中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验三）

学 院 名 称	：	数据科学与计算机学院
专业（班级）	：	18 计教学三班
学 生 姓 名	：	田蕊
学　　号	：	18340161
时　　间	：	2019　年　11　月　24　日

成 绩 ：

实验三 ： 单周期CPU设计与实现

一. 实验目的

1. 熟悉单周期CPU的数据通路的构造，原理和设计方法
2. 认识和掌握单周期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. 学会vivado和modelsim的使用方法

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

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100000

功能：GPR[rd] ← GPR[rs] + GPR[rt]。

（2）sub rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100010

功能：GPR[rd] ← GPR[rs] - GPR[rt]。

（3）addiu rt , rs ,**immediate**

001001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] + sign_extend(**immediate**)； **immediate** 做符号扩展再参加“与”运算。

==> 逻辑运算指令

（4）andi rt , rs ,**immediate**

001100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] and zero_extend(**immediate**)；**immediate** 做 0 扩展再参加“与”运算。

（5）and rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100100

功能：GPR[rd] ← GPR[rs] and GPR[rt]。

（6）ori rt , rs ,**immediate**

001101	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] or zero_extend(**immediate**)。

（7）or rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100101

功能：GPR[rd] ← GPR[rs] or GPR[rt]。

==>移位指令

（8）sll rd, rt,sa

000000	00000	rt(5 位)	rd(5 位)	sa(5 位)	000000

功能：GPR[rd] ← GPR[rt] << sa。

==>比较指令

（9） slti rt, rs,**immediate** 带符号数

001010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if GPR[rs] < sign_extend(**immediate**) GPR[rt] =1 else GPR[rt] = 0。

==> 存储器读/写指令

（10）sw rt , **offset** (rs) 写存储器

101011	rs(5 位)	rt(5 位)	**offset**(16 位)

功能：memory[GPR[base] + sign_extend(**offset**)] ← GPR[rt]。

（11) lw rt , **offset** (rs) 读存储器

100011	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：GPR[rt] ← memory[GPR[base] + sign_extend(**offset**)]。

==> 分支指令

（12）beq rs,rt, **offset**

000100	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] = GPR[rt]) pc←pc + 4 + sign_extend(**offset**)<<2

else pc ←pc + 4

特别说明：**offset 是从 PC+4 地址开始和转移到的指令之间指令条数。offset** 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 **offset** 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, **offset**

000101	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] != GPR[rt]) pc←pc + 4 + sign_extend(**offset**) <<2

else pc ←pc + 4

（14）bltz rs, **offset**

000001	rs(5 位)	00000	**offset** (16 位)

功能：if(GPR[rs] < 0) pc←pc + 4 + sign_extend (**offset**) <<2

else pc ←pc + 4。

==>跳转指令

（15）j addr

000010	**addr**(26 位)

功能：PC ← {PC[31:28] , **addr** , 2’b0}，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，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（16）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

（一）、概述

单周期CPU指的是一条指令的完成都在一个时钟周期内，然后再开始下一个时钟周期。

CPU在处理指令时，需要经过如下步骤:

