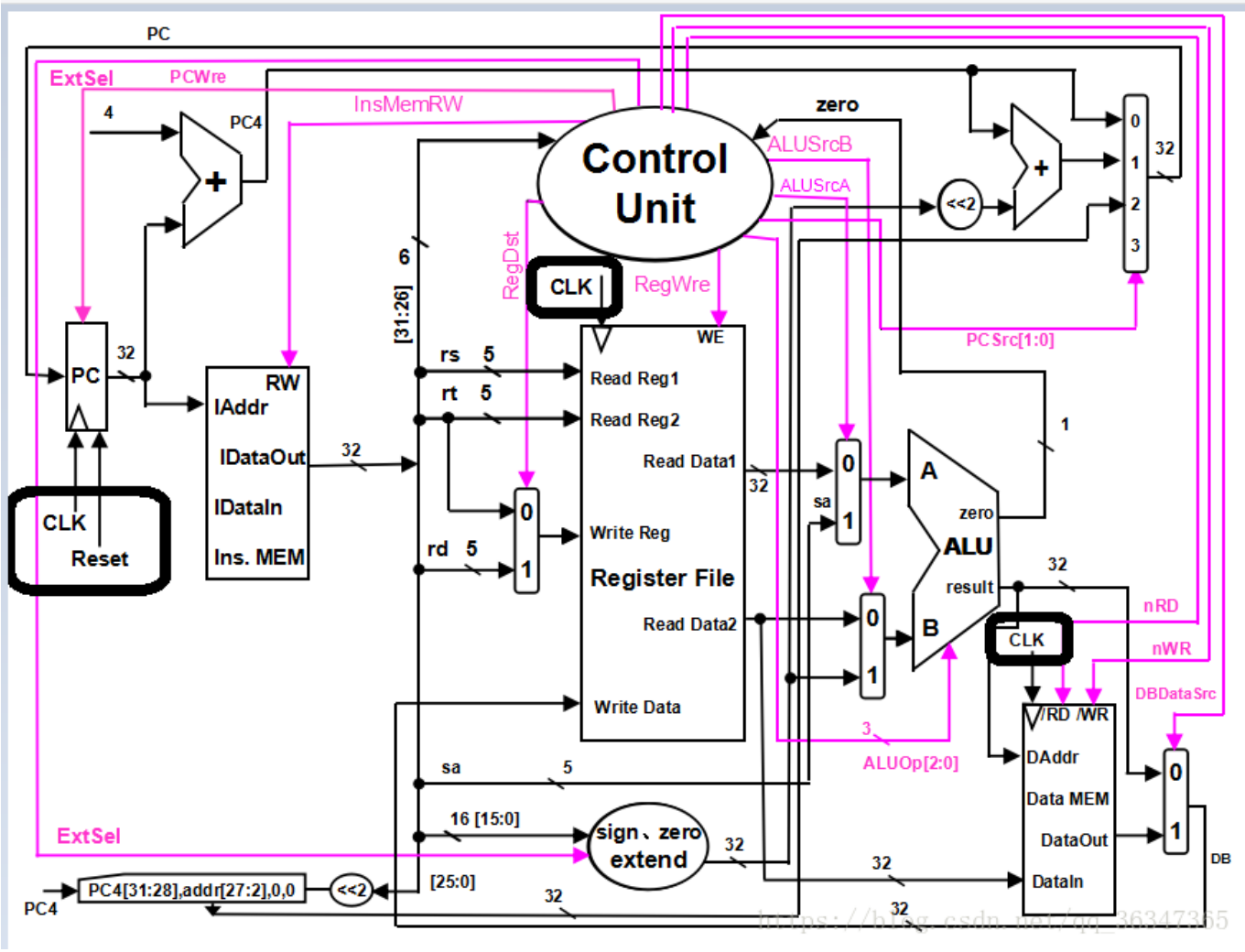
指令执行(**EXE**)跟寄存器以及ALU有关：根据控制单元所产生的操作控制信号进行数据的加载或者寄存器中数据的访问以及对应的逻辑运算。

存储器访问(**MEM**)跟数据存储区有关：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。

结果写回(**WB**)跟寄存器堆有关：将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。

设计思路：

根据老师所提供的单周期CPU的数据通路和控制路线图



我们可以看到CPU实际上是有很多个模块所构成的，而顶层CPU，即是所有模块的总和所构成的部件，所需要输入的外部数据为：

CLK（时钟信号）以及Reset（用于PC的清零）

所以顶层模块我们就可以简化为如下：

```
module s_cpu(
    //时钟信号
    input clk,
    //控制信号
    input Reset
    );
    //以下为各个模块
endmodule
```

而接下来所需要完成的就是每一个模块的划分以及设计工作了。

3. 设计模块过程：

1. 划分模块：

根据老师所给的资料以及单周期CPU的数据通路和控制线路图，然后结合上面的CPU在一个时钟周期内五个阶段的划分，我们可以将单周期CPU的数据通路和控制线路图划分为一下七大模块：

分别是：PC模块（用于程序计数器的控制，包含跳转指令，即左下角未被包含的那一部分）

Instructions Memory模块（用于存储指令以及读取指令）

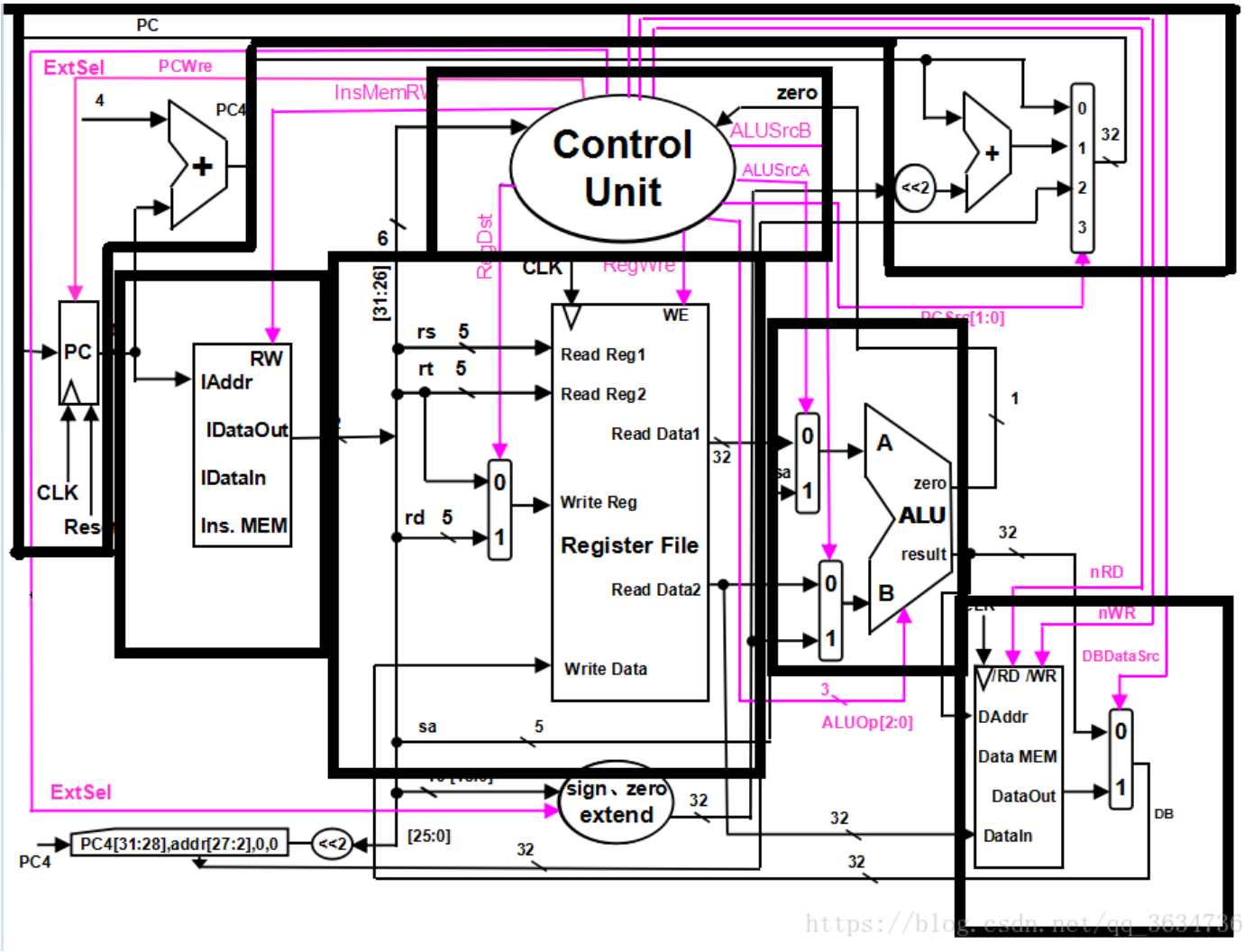
ControlUnit模块（用于ID，即指令的解析然后生成相应的对各个模块的控制信号）

RegisterFile模块（用于寄存器堆的模拟：读取寄存器数据以及将运算结果或者数据存储器中的数据装载到寄存器中）

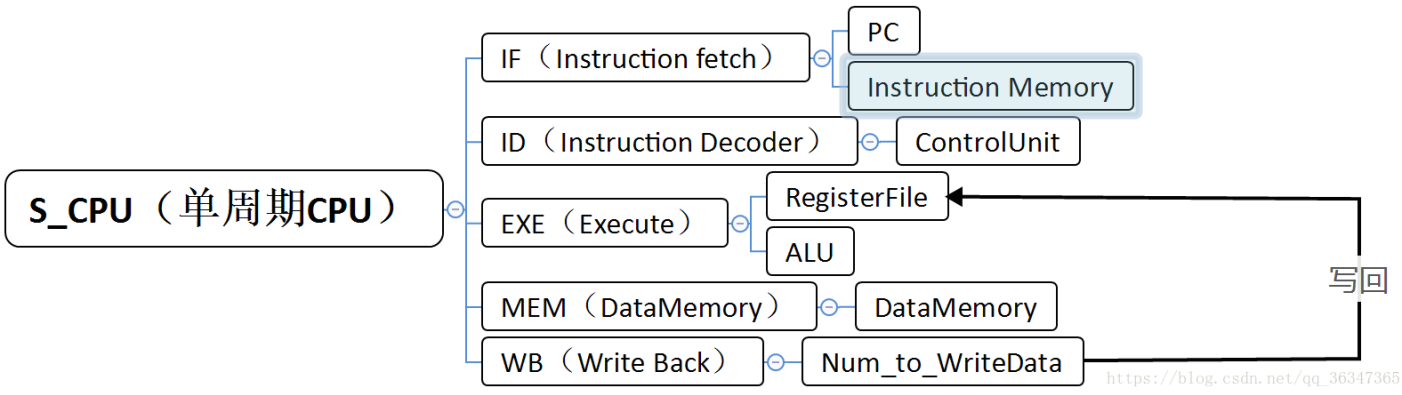
ALU模块（用于从寄存器中以及指令中的数据的逻辑处理）

DataMemory模块（充当外存，用于模拟数据的存储以及读取）

Num_to_WriteData模块（选择ALU的运算结果以及数据存储器的读取数值存到寄存器中，WB过程）



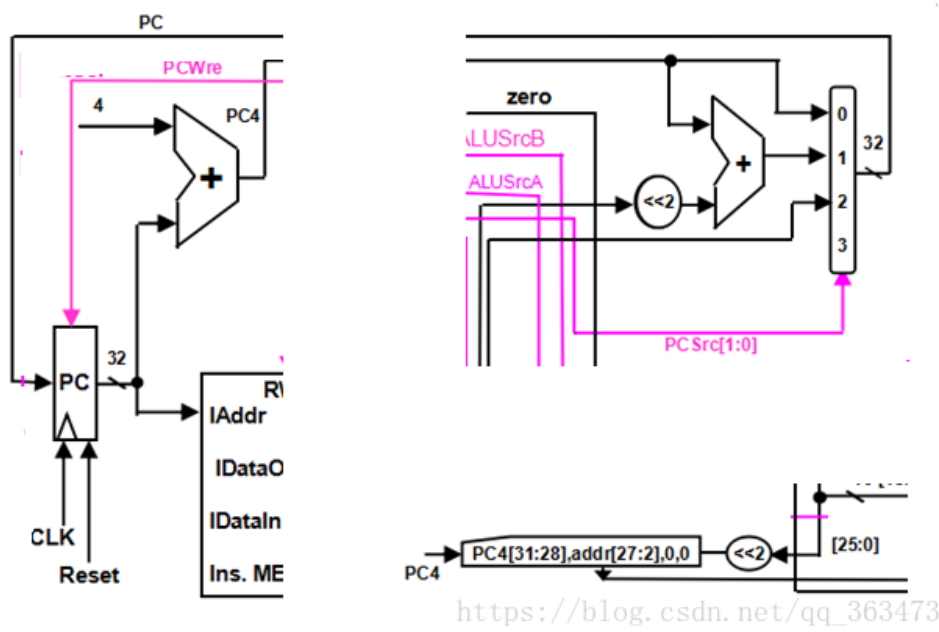
用树状图分工如下（分工依据是五个步骤）：



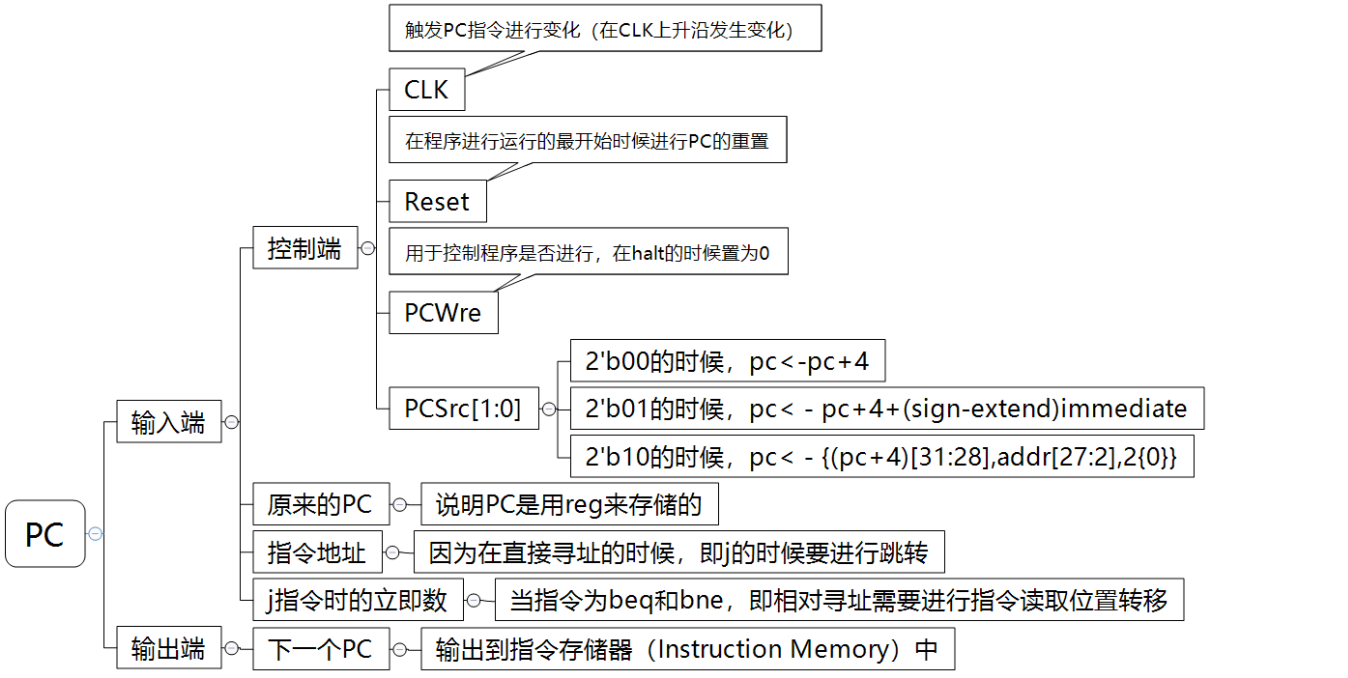
1. 根据划分的模块我们可以来开始工作了，首先便是我们的PC模块

1. PC模块的实现：

首先先截一个小图



根据这一个图可以得到PC的设计如下：

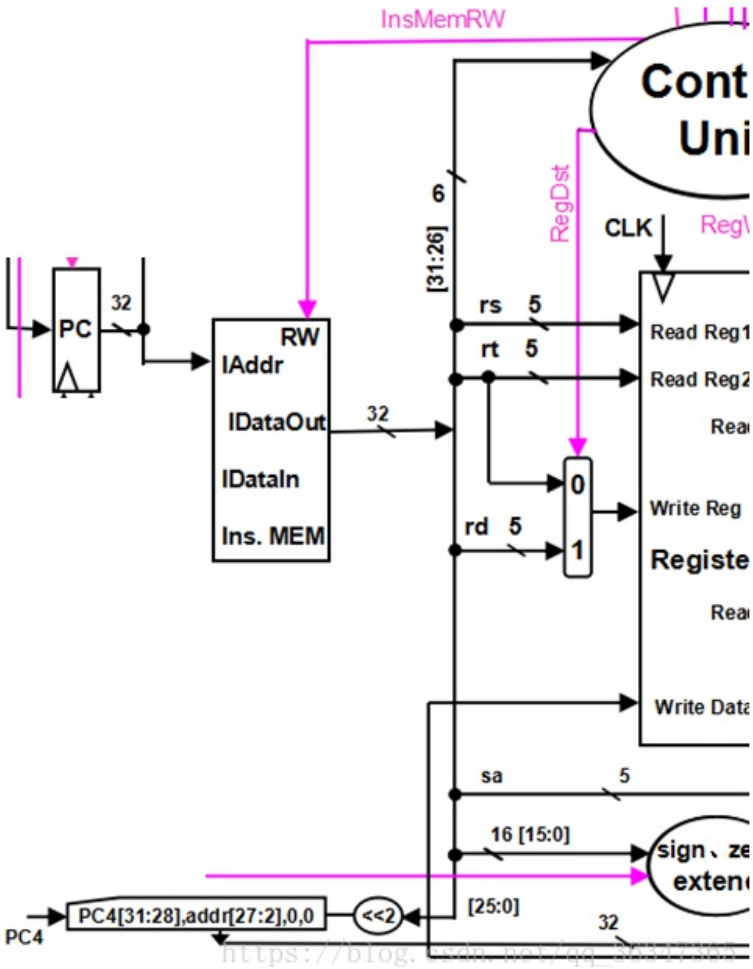


PC中主要是PCSrc（用于控制PC的变化，+4、间接寻址、以及直接寻址）三种情况的判断，同时触发条件是CLK上升沿以及Reset的下降沿时触发，关键代码如下：

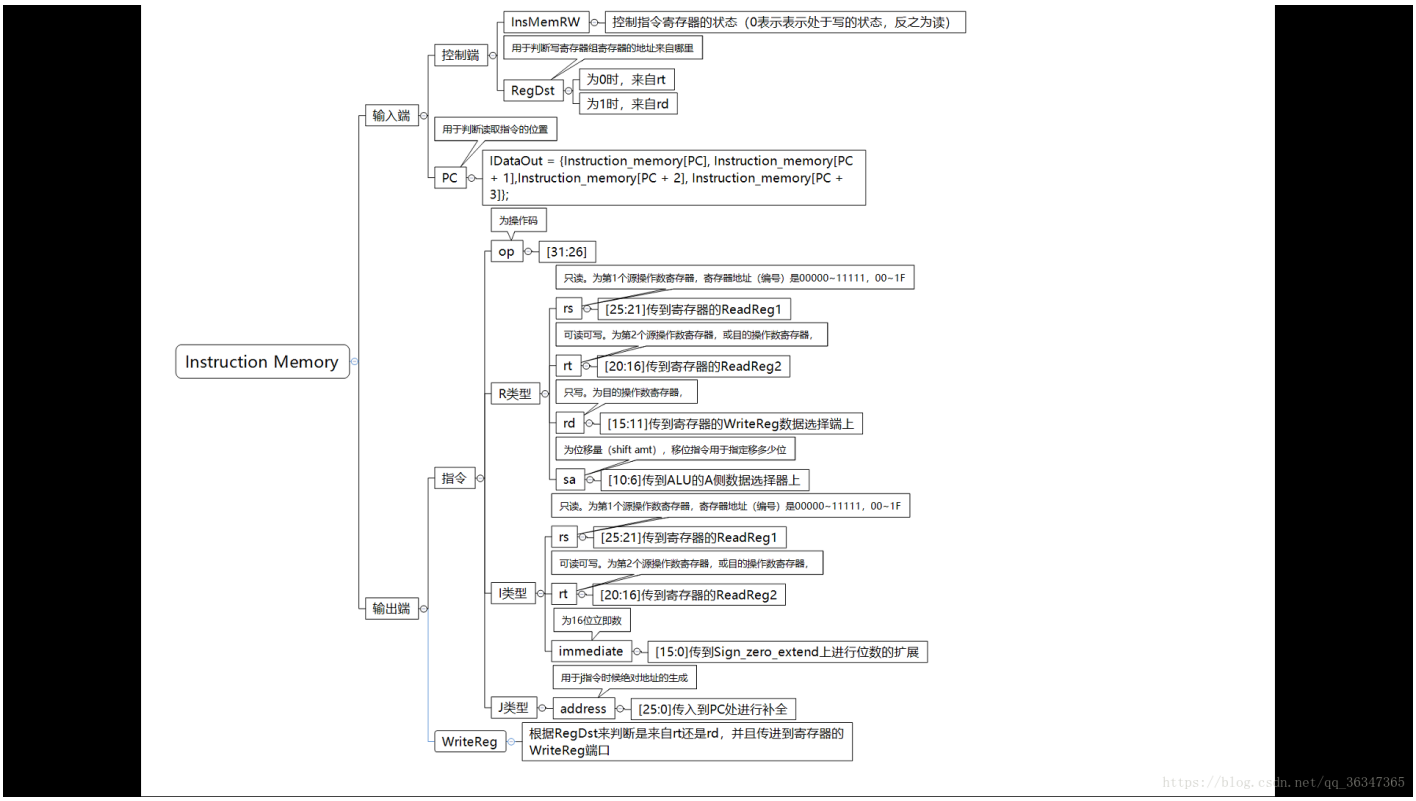
```
always @(posedge CLK or negedge Reset)begin
//在时钟的上升沿0->1以及Reset的下降沿的时候触发从1->0
    if(Reset == 0)
    PC = 0;
    else if(PCWre == 1) begin
        if(PCSrc == 2'b00) //当PCSrc为2'b00的时候PC = PC + 4
        PC = PC + 4;
        else if(PCSrc == 2'b01)//当PCSrc为2'b01的时候PC = pc+4+immediate*4
        PC = PC + 4 + immediate * 4;
        else begin //当PCSrc为2'b10的时候PC = {(pc+4)[31:28],addr[27:2],2{0}}
        PC = PC + 4;
        PC = { PC[31:28], address[25 : 0], 2'b00};
        end
    end
end
```


2. Instruction Memory模块的实现：

首先截一个小图：



根据这一个图可以得到Instruction Memory的设计如下：



另外，因为这是一个指令存储器，所以在存储器内部需要额外开辟一部分空间laicunchu指令，我这里设置的是在程序一开始运行就进行指令的装载。代码如下：

```
reg [7:0] Instruction_memory [0:60];
initial begin
$readmemb("E:/vivado/S_CPU/test.txt", Instruction_memory);//读取文件（
end
```


然而在写程序的时候并不需要返回那么多的参数，因为在顶层模块的时候返回的IDataOut（Instruction，即对应的指令）可以根据需要进行对应的取值，比如需要rt，只要传入DataOut[20:16]即可，但是当时因为一直在纠结这一个模块的设置，所以就还是返回了几个参数，op以及rs、rt和rd

```
output reg [31:0] IDataOut,          //指令
output reg [4:0] WriteReg,           //WriteReg是RegisterFile的写数据端的WriteReg
output reg [4:0] ReadReg1, ReadReg2,//ReadReg1 即是rs，ReadReg2即是rt
output reg [5:0] op
```


其中核心代码如下：

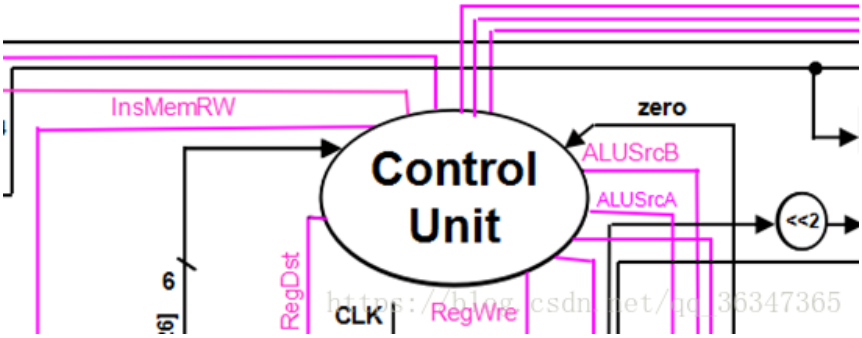
取指令：

```
always @ (PC or InsMemRW) begin
        IDataOut = {Instruction_memory[PC], Instruction_memory[PC + 1],Instruction_memory[PC + 2], Instruction_memory[PC + 3]}; //取出指令
        $display("Instruction is %b",IDataOut);
        assign op = IDataOut[31:26];
        assign ReadReg1 = IDataOut[25:21];
        assign ReadReg2 = IDataOut[20:16];
end
```

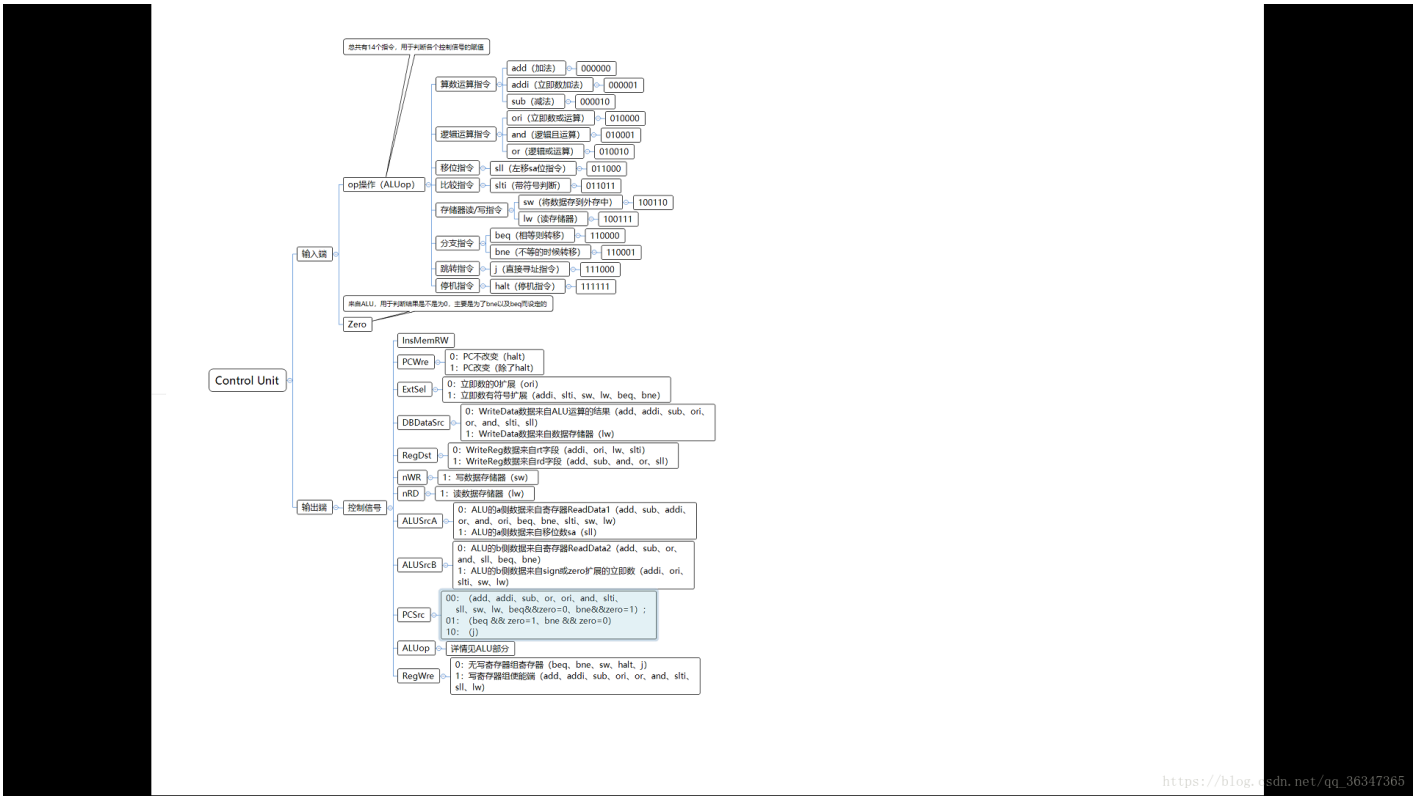

还有一个是将RegisterFile的WriteReg端口的数据选择器添加到了此处（因为先前考虑不周，捂脸）

```
always @(RegDst) begin       //RegDst用于判断WriteReg是sa还是rd
    if(RegDst == 0)
        assign WriteReg = IDataOut[20:16];
    else    assign WriteReg = IDataOut[15:11];
end
```

3.现在有了op了，根据上面的五个步骤，前面的IF步骤已经完成了，所以接下来就是ID过程了，也就是ControlUnit模块的实现了。继续按照前面的做法，截个图。（这一个部分觉得长的看起来似乎不是很难，但是却是比较复杂的）



从外观看起来我们可以看到这一个ControlUnit有两个输入，分别是Zero（来自ALU逻辑处理器）以及OP（来自指令存储器的指令高6位），以及有很多的输出，可能一开始还以为这一个很好写，但是看到这么多的输出的时候，你就应该意识到自己想多了，这一个很复杂。接下来就是根据老师所给的文档列出这一个模块的信息了。（超多）



一开始我使用的是case语句，所以就需要进行控制信号表的罗列，如下：

	Reset	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	nRD	nWR	RegDst	ExtSel	PCSrc	ALUOp
add	1	1	0	0	0	1	X	X	1	X	2'b00	3b'000
addi	1	1	0	1	0	1	X	X	0	1	2'b00	3b'000
sub	1	1	0	0	0	1	X	X	1	X	2'b00	3b'010
ori	1	1	0	1	0	1	X	X	0	0	2'b00	3b'011

and	1	1	0	0	0	1	X	X	1	X	2'b00	3b'100
or	1	1	0	0	0	1	X	X	1	X	2'b00	3b'011
sll	1	1	1	0	0	1	X	X	1	X	2'b00	3b'010
slti	1	1	0	1	0	1	X	X	0	1	2'b00	3b'110
sw	1	1	0	1	X	0	X	1	X	1	2'b00	3b'000
lw	1	1	0	1	1	1	1	X	0	1	2'b00	3b'000
beq	1	1	0	0	X	0	X	X	X	1	2'b01	3b'001
bne	1	1	0	0	X	0	X	X	X	1	2'b01	3b'001
j	1	1	X	X	X	0	X	X	X	X	2'b10	X
halt	1	0	X	X	X	0	X	X	X	X	X	X
xor	1	1	0	0	0	1	X	X	1	X	2'b00	3b'111
sltu	1	1	0	0	0	1	X	X	1	X	2'b00	3b'101
sllv	1	1	0	0	0	1	X	X	1	X	2'b00	3b'010
lui	1	1	0	1	0	1	X	X	0	1	2'b00	3b'010

(其中填写X的地方表示该处取值可以为任意值)

上面的方法消耗的时间会比较大，而且还有写很多的case语句进行op的判断，每一个case语句中还有很长的赋值操作。所以后来在同学的提醒下直接用赋值语句就好了。直接利用老师提供的表格，进行赋值。

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、xor、sltu、sllv、lui	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、xor、sltu、sllv	来自sign或zero扩展的立即数，相关指令：addi、ori、slti、sw、lw、lui
DBDataSrc	来自ALU运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、sll、xor、sltu、sllv、lui	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、xor、sltu、sllv、lui
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
nRD	输出高阻态	**读数据存储器，相关指令：lw**
nWR	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、lui	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、xor、sltu、sllv
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、lui
PCSrc[1..0]	00：pc<—pc+4，相关指令：add、addi、sub、or、ori、and、slti、xor、sltu、sllv、lui 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	

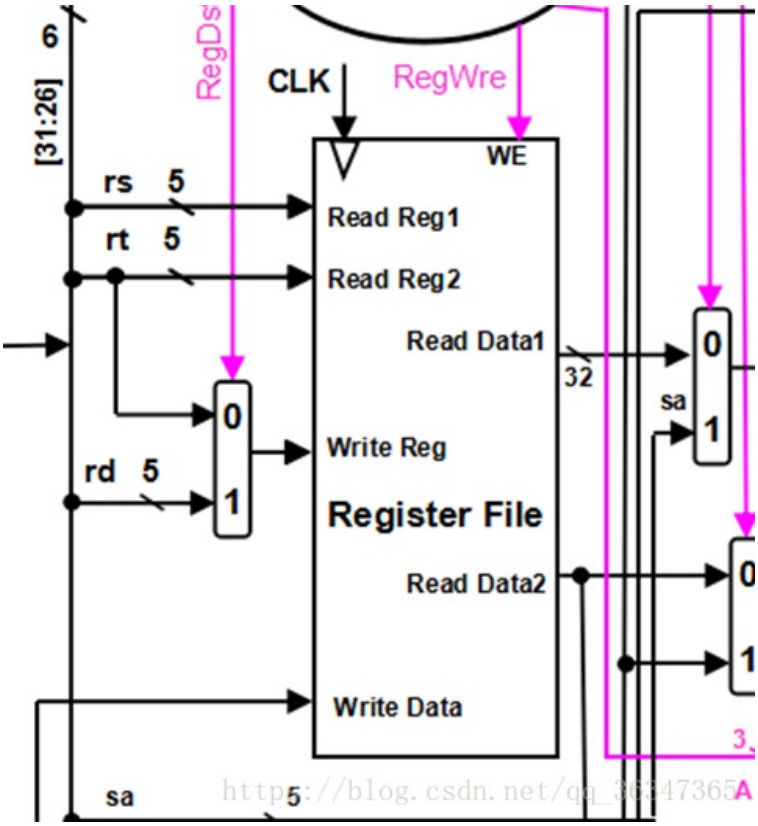
因为很多地方需要用到对op类型的判断，所以就将add的14个操作的op当作常量来处理，代码如下：

```
    //将14个操作当作常量
    parameter    add =6'b000000, addi =6'b000001,
                 sub =6'b000010, ori  =6'b010000,
                 And =6'b010001, Or   =6'b010010,
                 sll =6'b011000, slti =6'b011011,
                 sw  =6'b100110, lw   =6'b100111,
                 beq =6'b110000, bne  =6'b110001,
                 j   =6'b111000, halt =6'b111111;
```

赋值语句如下：（根据表格中情况分类进行赋值就行，这种方法比较无脑且简单）

```
always@(op or Zero) begin
    assign PCWre = (op == halt) ? 0 : 1;
    assign ALUSrcA = (op == sll) ? 1 : 0;
    assign ALUSrcB = ((op == addi) || (op == ori) || (op == slti)|| (op == sw) || (op == lw)) ? 1 : 0;
    assign DBDataSrc = (op == lw)? 1 : 0;
    assign RegWre = ((op == beq) || (op == bne) || (op == sw) || (op == halt)) ? 0 : 1;
    assign nRD = (op == lw) ? 1 : 0;
    assign nWR = (op == sw) ? 1 : 0;
    assign RegDst = ((op == addi) || (op == ori) || (op == lw) || (op == slti)) ? 0: 1;
    assign ExtSel = (op == ori) ? 0 : 1;
    if(op == j) assign PCSrc = 2'b10;
    else if(((op == beq) && (Zero == 1)) || ((op == bne) && (Zero == 0))) assign PCSrc = 2'b01;
    else assign PCSrc = 2'b00;

    if(op == add || op == addi)
    assign ALUOp = 3'b000;
    else if(op == sub || op == beq || op == bne)
    assign ALUOp = 3'b001;
    else if(op == sll)
    assign ALUOp = 3'b010;
    else if(op == Or || op == ori)
    assign ALUOp = 3'b011;
    else if(op == And)
    assign ALUOp = 3'b100;
    else if(op == slti)
    assign ALUOp = 3'b110;
end
```


4.到了这一步，算是完成了ID步骤了，接着就是EXE步骤了，现在主要是来实现RegisterFile这一个模块，即寄存器堆。首先先截一个小图：



可以看到这里的主要操作步骤是根据ReadReg和WriteReg进行寄存器中数据的读取以及写入。从而我们可以得到Register的结构如下：

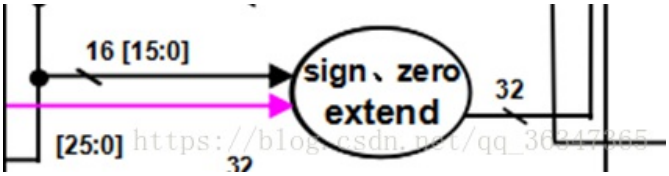
因为RegisterFile也是存放东西的模块，所以在实现的时候跟Instruction Memory一样是利用reg来实现的，代码如下：

```
reg [31:0] regFile[1:31]; // 定义32个寄存器
```

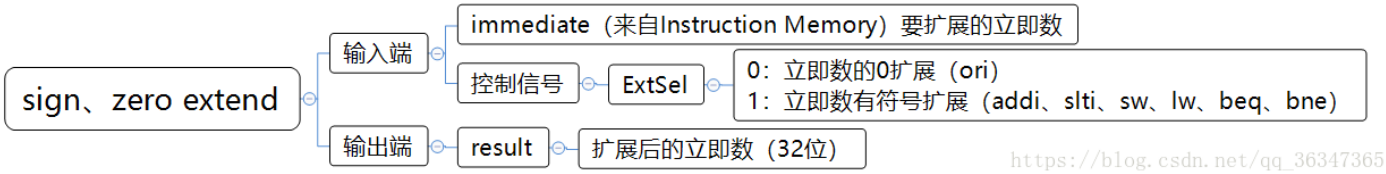
然后就是根据RegWre的状态判断是否进行写的操作，以及利用ReadReg作为下表索引返回对应的数值。代码如下：

```
assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1]; //
读寄存器数据
assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
always @ (negedge CLK or negedge Reset) begin // 必须用时钟边沿触
    if (Reset==0) begin
        for(i=1;i<32;i=i+1)
        regFile[i] <= 0;
    end
    else if(RegWre == 1 && WriteReg != 0) begin// WriteReg !=
    0, 0号寄存器不能修改
        regFile[WriteReg] <= WriteData; // 写寄存器
            $display("Instruction is %b",regFile[1]);
    end
end
```


5.接下来就是ALU逻辑运算器的实现了，在此之前先掺入一个小片段，那就是立即数的扩展，因为立即数在ALU以及前面的PC中需要用到，（前面忘了讲了。。。）先截一个小图：



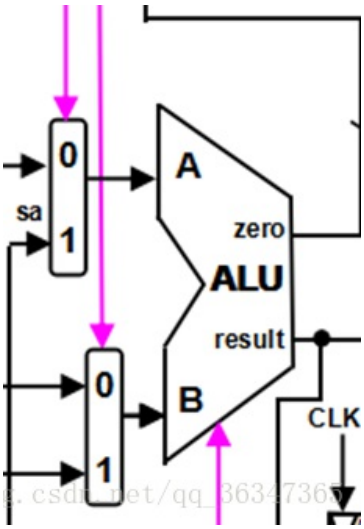
这一部分的逻辑比较简单，因为是否为有符号扩展主要取决于控制信号，而符号位的获取就是15号元素是否为1或0的一个判断而已，所以结构如下：



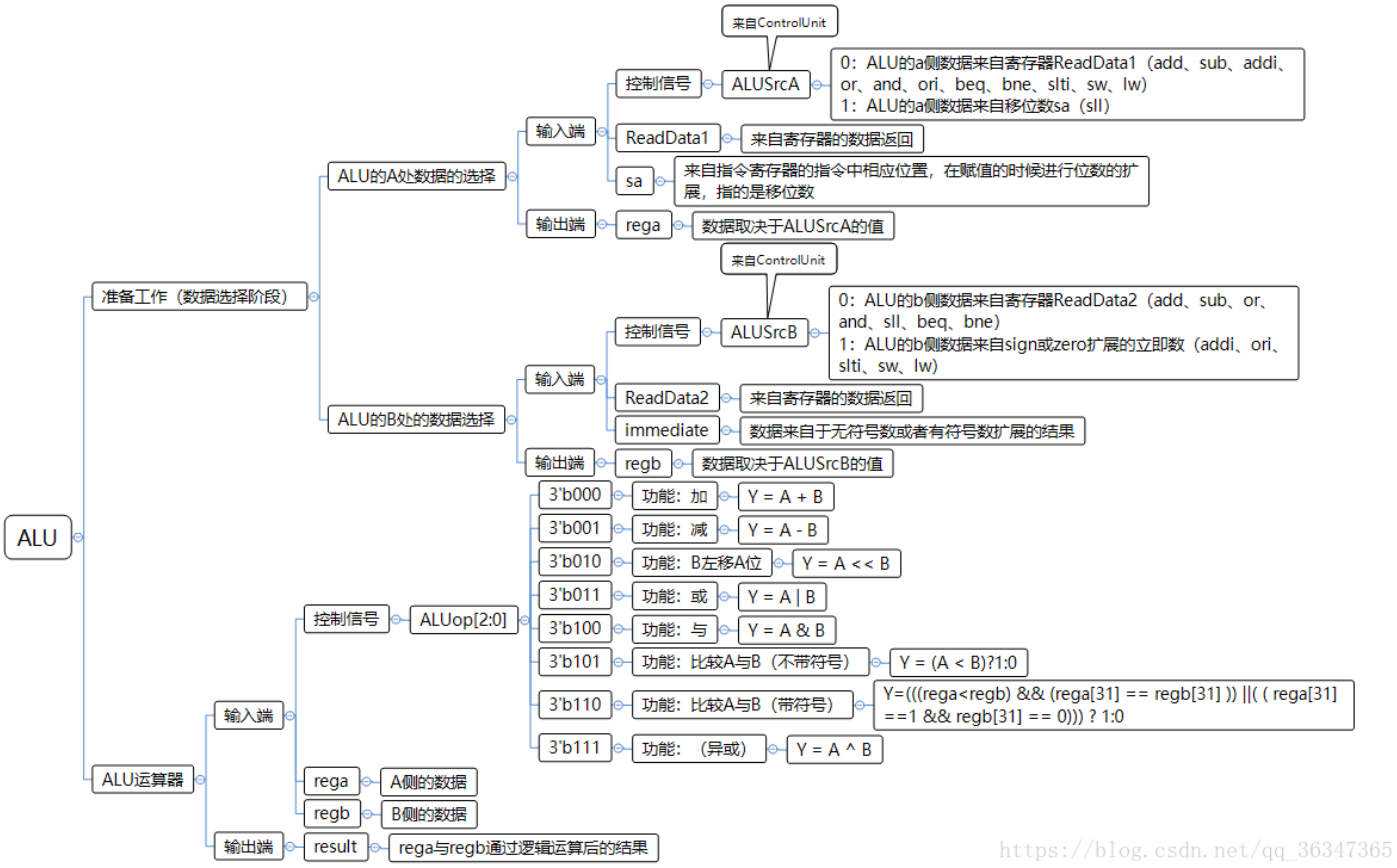
实现的代码也不会很长：

```
always @(immediate or ExtSel) begin
    //当ExtSel为0时，进行无符号扩展
    //当immediate为0时，高位补0
    if (ExtSel == 0 || immediate[15] == 0)
    result = {16'b0000000000000000, immediate};
    else
    result = {16'b1111111111111111, immediate};
    //当ExtSel为1且immediate为1时，高位补1
end
```


接着就进入了ALU的实现了，首先我们先看一下ALU在单周期CPU的数据通路和控制路线图中的详细图片：



我们可以看到在ALU逻辑运算器的前面还有两个数据选择器，主要是进行ALU两侧载入的数据的选择，因为有偏移量以及立即数，A侧是因为偏移量，而B侧是因为立即数，因为当时没有考虑详细，所以我把这一个模块分为了两个数据选择器以及ALU逻辑运算器这三个小模块来实现。从而可以得到一下的结构：



功能表如下：

ALUOp[2..0]	功能	描述	相关的指令
000	Y = A + B	加	addi、add、lw、sw、
001	Y = A – B	减	sub、beq、bne、
010	Y = B<<A	B左移A位	sll、sllv 、lui
011	Y = A ∨ B	或	or、ori

100	Y = A ∧ B	与	and、andi
101	Y=（A<B）?1: 0	比较A与B 不带符号	sltu、
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较A与B 带符号	slti、
111	Y = A Å B	异或	xor、

其中两个数据选择器的架构比较类似，主要都是由控制信号ALUSrcA以及ALUSrcB来实现数据选择，只不过A端口需要进行sa的无符号扩展罢了：

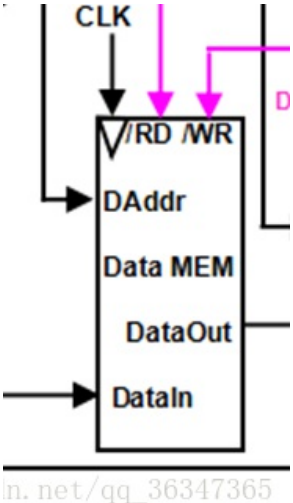
这两个数据选择器的核心代码如下：

```
always @(ReadData1 or ALUSrcA or sa)begin
    if (ALUSrcA == 0)    //ALUSrcA为0时，ALU的a侧数据来自寄存器
        assign rega = ReadData1;
    else
    //ALUSrcA为1时，ALU的a侧数据来自指令中的偏移量sa
        assign rega = { 27'b000000000000000000000000000, sa};
end

always @(ReadData2 or ALUSrcB or immediate)begin
    if (ALUSrcB == 0)
    //当ALUSrcB为0的时候ALU的b侧数据来自寄存器
        assign regb = ReadData2;
    else//当ALUSrcB为1的时候ALU的b侧数据来自立即数
        assign regb = immediate;
end
```

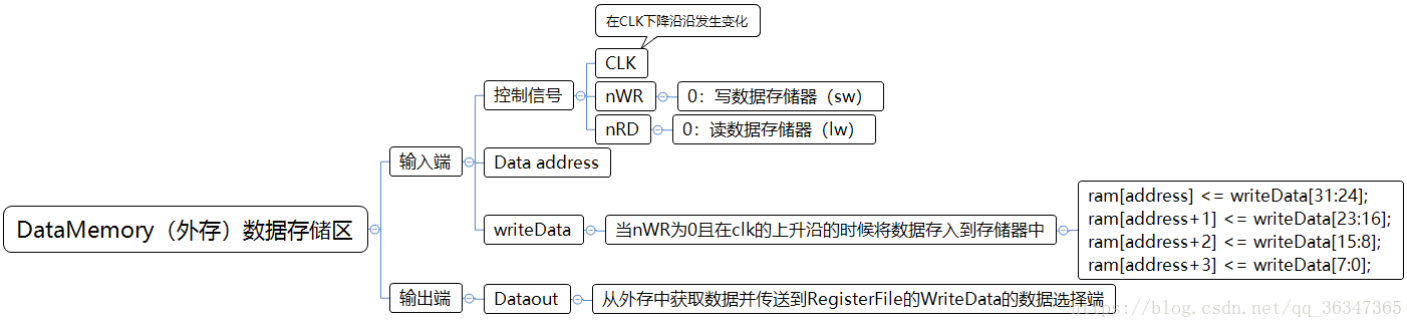

然后就是ALU的代码了，ALU主要是要依据ALUop指令的类型来执行对应的操作，同时还需要返回一个Zero参数，只要是用于判断PC的状态，因为beq以及bne在CPU中实际是进行减法操作。核心代码如下：

```
assign zero = (result==0)?1:0;
always @( ALUopcode or rega or regb ) begin
    case (ALUopcode)
        3'b000 : result = rega + regb;          //加法
        3'b001 : result = rega - regb;          //减法
        3'b010 : result = regb << rega;         //regb左移rgba
        3'b011 : result = rega | regb;          //或
        3'b100 : result = rega & regb;          //且
        3'b101 : result = (rega < regb)?1:0;    // 无符号大小比较
        3'b110 : begin                          // 有符号大小比较
                    if(rega < regb && (rega[31] == regb[31]))result
                     = 1;
                    else if (rega[31] == 1 && regb[31] == 0) result
                     = 1;
                    else result = 0;
                end
        3'b111 : result = rega ^ regb;          //异或
    endcase
end
```


6.实现完RegisterFile以及ALU两个部分，也就意味着EXE环节的完成了，接下来就是MEM部分的实现了，也就是DataMemory（外存）数据存储区，继续前面的操作，先截一个小图：



可以看到DataMemory的存储以及读取数据操作是由两个控制信号来控制的，同时这些操作是在CLK的下降沿触发的。而因为这也是一个存储区域，所以内部同样需要reg数组作为内存来存储内容。所以我们可以分析得到DataMemory的结构如下：

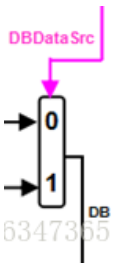


其中存储器还是8位的存储器。代码如下：

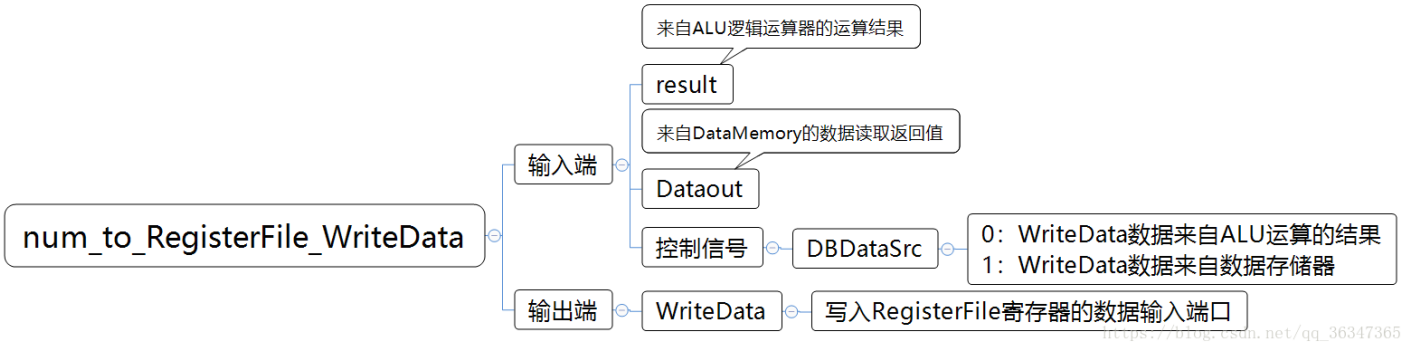
```
    input nRD, // 为0，正常读；为1,输出高组态
    input nWR, // 为0，写；为1，无操作
```

```
reg [7:0] ram [0:60]; //存储器
// 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
    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
```


7.实现完MEM模块，接下来就是WB步骤了，这里简化为了一个数据选择器Num to WriteData，先截个小图：



其中选择的数据分别是来自ALU逻辑运算器的运算结果以及数据存储器的读取数据。而DBDataSrc作为控制信号进行区分，结果是用于写回RegisterFile的WriteData的数据输入端口的。结构如下：



代码部分跟之前的数据选择器都是很类似的，只不过是触发条件换了而已，所以这里就不做代码展示了。

8.到此为止我们已经实现了CPU工作的5个基本步骤了，所以接下来就是应该来写顶层模块了，顶层模块在前面就有提到过，也就是：

```
1  module s_cpu(
2      //时钟信号
3      input clk,
4      //控制信号
5      input Reset
6      );
7      //以下为各个模块
8  endmodule
```

而这里需要做的是将各个模块填充进去，比如下面这样子：

```
    PC pc(
    .CLK(clk),
    .Reset(Reset),
    .PCWre(PCWre),
    .immediate(immediate_num),
    .address(IDataOut),
    .PCSrc(PCSrc),
    .PC(PC_out)
    );
```


将以上7个模块添加到这里面就好了。（因为这一段很长而且没有任何技术含量，所以就没有展示完全）。

另外，其中有一些变量是有关联的，比如immediate_num是从扩展那一个模块来的，那么就应该用同一个变量名。这样就是表示这两个模块用的是同一个数据，而不会因为不同的数据而导致出现截断的问题或者程序运行出错。

到此为止，我们算是完成了一个S_CPU单周期CPU），但是驱动CPU运行的时钟写在哪里呢？这个就是仿真的事件了，所以仿真的模块就见第五部分的说明吧。

实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

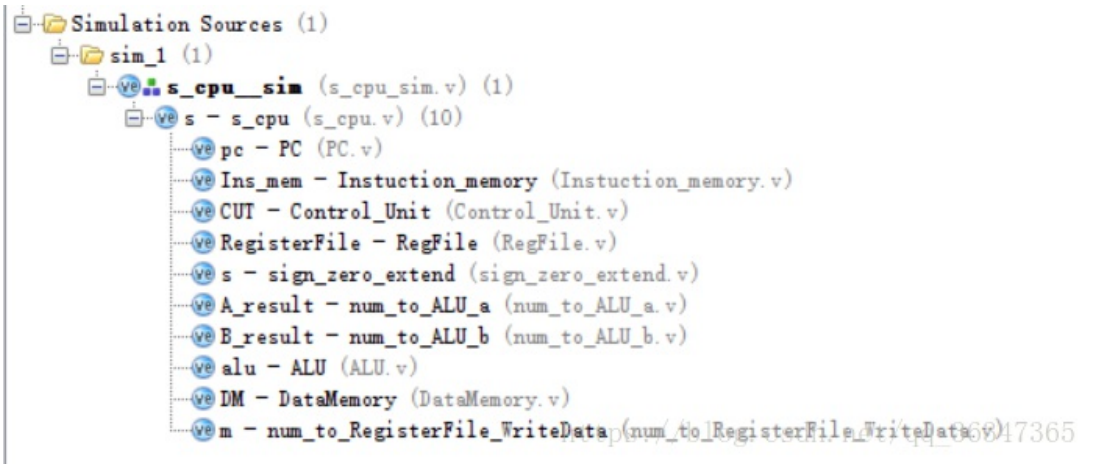
实验过程与结果

仿真模块：

根据上面的顶层模块S_CPU，我们可以看到需要两个输入控制，分别是CLK以及Reset，所以仿真模块一开始需要将Reset置为0以使PC处于0的状态以便程序的执行。所以仿真模块如下：

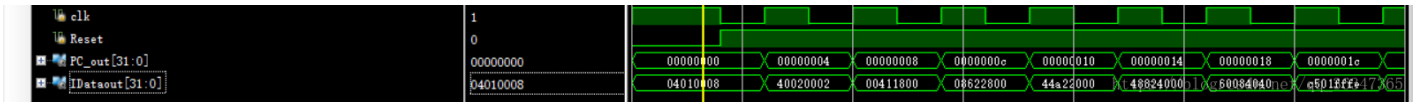
```
module s_cpu_sim( );
    //时钟信号
    reg clk;
    //控制信号
    reg Reset;

    s_cpu s(
    //时钟信号
    .clk(clk),
    //控制信号
    .Reset(Reset),
    );
    initial begin
        clk = 1;
        assign Reset = 0;
        assign InsMemRW = 1;
        // Initialize Inputs

        #40
        clk = 0;
        assign Reset = 1;
        // Wait 40 ns for global reset to finish
        forever #20 clk = !clk;
       #600 $stop;
    end
endmodule
```


建完这一个文件我们可以看到Vivado中的文件结构如下：



其中s_cpu_sim是仿真文件，而s_cpu是顶层模块，后面的就是7个模块（10个小模块）。

点击仿真之后的效果如下：



其中clk是时钟信号，而Reset是控制PC变化的信号，同时IDataout指的是指令，这些指令是来自于老师给的测试文件，下文便会对这个测试进行测验。根据上面的情况我们成功的对CPU进行了仿真。

2.仿真测验：

这一部分就是根据老师所提供的测试指令，也就是程序运行的时候需要装载的指令文件区域，如图（一开始想用InsMemRW对指令寄存器的状态进行控制，后来发现似乎启动的时候会比较坑，所以就放弃了，改成了initial的时候就装载指令了）

```
initial begin
$readmemb("E:/vivado/S_CPU/test.txt", Instruction_memory);//读取文件
end
```

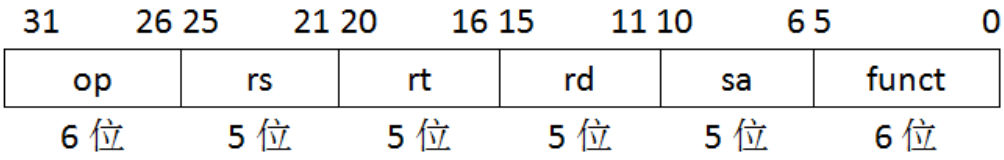

然后就是老师给的表格填写了（需要结合前面的指令op以及将$n寄存器（n代表任意数字）用n作为下标来代替，为了简化），表格如下：

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	=	00411800
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000	=	08622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	=	44a22000

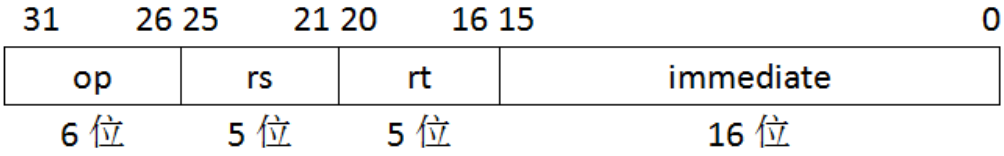
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000	=	48824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001C	bne $8,$1,-2 (≠,转18)	110001	01000	00001	1111 1111 1111 1110	=	C501fffe
0x00000020	slti $6,$2,8	011011	00010	00110	0000 0000 0000 1000	=	6c460008
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	=	04e70008
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	00000	00000	0000 0000 0001 0000	=	E0000010
0x0000003C	addi $10,$0,10	000001	01010	00000	0000 0000 0000 1010	=	0540000a
以下为添加的指令							
0x00000040	xor $10,$1, $2	010011	00001	00010	0101 0000 0000 0000	=	4c225000
0x00000044	sllv $10,$6, $9	011001	01001	00110	0101 0000 0000 0000	=	65265000
0x00000048	sub $10,$9, $5	000010	01001	00101	0101 0000 0000 0000	=	09255000
0x0000004C	sltu $2,$10, $7	011100	01010	00111	0001 0000 0000 0000	=	71471000
0x00000050	lui $6, 10	100001	10000	00110	0000 0000 0000 1010	=	8606000a
0x00000054	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000

此处补充一下，前面忘记写了：就是MIPS的三种指令类型，有I（立即数指令），R（普通指令），J（跳转指令）

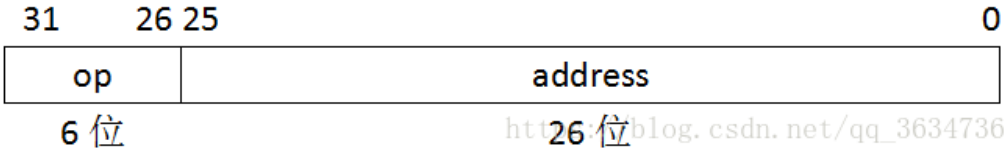
R 类型：



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；

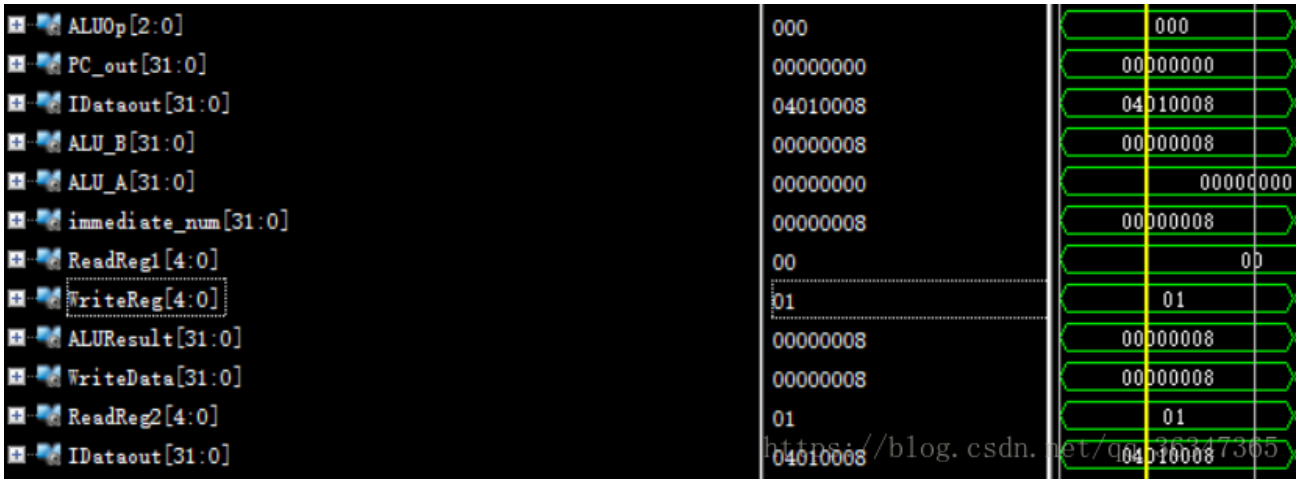
immediate：为16位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Load）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

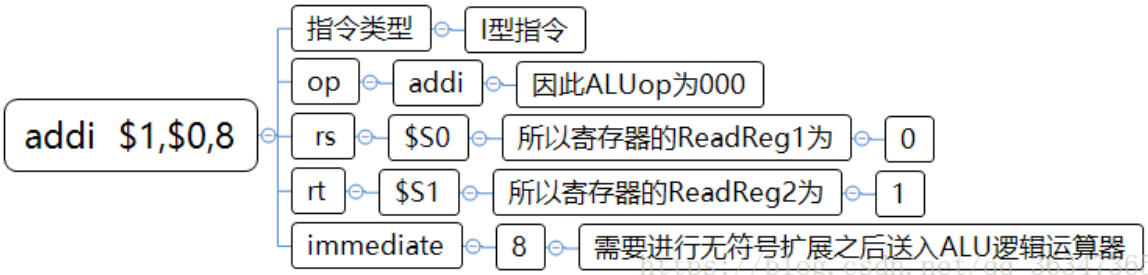
address：为地址。

好了，接下来就是测试每一条指令了：

a) addi $1,$0,8

仿真结果如下：



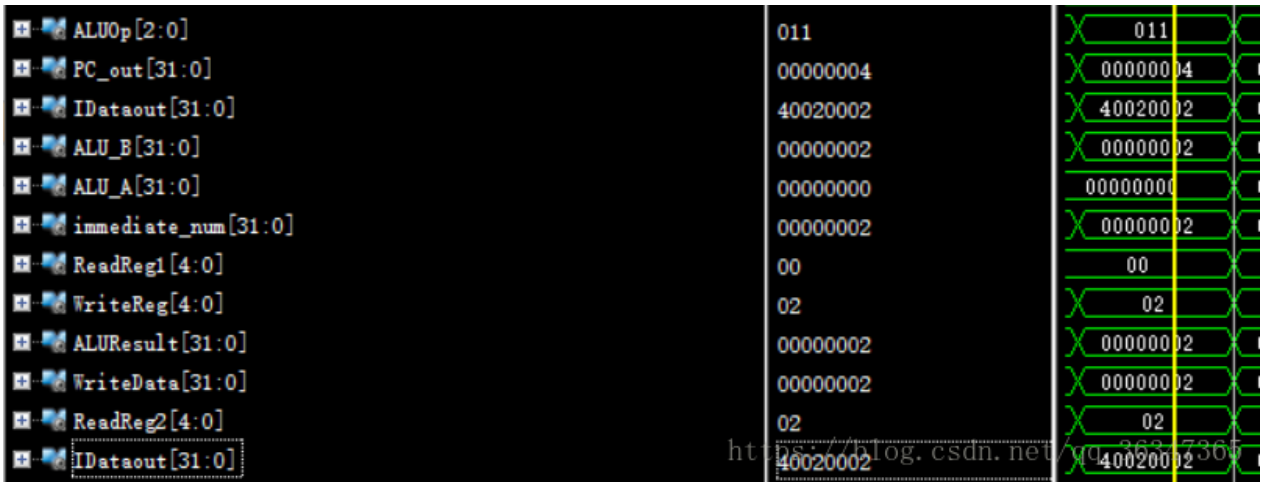


所以根据这一个我们可以知道写入的寄存器为1号寄存器，上面的仿真显示的WriteReg是正确的，同时ALU两侧的数据选择后分别为0与8，因为开机时的内存寄存器的数值都是0，同时是0号寄存器，所以返回的数值是0，也就是ALU_a是0，而另一侧的数据则是来自于立即数的扩展，所以为8，然后再经过WB组件，因为不包含MEM的操作，所以WriteData也就是ALU的运算结果：8.这一个语句是正确的。此时，寄存器数值为：

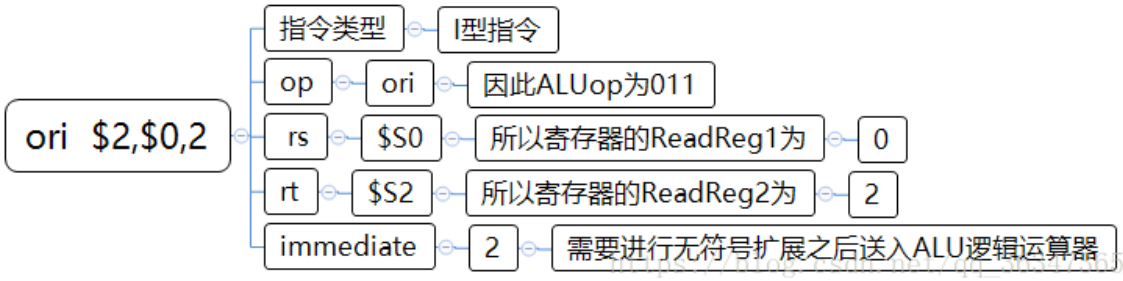
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8									

b) ori $2,$0,2

仿真结果如下：



指令分析如下：

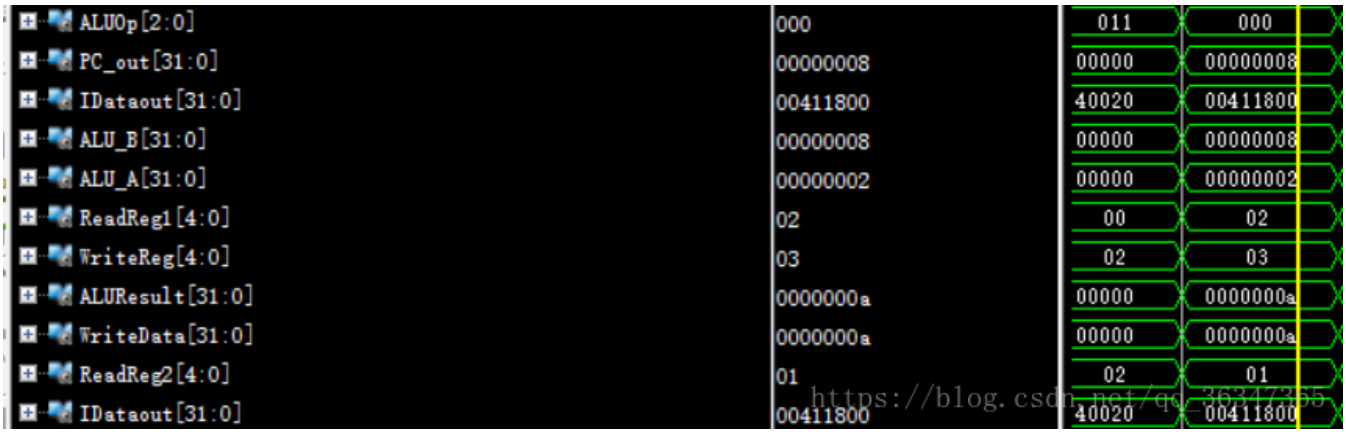


所以根据这一个我们可以知道写入的寄存器为2号寄存器，上面的仿真显示的WriteReg是正确的，同时ALU两侧的数据选择后分别为0与2，然后再经过WB组件，因为不包含MEM的操作，所以WriteData也就是ALU的运算结果：2。这一个语句是正确的。此时，寄存器数值为：

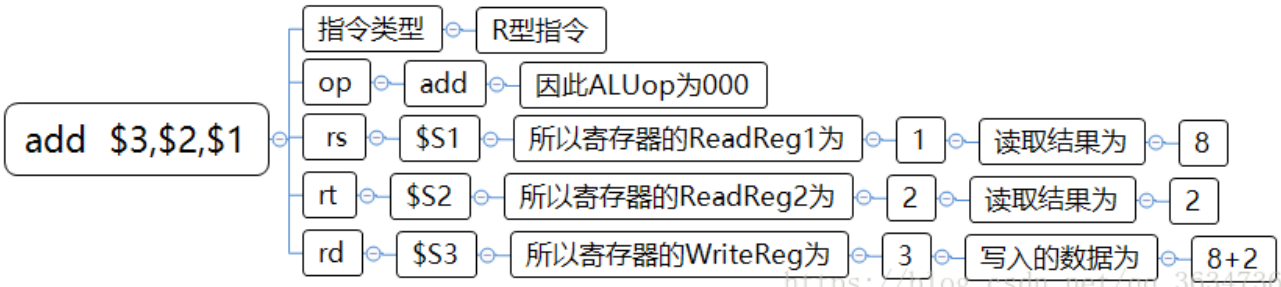
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2								

c) add $3,$2,$1

仿真结果如下：



指令的分析如下：



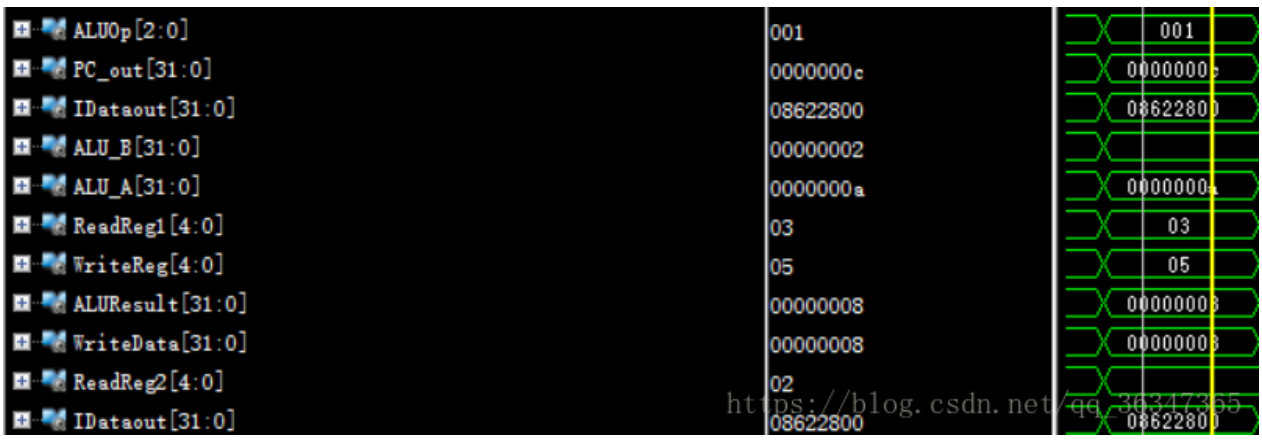
这是一条R型指令，所以rd为寄存器写入端的下标，也就是WriteReg = $S3 = 3， 所以上面的图是正确的，而WriteData是由ALU逻辑运算器传过来的，所以值为8+2 = 10，表示为16进制也就是a，这也是正确的。

到此为止，寄存器堆数值为：

寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10							

d) sub $5,$3,$2

仿真结果如下：



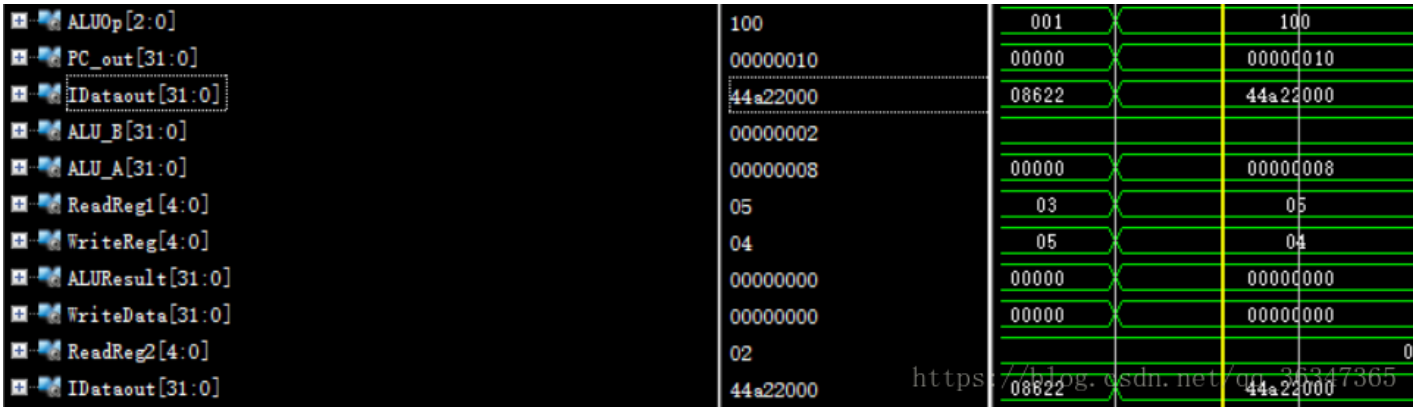
指令分析如下：

这是一条R型指令，所以就没有跳转也没有立即数的扩展。所以rd为寄存器写入端的下标，也就是WriteReg = $S5 = 5， 所以上面的图是正确的，而WriteData是由ALU逻辑运算器传过来的，所以值为10-2= 8，这也是正确的。此时的寄存器堆的数据为：

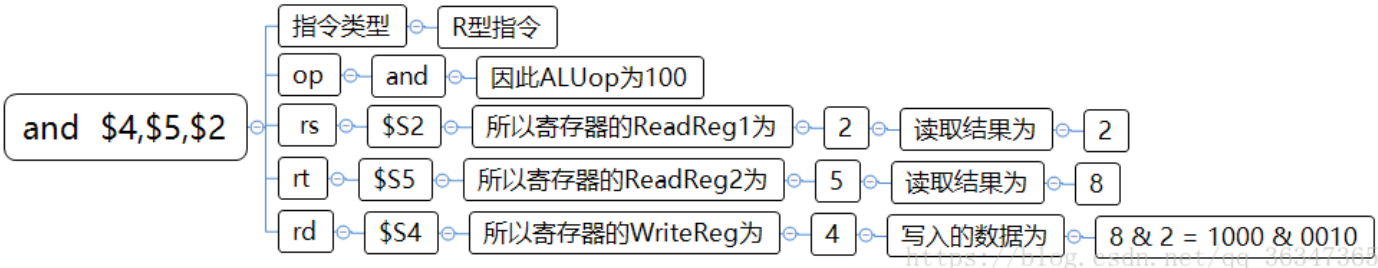
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10		8					

e) and $4,$5,$2

仿真结果如下：



指令分析如下：

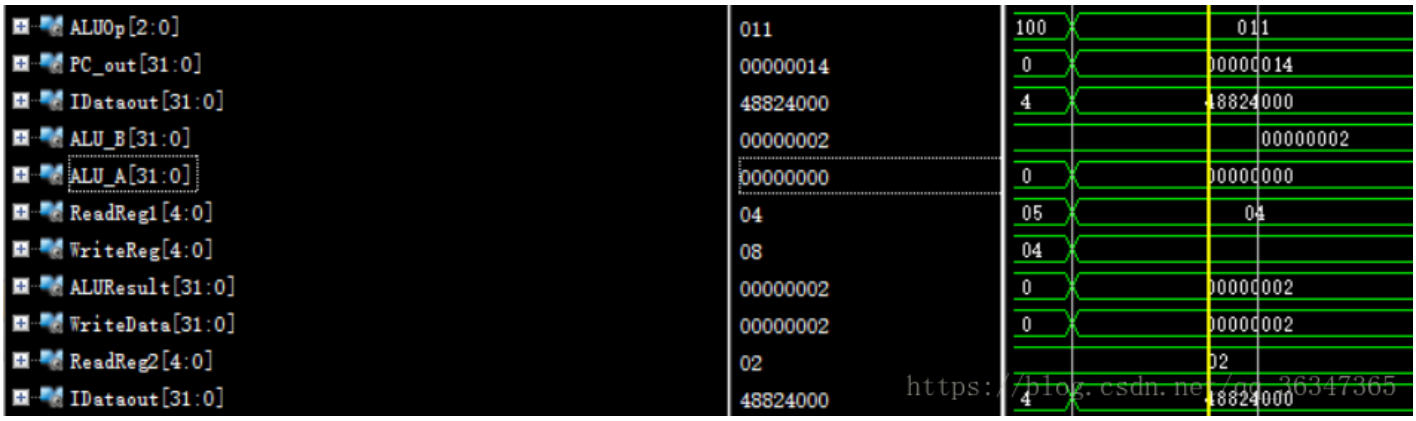


这仍然是一条R型指令，所以ALU两侧的数据来自于寄存器的读取数据，也就是ReadData1：2（2号寄存器的数值）， 以及ReadData2：8（5号寄存器的数值），然后写入寄存器的地址为WriteReg为4，ALU运算的结果是8&2 = 1000 & 0010 = 0000，所以写入的数据为0，这是正确的，此时寄存器堆的数据为：

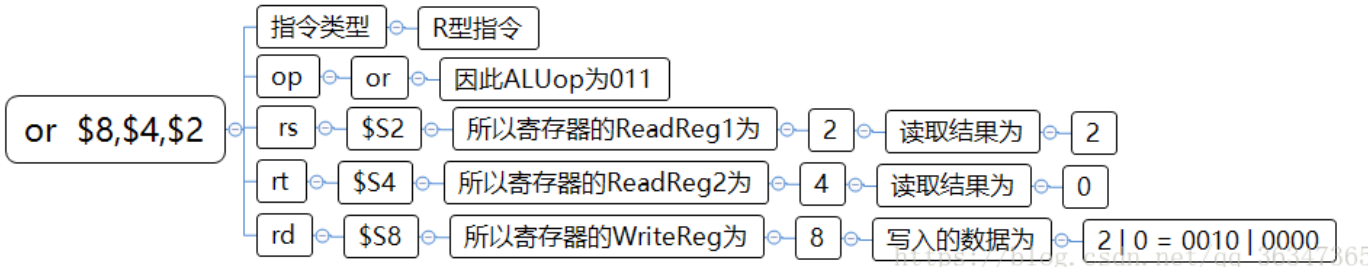
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8					

f) or $8,$4,$2

仿真结果如下：



指令分析如下：

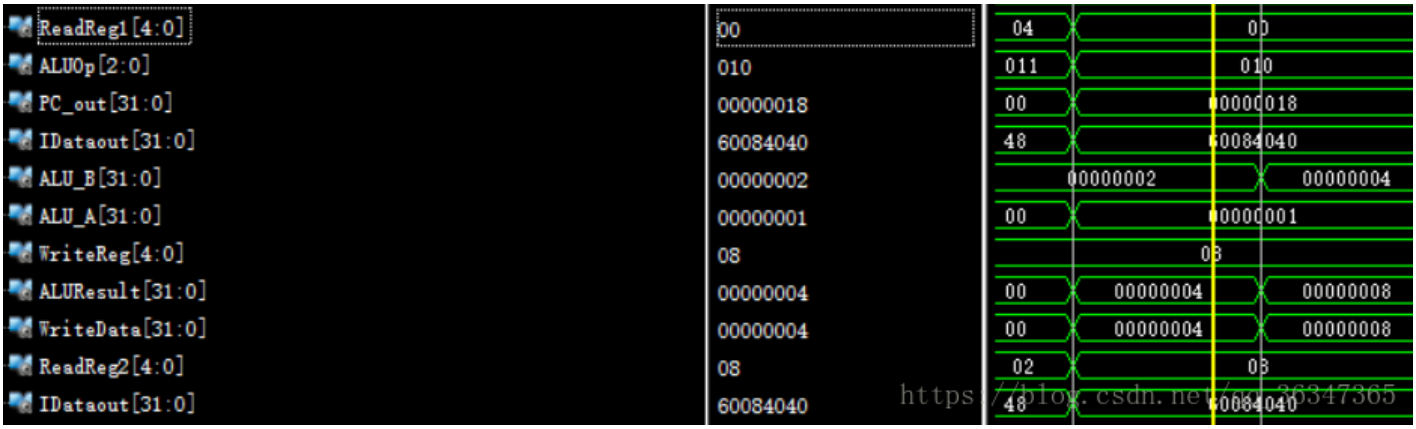


这是一条R型指令，所以ReadReg1也就是2，ReadReg2也就是4，WriteReg是8，然后因为是or运算，所以ALUop为011，然后ReadData1 = $S2 = 2, ReadData2 = $S4 = 0,所以WriteData也就是经过ALU运算后的结果是2|0 = 0010 | 0000 = 0010 = 2。所以上面的仿真是正确的。此时寄存器堆的数据为：

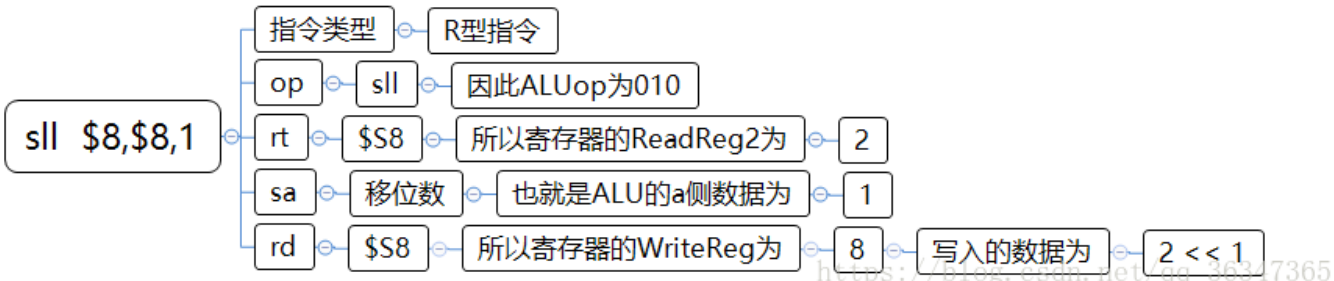
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8			2		

g) sll $8,$8,1

仿真结果如下：



指令分析如下：

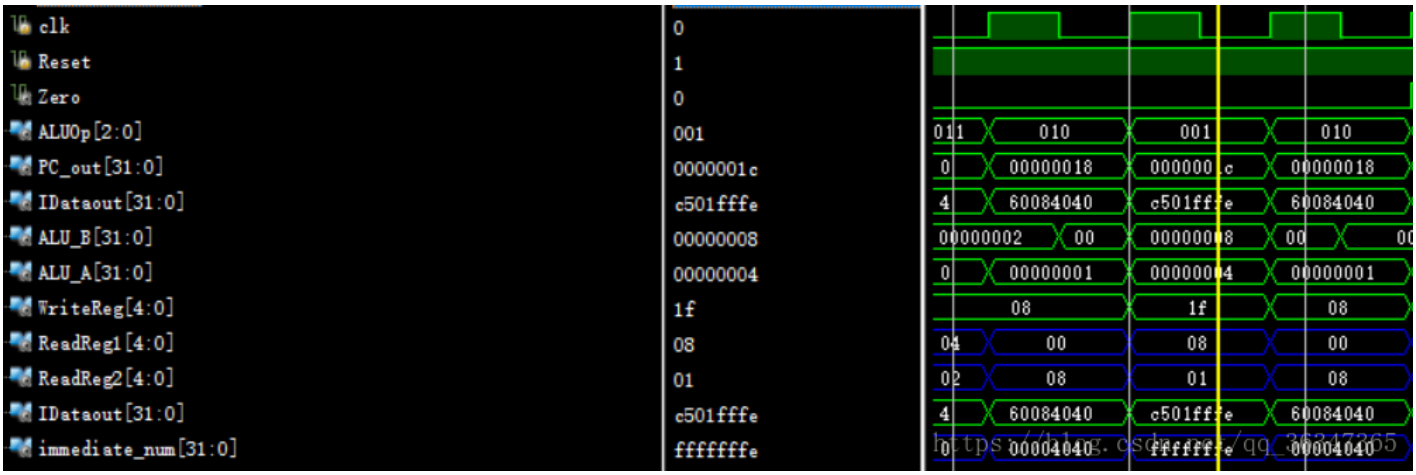


这虽然似乎包含一个数字，但是其实还是一条R型指令，只不过这一次因为是sll即是移位，所以这次ALU的a侧数据不再是rs，而是sa，也就是指令的6-10位，在此处为1，可以通过ALU_A看到数值为1，然后就在ALU中做移位运算后结果为2 << 1 = 100 = 4。所以WriteData是4，这是正确的。此时寄存器堆中的数据为：

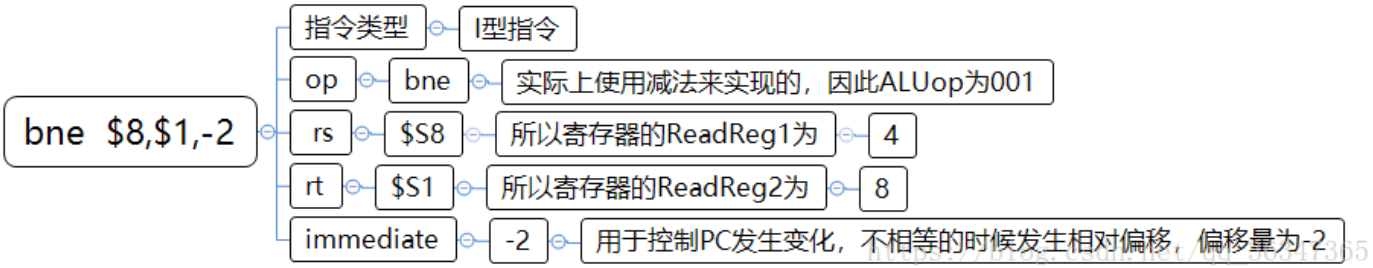
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8			4		

h) bne $8,$1,-2 (≠,转18)

仿真结果如下：



指令分析如下：

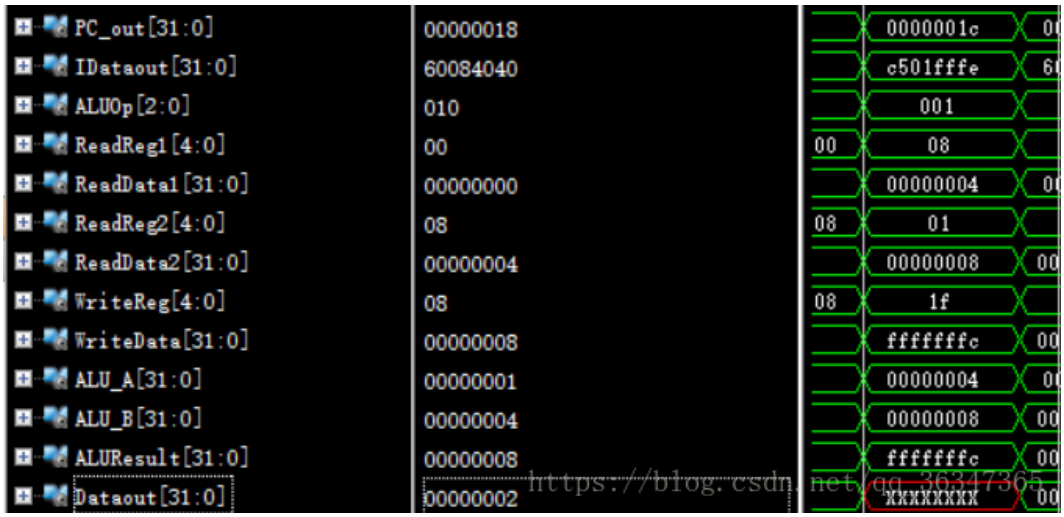


bne是I型指令，用于比较两个数字是否相同，不同则发生跳转指令（相对跳转），里面包含有立即数，为-2，但是计算机里面是用补码来表示负数的，所以为0xfffffffe，上面的显示是正确的。同时rs是8，也就是ReadReg1为8，rt为1，即ReadReg2为1，根据上面的寄存器堆的表格可以得出ReadData1为4，ReadData2为8，因为两个数值不等，所以在ALU逻辑运算运算后得到的Zero值为0，而根据op为bne的操作符同时Zero为0，所以ControlUnit就触发PCWre转变为2'b01而导致PC的变化进行相对寻址变化，也就是PC = PC + （-2）×4 + 4，也就是回到了上一条指令，所以可以看到目前的PC值为0x0000001c，而下一条指令的PC变为了0x00000018，也就是减少了4。所以上面的都是正确的。

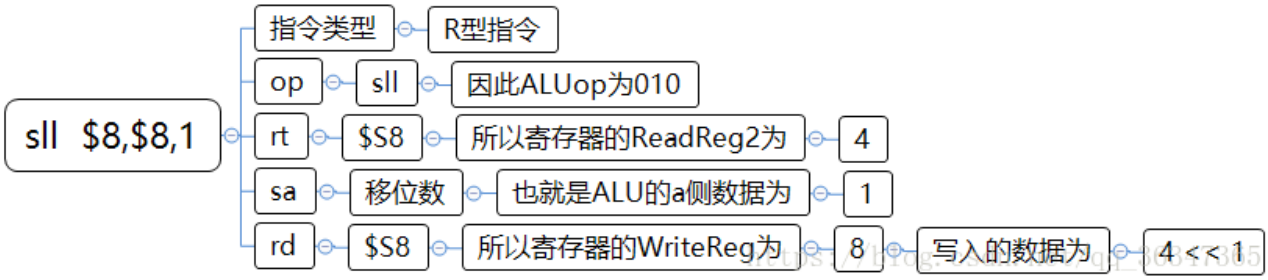
此时的寄存器堆的数值未发生变化。

i) sll $8,$8,1（因为在上面的bne指令判断之后又重新跳转了回来）

仿真结果如下：



指令分析如下结果如下：

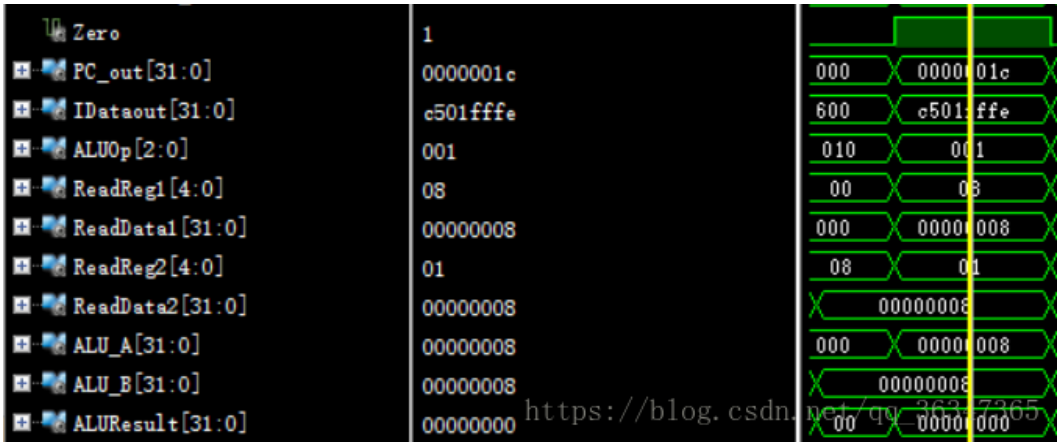


跟前面一样的，移位数为1，然后就在ALU中做移位运算后结果为4 << 1 = 1000 = 8。所以WriteData是8，仿真结果正确的。此时寄存器堆中的数据为：

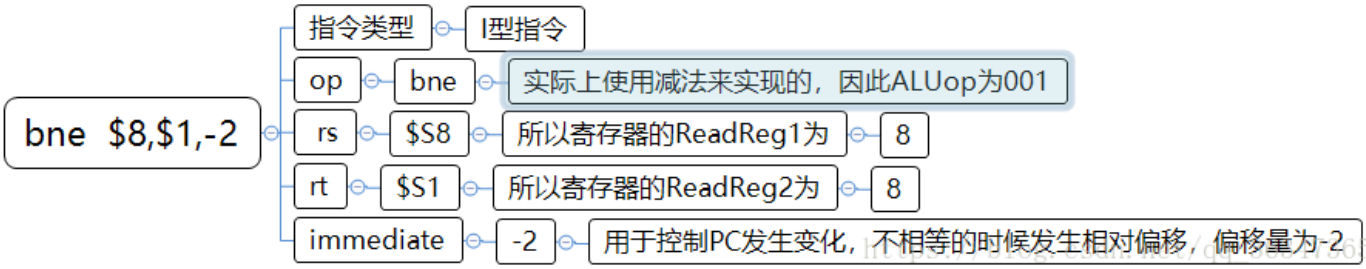
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8			8		

j) bne $8,$1,-2 (≠,转18)

仿真结果如下：



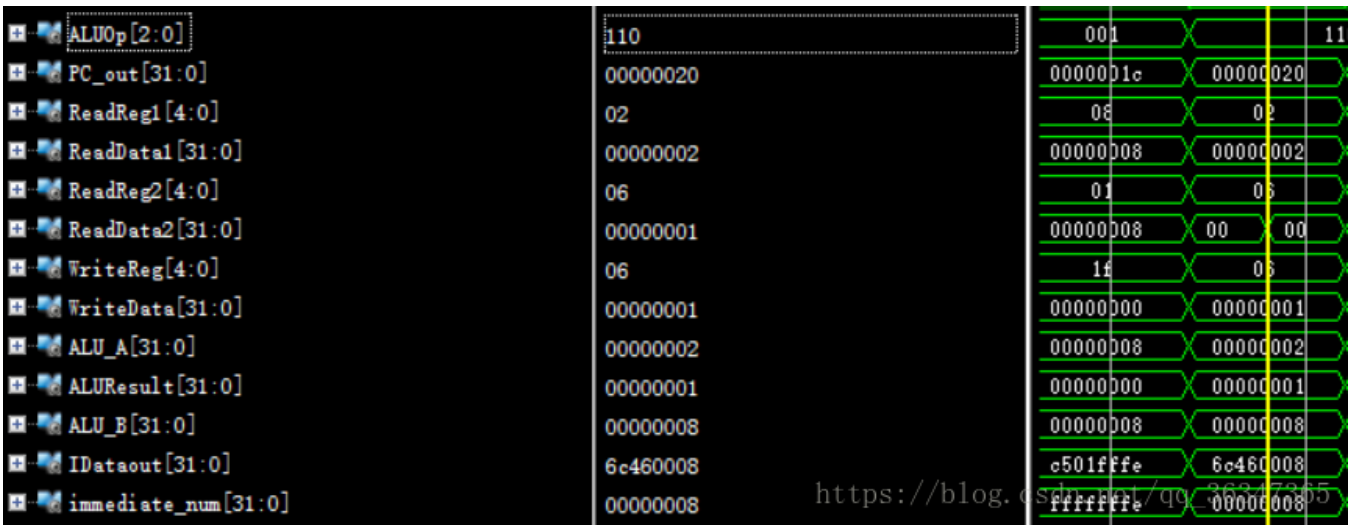
指令分析如下：



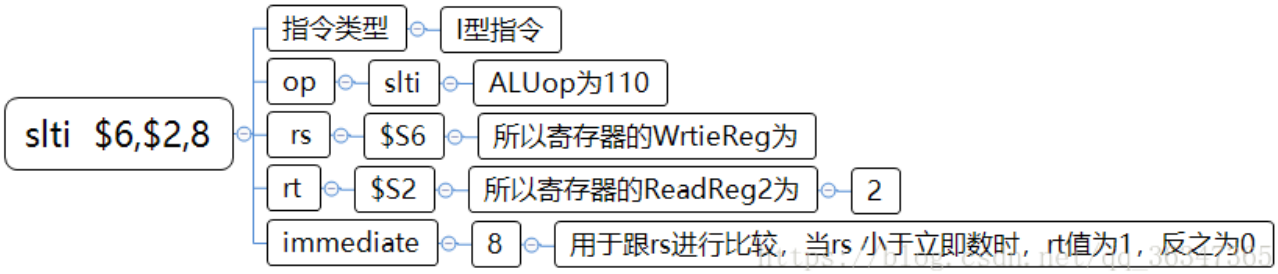
经过了上面的再一次左移之后$S8以及$S1的数值都变为了8，上面的ALU_A以及ALU_B都为8，这是正确的。而在做了减法之后，ALU逻辑运算其的结果为0，这也就导致了Zero转变为1，从而导致ControlUnit进行综合判断之后决定PCWre为00从而不再跳转而是使PC发生普通变化。上面显示的结果也是正确的。这个操作并未对寄存器进行修改操作，所以就不列表格了。

k) slti $6,$2,8

仿真结果如下：



指令分析如下：



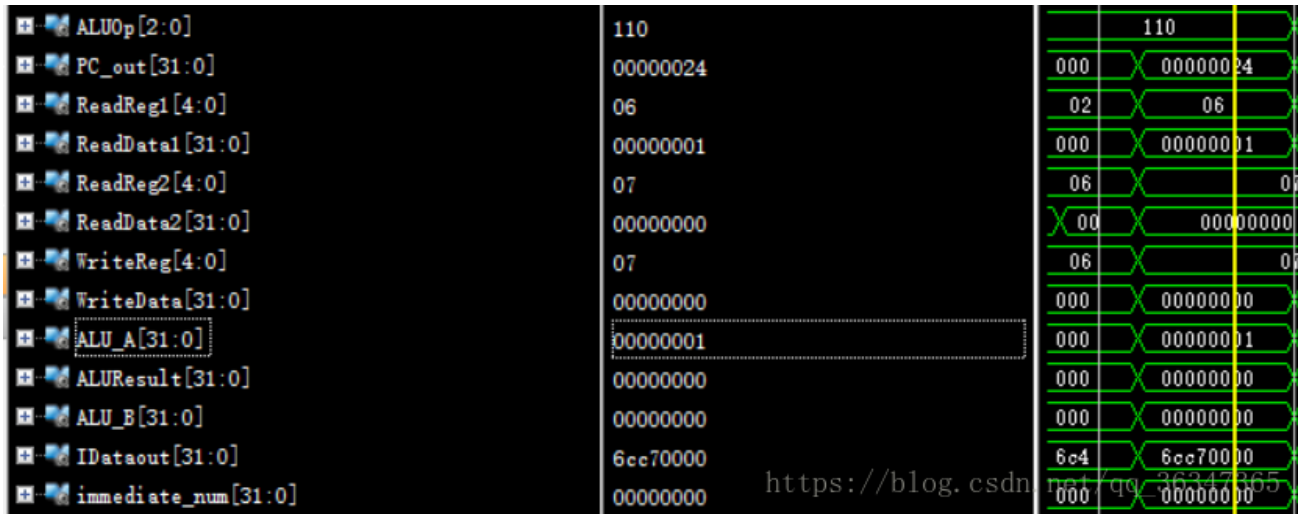
slti是用于带符号比较，而且是为I型指令，因为是带符号比较，所以ALUop为110，而因为是立即数，所以immediate_num为8，而读取2号寄存器中的数据为2，2<8，所以6号寄存器的值被赋予了1，也就是ALUResult以及WriteData为1。上面显示的结果也是正确的。

此时寄存器堆的数据为：

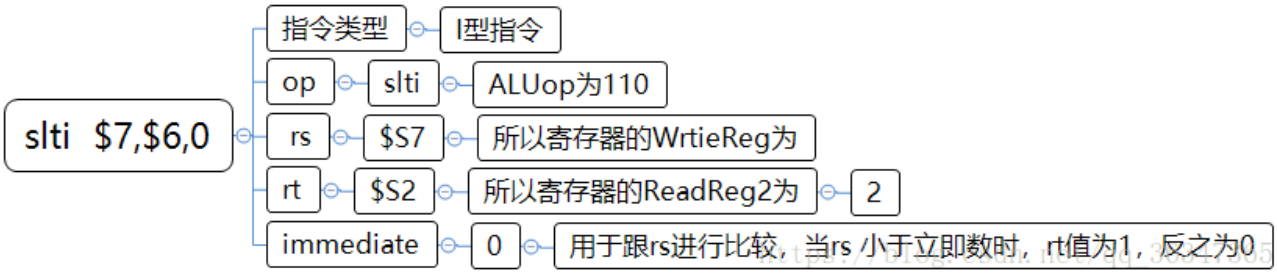
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1		8		

l) slti $7,$6,0

仿真结果如下：



指令分析如下：



分析如下

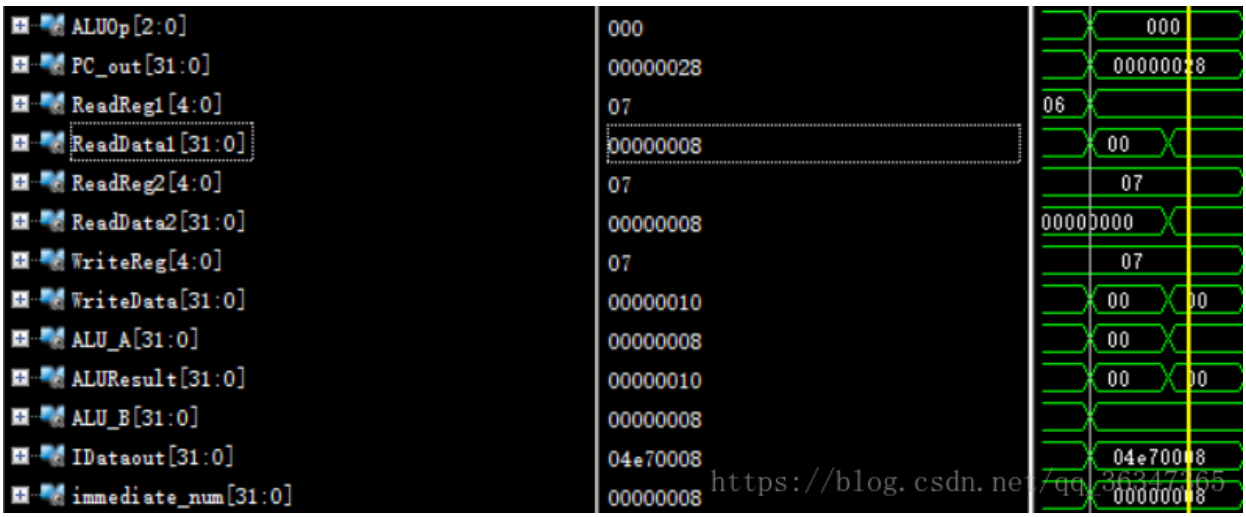
slti是用于带符号比较，而且是为I型指令，因为是带符号比较，所以ALUop为110，而因为是立即数，所以immediate_num为0，而读取2号寄存器中的数据为2，2>0，所以7号寄存器的值被赋予了0，也就是ALUResult以及WriteData为0。上面显示的结果也是正确的。

此时寄存器堆的数据为

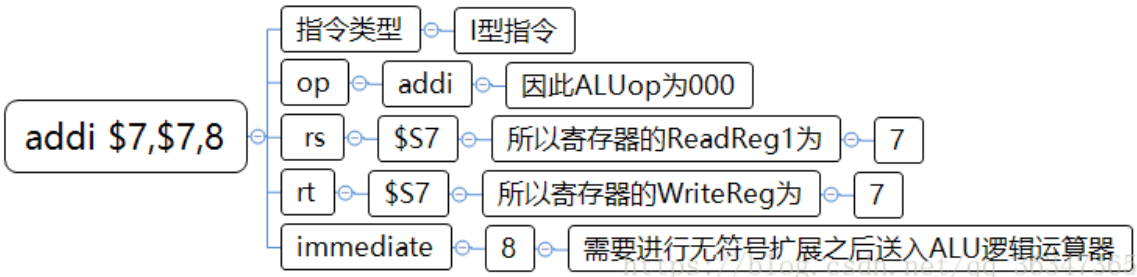
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1	0	8		

m) addi $7,$7,8

仿真结果如下：



指令分析如下：

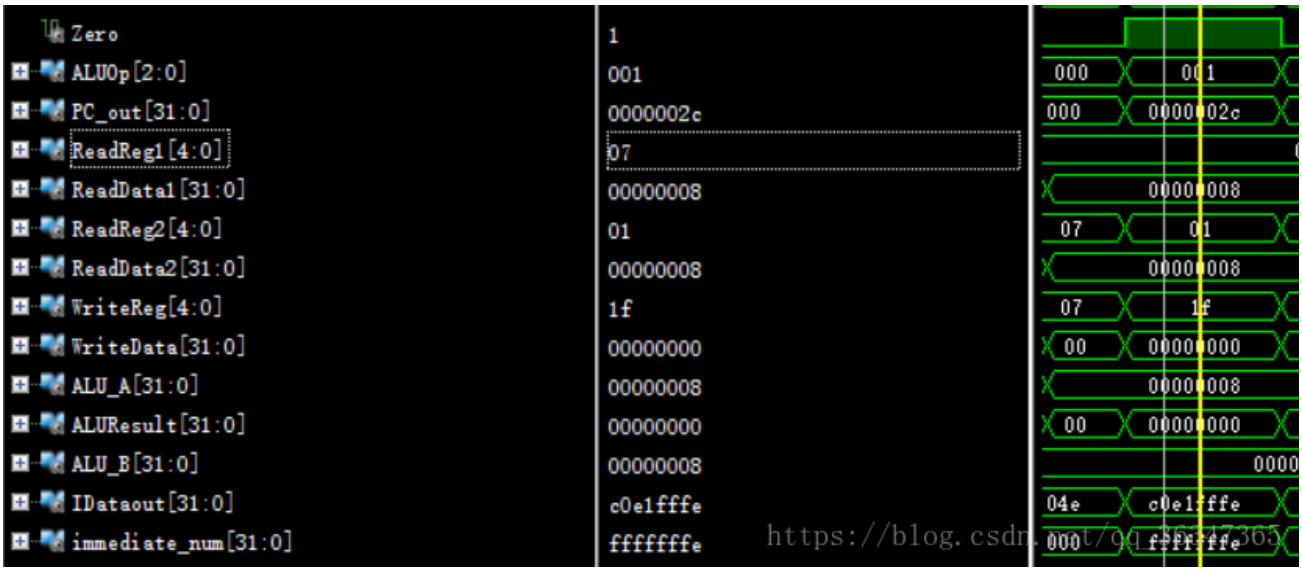


我们可以知道写入的寄存器为7号寄存器，上面的仿真显示的WriteReg是正确的，同时ALU两侧的数据选择后分别为0与8，因为7号寄存器的数值是0，也就是ALU_a是0，而另一侧的数据则是来自于立即数的扩展，所以为8，然后再经过WB组件，因为不包含MEM的操作，所以WriteData也就是ALU的运算结果：8.这一个语句是正确的。此时，寄存器堆数值为

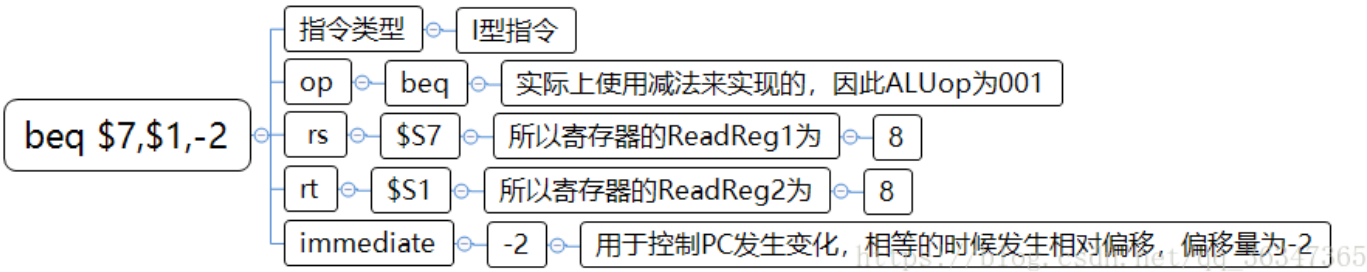
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1	8	8		

n) beq $7,$1,-2 (=,转28)

仿真结果如下：



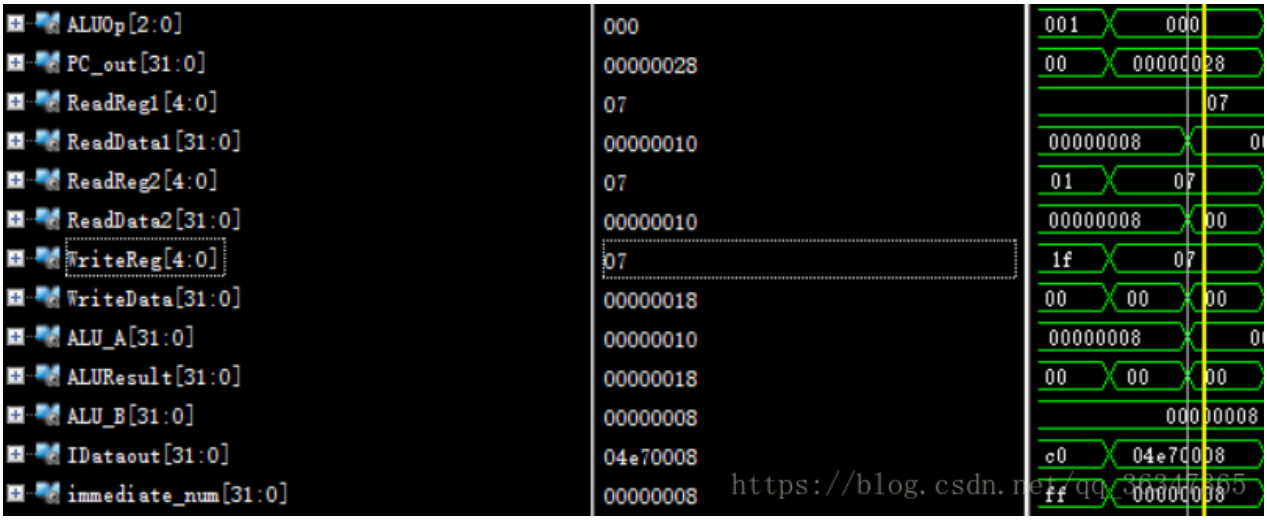
指令分析如下：



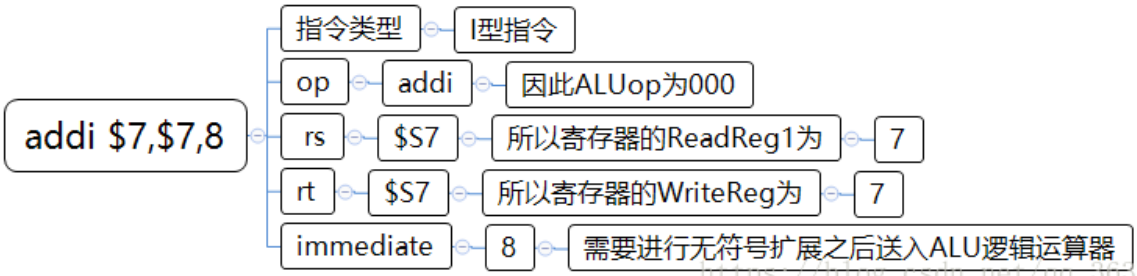
beq是I型指令，用于比较两个数字是否相同，然而刚好与bne相反，beq是相同则发生跳转指令（相对跳转），里面包含有立即数，为-2，但是计算机里面是用补码来表示负数的，所以为0xfffffffe，上面的显示是正确的。同时rs是7，也就是ReadReg1为7，rt为1，即ReadReg2为1，根据上面的寄存器堆的表格可以得出ReadData1为8，ReadData2为8，因为两个数值相等，所以在ALU逻辑运算运算后得到的Zero值为1，而根据op为beq的操作符同时Zero为1，所以ControlUnit就触发PCWre转变为2'b01而导致PC的变化进行相对寻址变化，也就是PC = PC + （-2）×4 + 4，也就是回到了上一条指令，所以可以看到目前的PC值为0x0000002c，而下一条指令的PC变为了0x00000028，也就是减少了4。所以上面的都是正确的。因为没有对寄存器堆的数值进行改变，所以此处就不做展示了。

o) addi $7,$7,8（因为前面的指令是beq指令，同时两个数值一样，所以就发生了相对寻址的变化）

仿真结果如下：



指令分析如下：

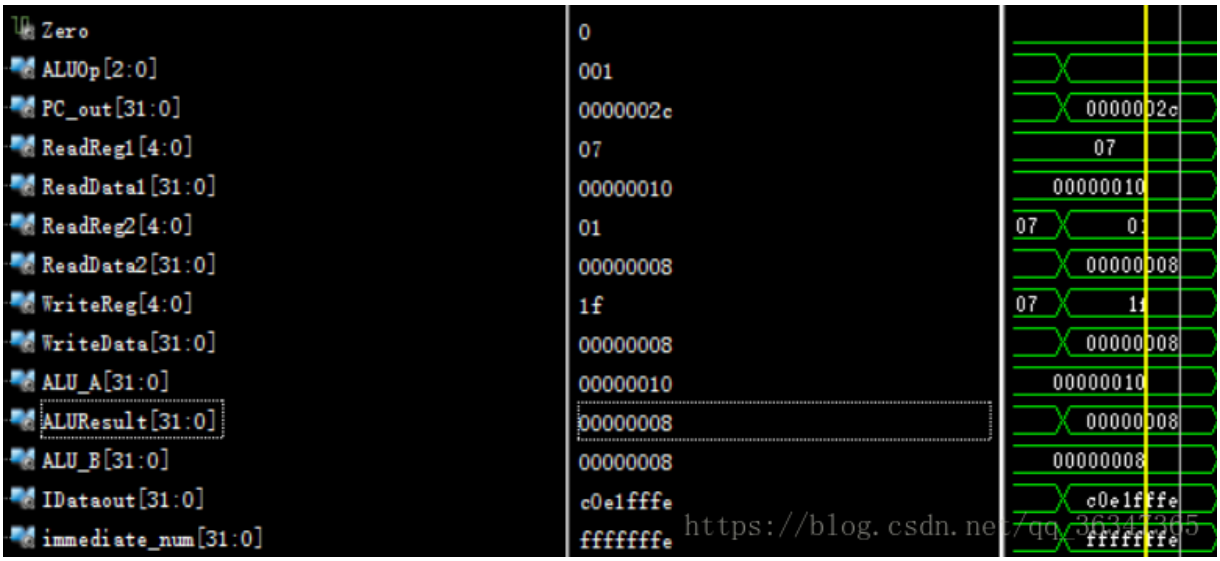


因为是跟前面的语句一样的，所以很多的东西都是一样的，不同的只是ReadData1的数值变为了8，因为此时的7号寄存器的值为8，所以经过ALU逻辑运算器之后的ALU_result（也是WriteData）为8+8 = 16，上面的仿真是正确的。此时寄存器组的数据为：

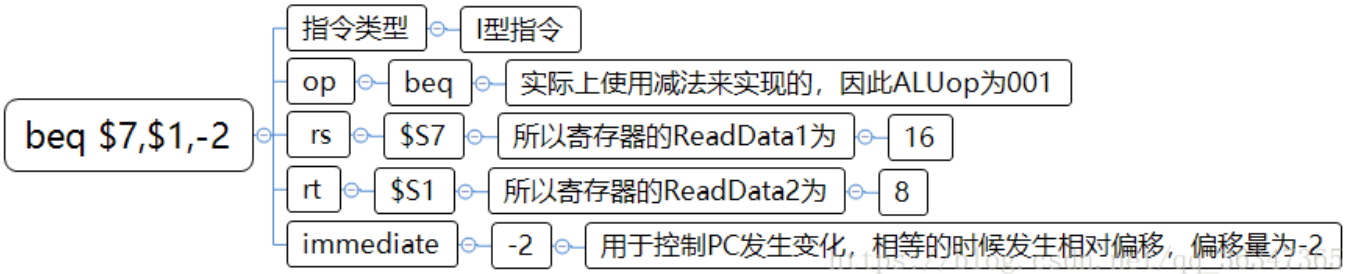
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1	16	8		

p) beq $7,$1,-2 (=,转28)

仿真结果如下：



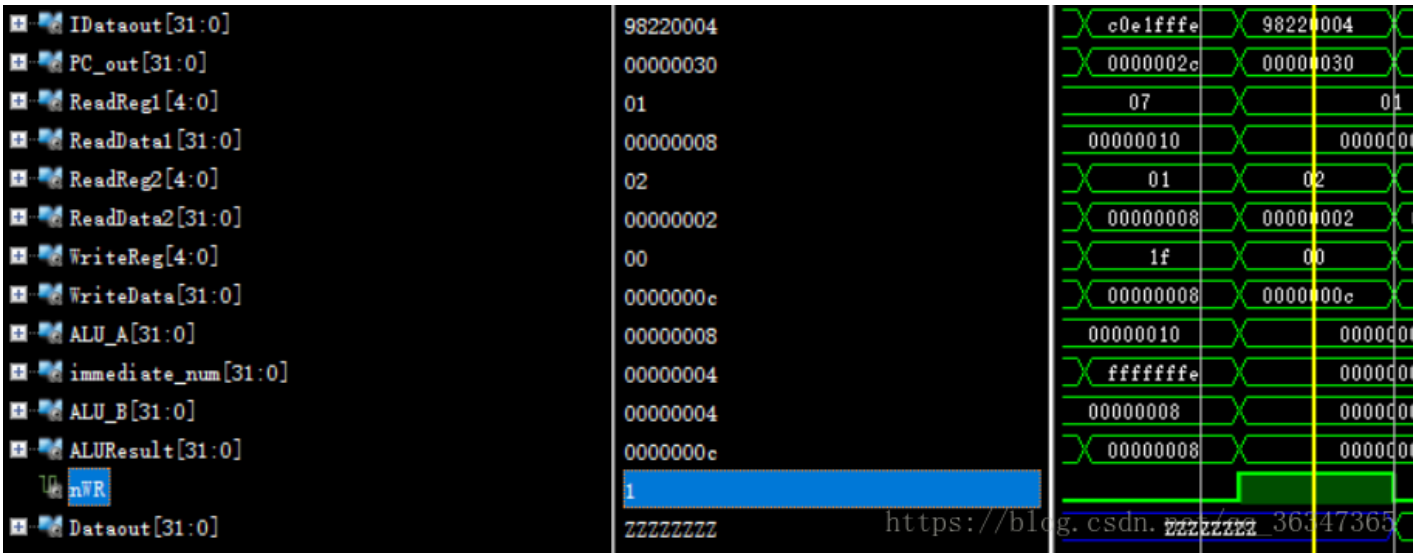
指令分析如下：



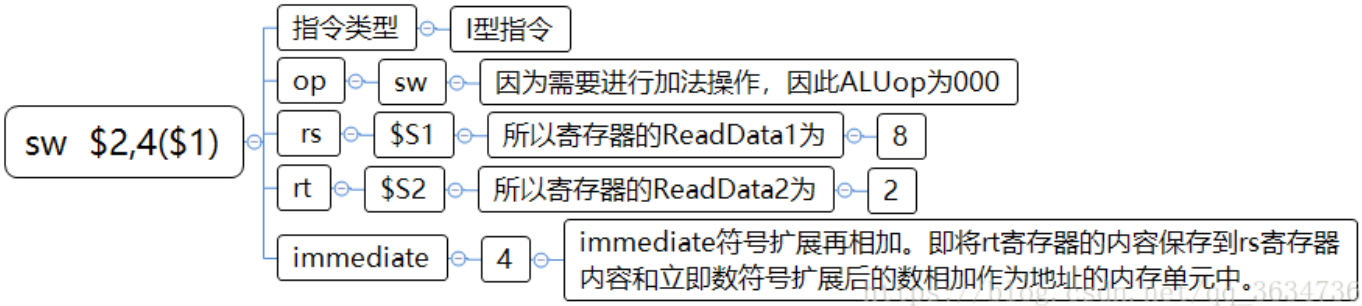
经过了上面的再一次自加之后$S7的数值变为了16，上面的ALU_A为16而ALU_B为8，这是正确的。而在做了减法之后，ALU逻辑运算其的结果为8，这也就导致了Zero转变为0，从而导致ControlUnit进行综合判断之后决定PCWre为00从而不再跳转而是使PC发生普通变化。上面显示的结果也是正确的。这个操作并未对寄存器进行修改操作，所以就不列表格了。

q) sw $2,4($1)

仿真结果如下：



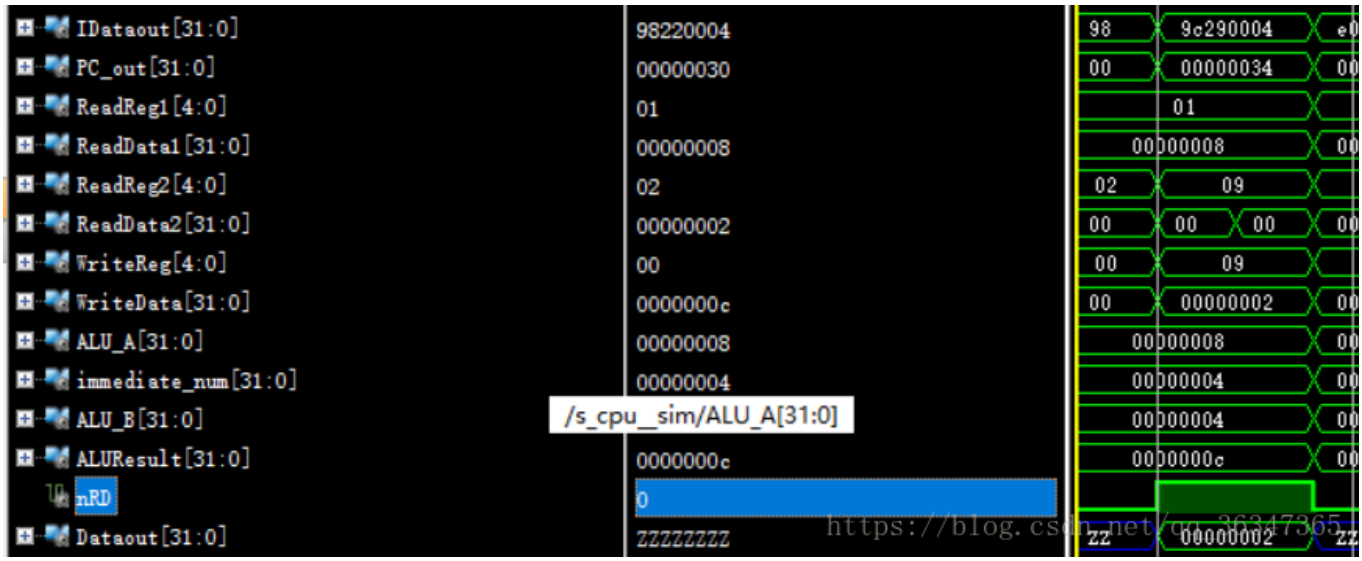
指令分析如下：



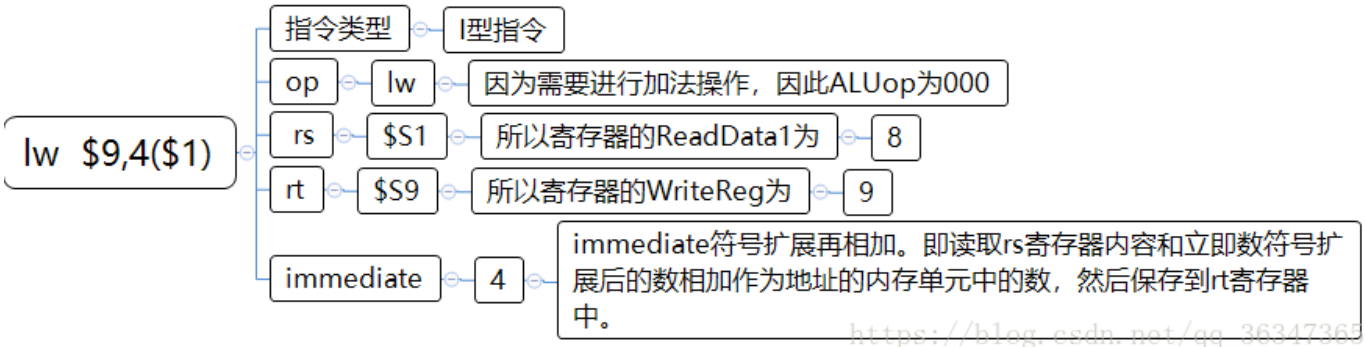
sw:是I型指令，是为了通过寄存地存放的信息作为地址然后在进行立即数变化之后找到相应的数据存储器中的元素之后将$S2的数据存到对应的位置上。所以ALU_A是8，即是$S1的值，而ALU_B则为4，来自于立即数，因为这需要进行加法操作，所以ALUop是000，然后经过ALU运算完之后结果是8+4 = 12，因为是写入数据存储器的操作，所以导致了nWR数值的变化，也就是转变为了1，然后在数据存储区中找到对应的位置并将数据写入。上面的数据是正确的，因为不涉及寄存器区域值的修改，所以此处不显示寄存器区域的数据。

r) lw $9,4($1)

仿真结果如下：



指令分析如下：



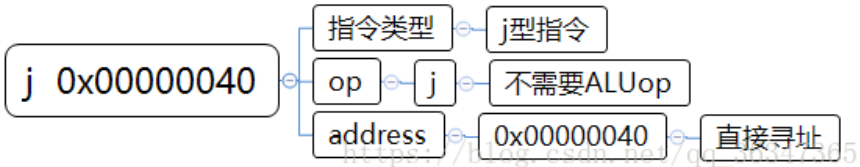
lw：这是一条I型指令，作用在于读取数据存储区的数据并加载到寄存器中。这里是将$S1中的数据作为基址然后以4，也就是立即数作为偏移量进行寻址，然后再将数据写到第9号寄存器中，所以这里ALU_A的数值为8，也就是1号寄存器中值，ALU_B为4，因为这也是一个加法操作，所以经过了ALU逻辑运算器之后，结果为12，然后就将数据传入到数据存储器中，此时因为是lw操作，所以控制信号nRD转变为1，使得DataOut从高阻态转变为对应的数值，也就是2，刚刚在上一步用sw存进去的数值，这是正确的。然后在通过数据选择之后写入寄存器中，所以WriteReg为9，WriteData为2。此时寄存器堆的数据为：

寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1	16	8	2	

s) j 0x00000040

仿真结果如下：

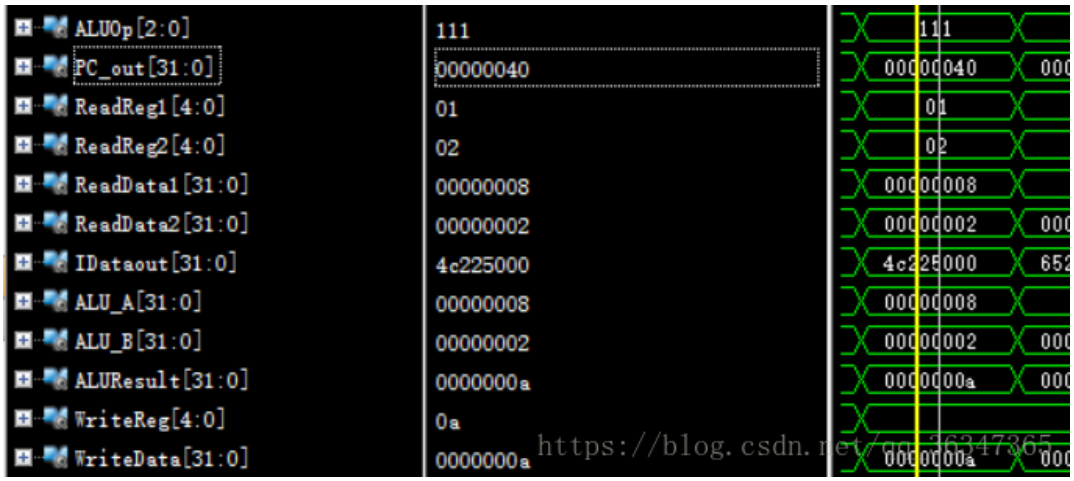
指令分析如下：



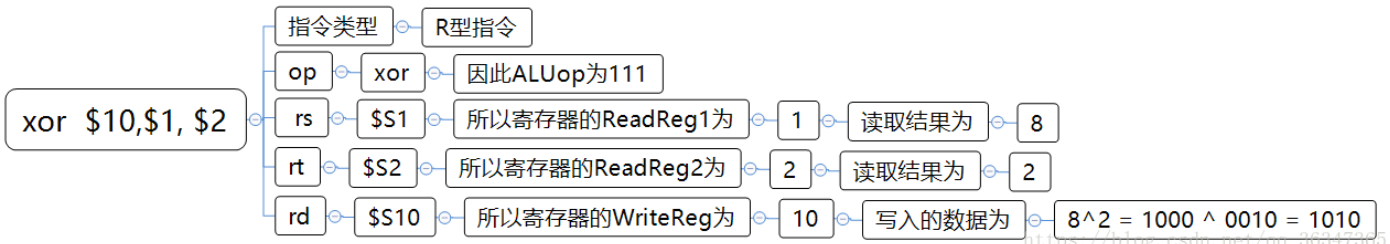
j: 这是J型指令，通过ControlUnit使得PcSrc发生改变变为2'b10进而使得PC发生无条件跳转，pc <－{(pc+4)[31..28],addr[27..2],2{0}}，此处的address为0x00000040，所以下一条指令的位置是0x0000040，根据上面的仿真图中可以看到PC的下一个状态是0x00000040，这是正确的。

t) xor $10,$1, $2

仿真结果如下：



指令分析如下：



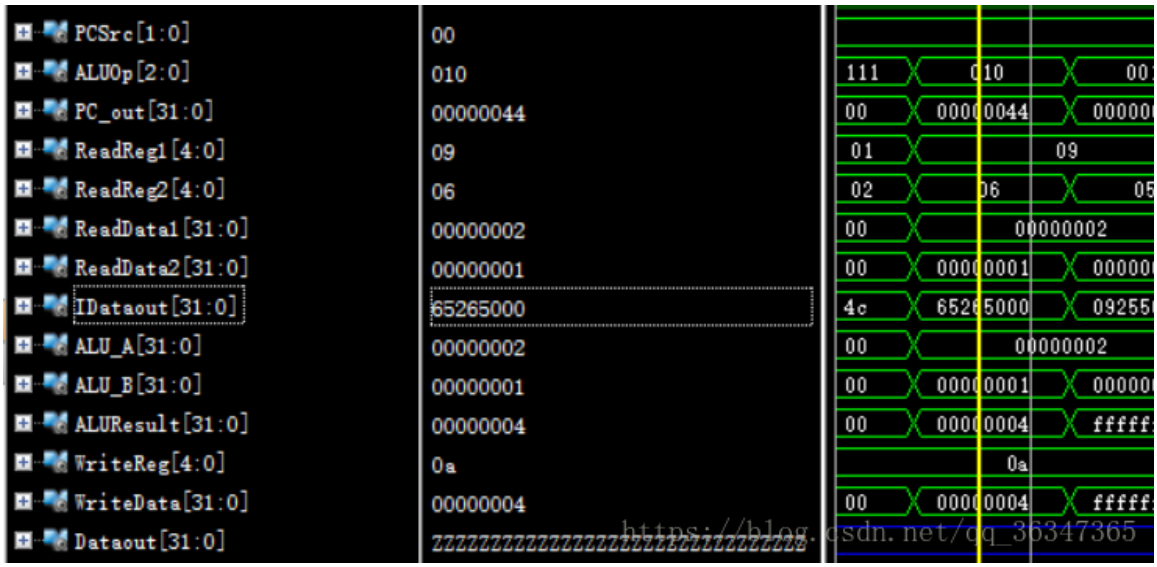
xor：R型指令。这一条指令写的原因是发现ALU中的异或方法并没有被利用。所以就添加了这一个指令，这个指令其实与or是一样的，也就只有在ControlUnit中需要对每个符合or的地方添加一下xor这一个判断就行了。然后再根据opcode对ALUop进行赋值就可以了。在此处其中rs是1，也就是ReadReg1为1，然后rt是2，ReadReg为2，从寄存器中取出对应的寄存器的值之后，得到ReadDate1为8，ReadDate为2，因为是R型指令，所以ALU的两端就是这两个值，通过异或运算之后得到结果为1000^0010 = 1010，然后写入寄存器中，对应的寄存器为WriteReg = rd = 10。所以上面的仿真是正确的。

此时寄存器堆的数据为：

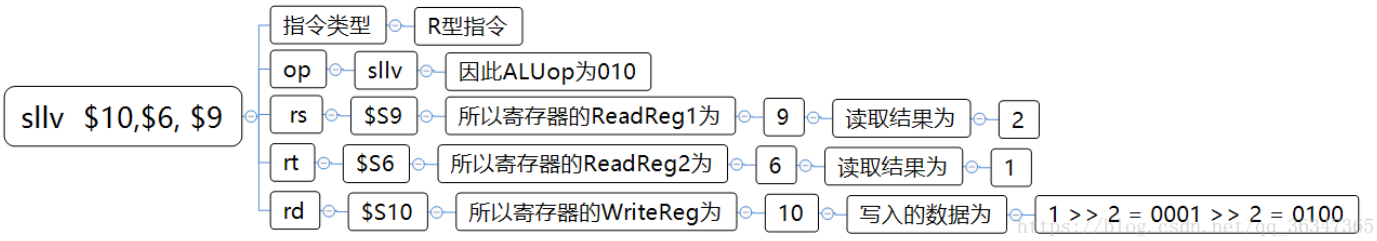
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1	16	8	2	10

u） sllv $10,$6, $9

仿真结果如下：



指令分析如下：



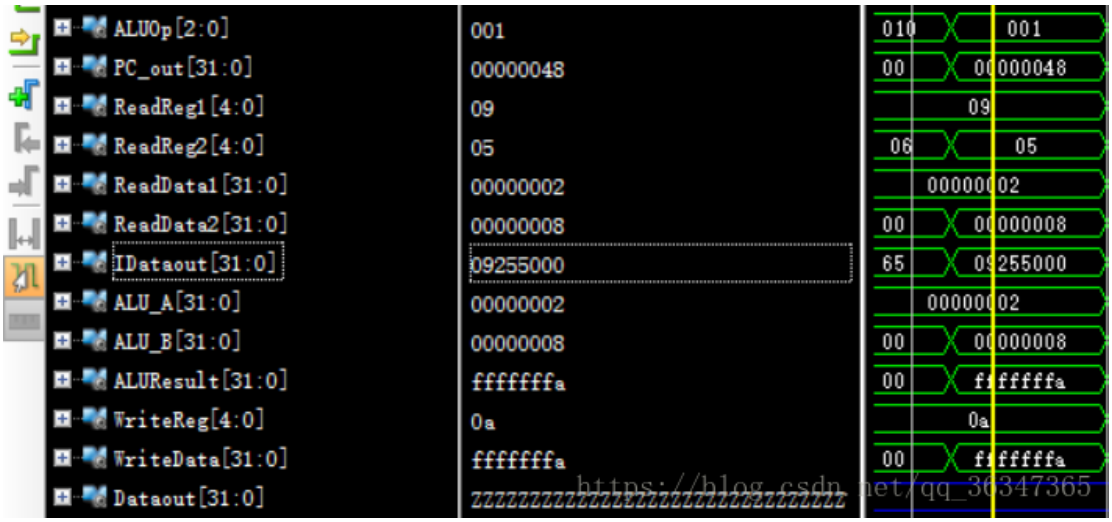
sllv： 这是R型指令。操作进行移位操作，与sll不同之处是用rs对应的寄存器中的数据作为sa来进行移位的。所以ALUop为3’b000，在此处其中rs是9，也就是ReadReg1为9，然后rt是6，ReadReg为6，从寄存器中取出对应的寄存器的值之后，得到ReadDate1为2，ReadDate为1，因为是R型指令，所以ALU的两端就是这两个值，通过移位运算之后得到结果为1 >> 2 = 0001 >> 2 = 0100 = 4，然后写入寄存器中，对应的寄存器为WriteReg = rd = 10。所以上面的仿真是正确的。

此时寄存器堆的数据为：

寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1	16	8	2	4

v） sub $10,$9, $5

仿真结果如下：



指令分析如下：

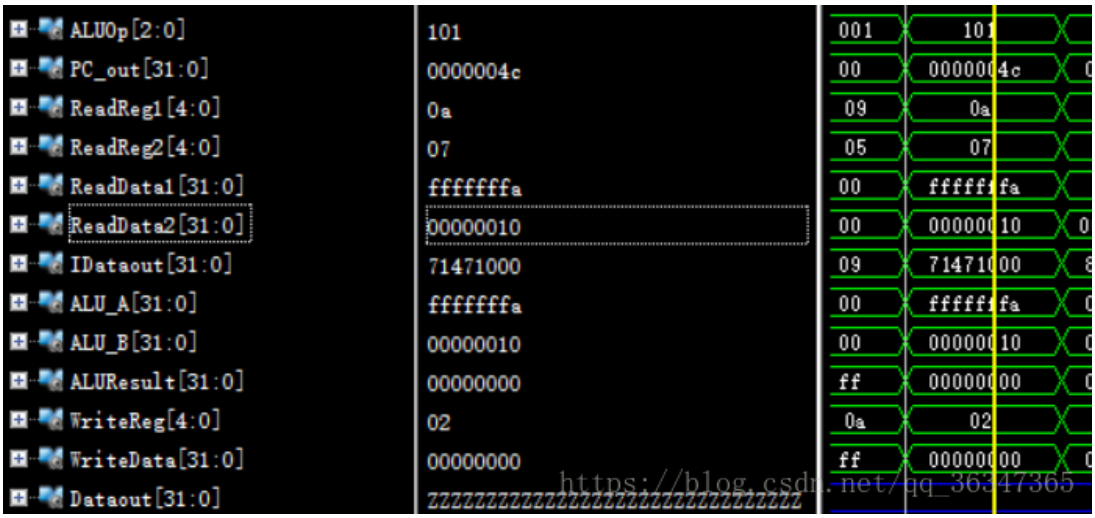
sub：这是R型指令，主要是为了后面的无符号比较做准备，因为为了体现无符号的比较，所以就将$10置为负数，所以就采用了sub的方法。这里rs、rt、rd分别为9、5和10，也就是ReadReg1为9，ReadReg2为5，WriteReg为10，从寄存器中读取出相应的数据后可得到ReadDate1为2，ReadDate2为8，因为是R型指令，所以ALU两端的数据也就是这两个了，两者做减法之后得到的结果是：2-8 = -6 = fffffffa（因为计算机是用补码来存储负数的），所以就将-6写入10号寄存器中。所以上面的仿真结果是正确的。

此时寄存器堆的数据为：

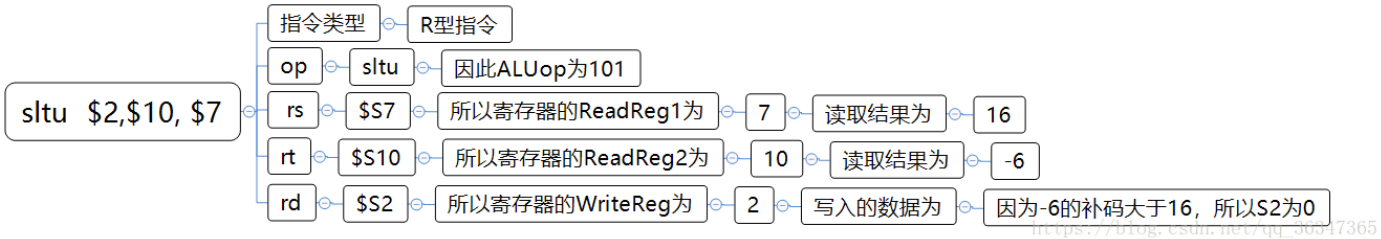
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	2	10	0	8	1	16	8	2	-6（补码存储）

w） sltu $2,$10, $7

仿真结果如下：



指令分析如下：



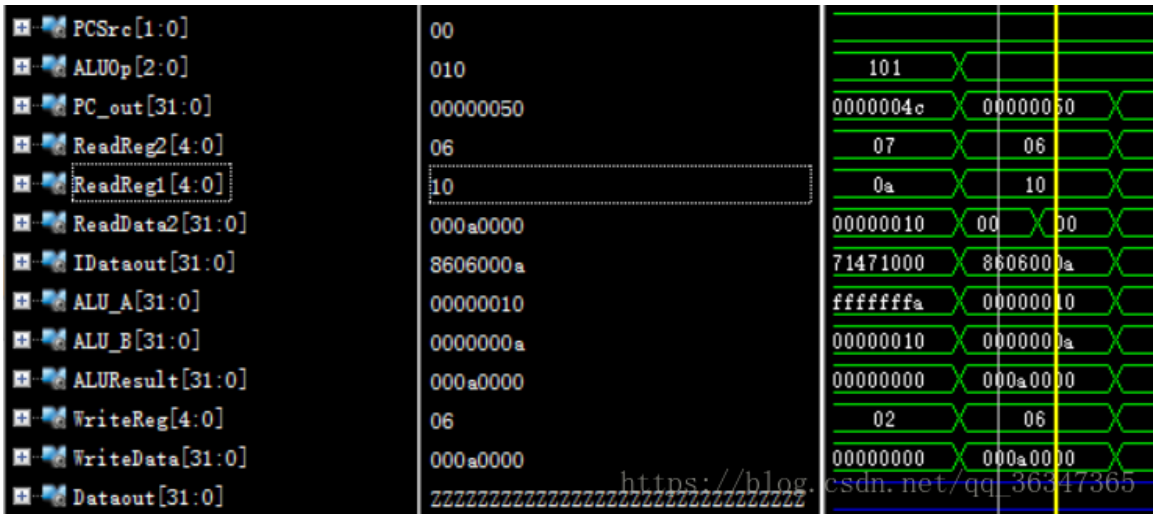
sltu：这是R型指令。写这个操作主要是因为前面的指令没有调用无符号比较的。主要进行的操作是比较两个无符号数的大小，当ReadDate1> ReadDate2的时候将目的寄存器的值赋为1。此处rs、rt、rd为7、10、2，所以对应的ReadReg1、 ReadReg2、WriteReg分别为7、10、2。从寄存器中取出数据之后得到ReadData1为-6，ReadData2为16，所以通过ALUop之后得到的结果为0，因为-6的补码为fffffffa，远大于16。所以写入2的数据为0。所以上面仿真的结果是正确的。

此时寄存器堆的数据为：

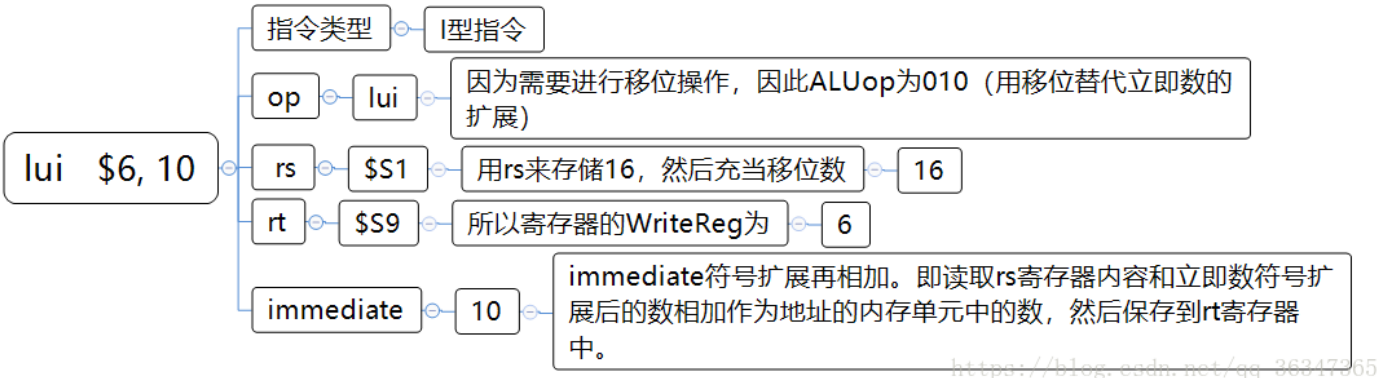
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	0	10	0	8	1	16	8	2	-6（补码存储）

x） lui $6, 10

仿真结果如下：



指令分析如下：



lui指令：这是一条I型指令。作用是用于直接将数据写入寄存器中，将立即数存放在对应的寄存器的高16位，然后对应的寄存器的低16位用0补充。我这里的实现方式是扩充ALUsrcA端口的选择器的选择项（因为实际需要额外添加一个模块进行赋值操作，那会更加麻烦一点，所以就使用了这一种不太严谨的方法），所以ALUsrcA就需要从一位改为两位。

```
output reg [1:0]ALUSrcA,
```

然后就是ALUSrcA的赋值操作需要添加一个判断了：

ALUSrcA [1..0]	00：来自寄存器堆data1输出，相关指令：add、sub、addi、or、and、ori、beq、bne、slti、sw、lw xor sltu sllv 01：来自移位数sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll 10：来自rs寄存器的ReadReg数值： lui 11：未用

所以这一块的代码改动如下：

```
assign PCWre = (op == halt) ? 0 : 1;
if(op == sll)
assign ALUSrcA = 2'b01;
else if(op == lui)
assign ALUSrcA = 2'b10;
else
assign ALUSrcA = 2'b00;
```

然后就是需要改动ALU_a端口的选择器了：

```
module num_to_ALU_a(
    input [31:0] ReadData1,
    input [1:0] ALUSrcA,
    input [4:0] sa,
    input [4:0] rs,
    output reg [31:0] rega
    );
    always @(ReadData1 or ALUSrcA or sa)begin
        if (ALUSrcA == 2'b00)    //ALUSrcA为0时，ALU的a侧数据来自寄存器
            assign rega = ReadData1;
        else if(ALUSrcA == 2'b01)                //ALUSrcA为1时，ALU的a侧数据来自指令中的偏移量sa
            assign rega = { 27'b000000000000000000000000000, sa};
        else
            assign rega = { 27'b000000000000000000000000000, rs};
    end
endmodule
```


将rs传递进来在进行扩展为32位数字就可以了。

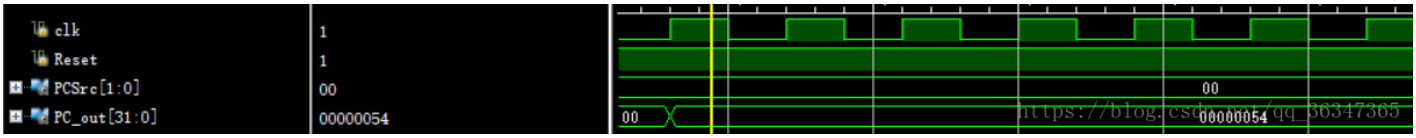
然后就利用左移操作，将ALU_b的数据向左移动16位来模仿真实的扩充操作。此处rt为9，立即数为10，根据ALU计算过后的结果是10>>16 = 1010 >> 16 = 655360也就是000a0000。所以上面的仿真结果是正确的。

此时寄存器堆的数据为：

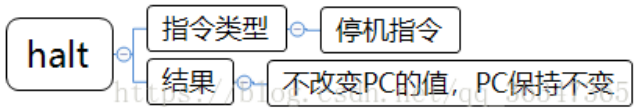
寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	0	10	0	8	655360	16	8	2	-6（补码存储）

y）halt（停机指令）

仿真结果如下：



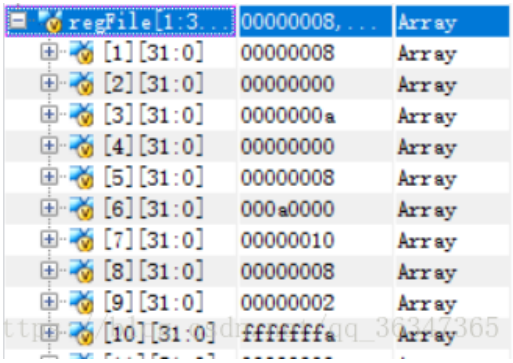
指令分析如下：



根据仿真结果显示，PC在停机之后不再发生变化，所以这是正确的。

从仿真中获得最后寄存器中每个寄存器的数值，可以看到与自己分析判断的是一样的。

仿真结果：



寄存器的内容：

寄存器	$S1	$S2	$S3	$S4	$S5	$S6	$S7	$S8	$S9	$S10
数值	8	0	10	0	8	655360	16	8	2	-6（补码存储）

到此为止，仿真模块已经完成了。

3、实现。如何在Basys3板上运行所设计的CPU？运行结果情况说明。

切记！以上内容，在书写实验报告时，必须删除，不能保留在自己的实验报告中。

这个部分也是需要进行分模块，分为LED七段数码管的显示以及以及顶层模块实现以及显示数据组成。其中LED七段数码管的显示部分老师已经提供了代码。

也就是：

```
always @( display_data ) begin
    case (display_data)
        4'b0000 : dispcode = 8'b11000000; //0锛?0'-浜  伅锛?1'-鐔勫伅
        4'b0001 : dispcode = 8'b11111001; //1
        4'b0010 : dispcode = 8'b10100100; //2
        4'b0011 : dispcode = 8'b10110000; //3
        4'b0100 : dispcode = 8'b10011001; //4
        4'b0101 : dispcode = 8'b10010010; //5
        4'b0110 : dispcode = 8'b10000010; //6
        4'b0111 : dispcode = 8'b11011000; //7
        4'b1000 : dispcode = 8'b10000000; //8
        4'b1001 : dispcode = 8'b10010000; //9
        4'b1010 : dispcode = 8'b10001000; //A
        4'b1011 : dispcode = 8'b10000011; //b
        4'b1100 : dispcode = 8'b11000110; //C
        4'b1101 : dispcode = 8'b10100001; //d
        4'b1110 : dispcode = 8'b10000110; //E
        4'b1111 : dispcode = 8'b10001110; //F
        default : dispcode = 8'b00000000; //涓嶅湪寒
    endcase
end
```


传入的数据是display_data是需要显示的数字。返回的值是对应的7段数码管的显示（另外一位是点的显示）。外面的仿真模块只要是做两个数据选择，一个是显示的内容的选择（button状态对应不同的显示），另一个是4个数字的显示利用扫描进行显示。

显示的内容的选择（button状态对应不同的显示）代码如下：

```
always @(*)
    begin
        case(state)
        0:begin
            displayed_number <= {8'b00000000, PC_out[7:0]};
        end
        1:begin
            displayed_number <= {3'b000, ReadReg1[4:0], ReadData1[7:0]};
        end
        2:begin
            displayed_number <= {3'b000,ReadReg2[4:0], ReadData2[7:0]};
        end
        3:begin
            displayed_number <= {ALUResult[7:0], DB[7:0]};
        end
        endcase
    end
```


另外扫描显示的部分代码如下：（其中display_data是用于筛选需要显示的数字的，LED_BCD是需要显示的数字）

```
always @(*)
    begin
        case(LED_activating_counter)
        2'b00: begin
            display_data = 4'b0111;
            LED_BCD = displayed_number[15:12];
        end
        2'b01: begin
            display_data = 4'b1011;
            LED_BCD = displayed_number[11:8];
        end
        2'b10: begin
            display_data = 4'b1101;
            LED_BCD = displayed_number[7:4];
        end
        2'b11: begin
            display_data = 4'b1110;
            LED_BCD = displayed_number[3:0];
        end
        endcase
    end
```

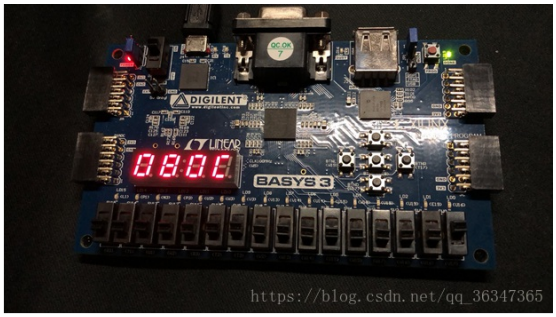

接着就是引脚了，我用的是直接写代码给引脚赋值的，而不是利用里面的design。

```
# Clock signal
set_property PACKAGE_PIN W5 [get_ports clock]
    set_property IOSTANDARD LVCMOS33 [get_ports clock]
set_property PACKAGE_PIN T18 [get_ports clk_]
    set_property IOSTANDARD LVCMOS33 [get_ports clk_]
    set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_]
#seven-ment LED display
set_property PACKAGE_PIN V7 [get_ports {LED_out[7]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[7]}]
set_property PACKAGE_PIN W7 [get_ports {LED_out[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[0]}]
set_property PACKAGE_PIN W6 [get_ports {LED_out[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[1]}]
set_property PACKAGE_PIN U8 [get_ports {LED_out[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[2]}]
set_property PACKAGE_PIN V8 [get_ports {LED_out[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[3]}]
set_property PACKAGE_PIN U5 [get_ports {LED_out[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[4]}]
set_property PACKAGE_PIN V5 [get_ports {LED_out[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[5]}]
set_property PACKAGE_PIN U7 [get_ports {LED_out[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[6]}]
set_property PACKAGE_PIN U2 [get_ports {display_data[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {display_data[0]}]
set_property PACKAGE_PIN U4 [get_ports {display_data[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {display_data[1]}]
set_property PACKAGE_PIN V4 [get_ports {display_data[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {display_data[2]}]
set_property PACKAGE_PIN W4 [get_ports {display_data[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {display_data[3]}]
set_property PACKAGE_PIN V17 [get_ports Reset]
    set_property IOSTANDARD LVCMOS33 [get_ports Reset]
set_property PACKAGE_PIN T1 [get_ports {state[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {state[0]}]
set_property PACKAGE_PIN R2 [get_ports {state[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {state[1]}]
```

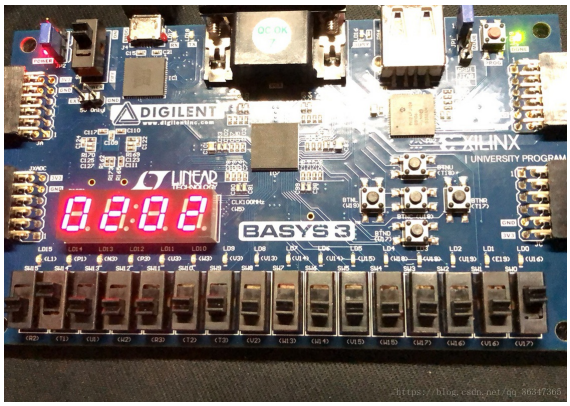

这个过程有很多的坑，比如需要该很多代码的赋值<=，同时还要面对各种资源不够来进行修改，而且因为多个文件而改错文件的风险，同时还有没有变量声明但是不会有错误信息然后要自己花上半天的时间去寻找这个问题，然后添加一下声明就完成了。然后接下来就是展示板子了。（因为怕板子的资源不够而且指令是后面写的所以自己添加的指令没有烧进去）

其中最右下角的开关为Reset，第二个为clk开关，而最左下角为两个数据选择端。

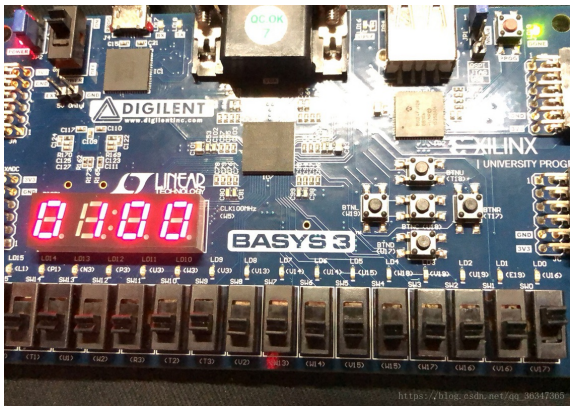
00时的PC显示：



01时的rs寄存器以及数值：（0202）

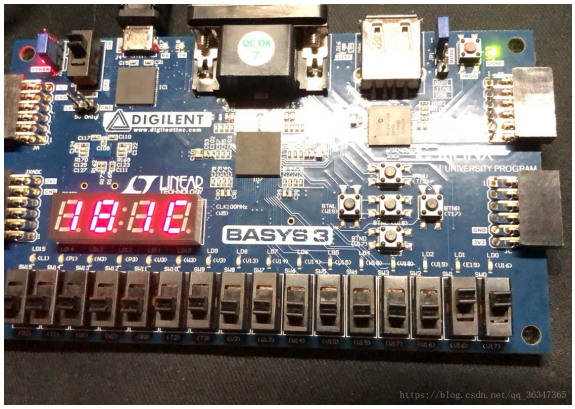


10时的rt寄存器以及寄存器中的数值：

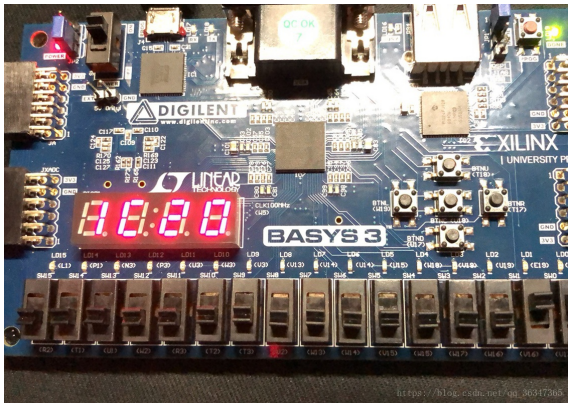


跳转指令：

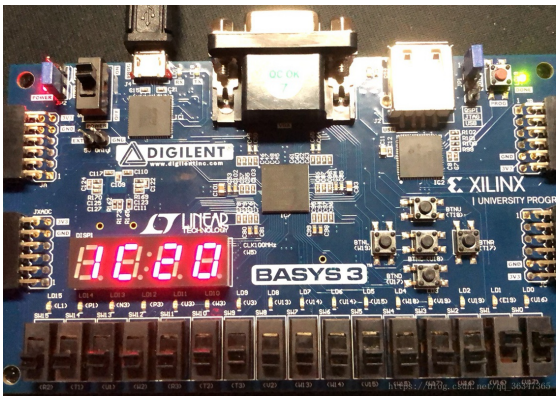
（18-1C）



1C->18



1C->20（跳出判断语句）


DIGILENT
XILINX
BASYS 3