（1）、取指令

存指令存储器中读取指令到内存中

（2）、指令译码

对指令进行译码产生控制信号，有控制信号对后续的指令执行进行控制

（3）、指令执行

根据控制信号具体地执行指令动作

（4）、访问存储器

对于需要访问存储器的操作在这里进行，该步骤给出存储器的数据地址，把数据写入存储器或读出该地址的数据。

（5）、结果写回

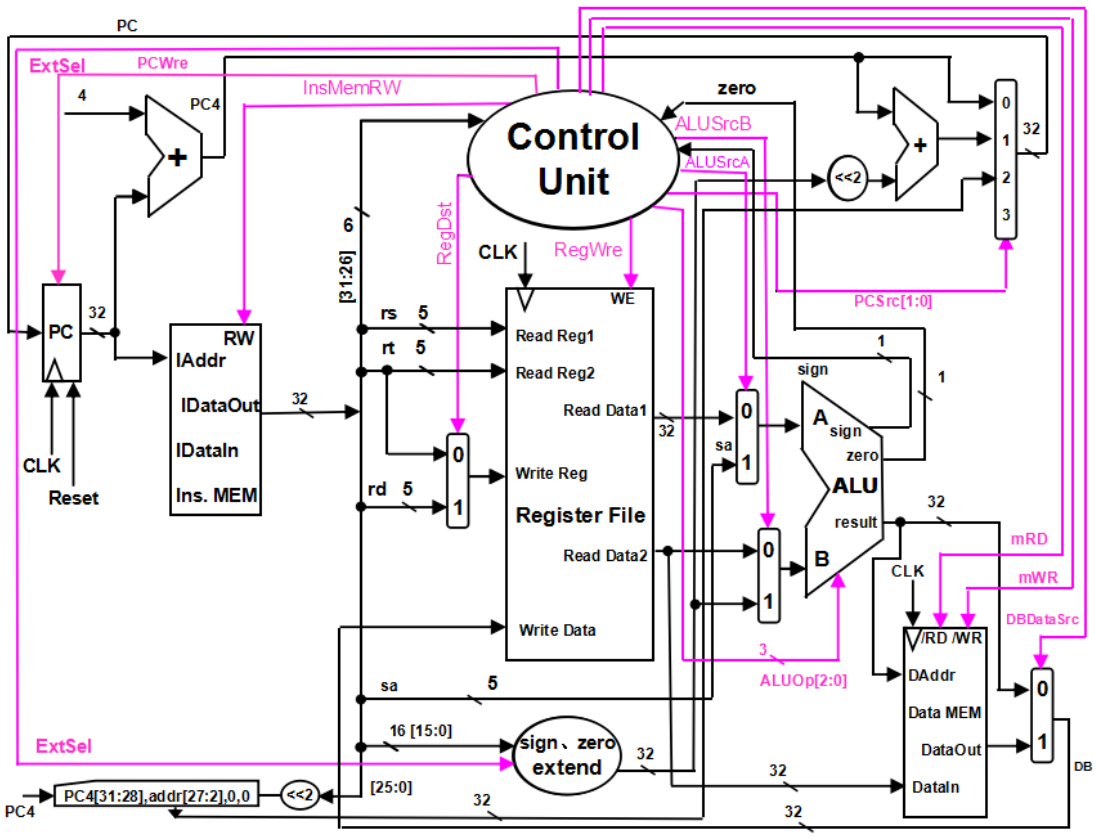
指令执行的结果或者存储器中得到的数据写回到相应的目的寄存器中。

对于单周期CPU来说，这些执行步骤均在一个时钟周期内完成。

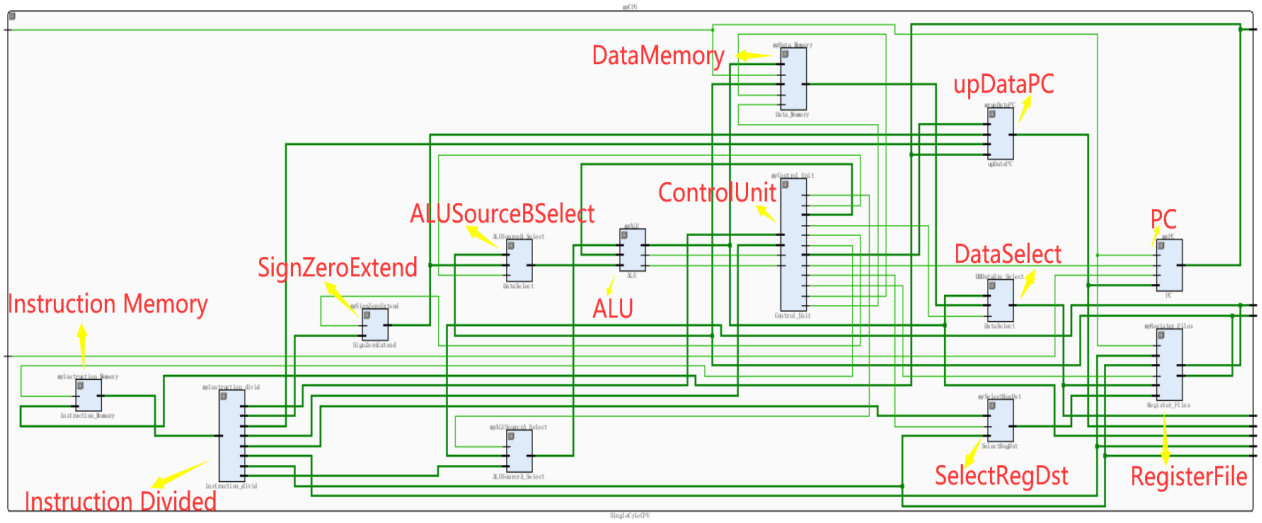
图 1　单周期 CPU 指令处理过程

（二）、单周期CPU的数据通路

本实验设计的单周期CPU的数据通路原理图如下所示



实际实现的数据通路结果图如下



（三）、控制信号的产生

本实验涉及到的控制信号包括Reset，PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRw, mRD, mWR, RegDst, ExtSel, PCSrc[1:0], ALUop[2:0].

1、Reset：

状态为0：初始化PC为0

状态为1：PC接受新地址

2、PCWre：

状态为0：PC不更改，相关指令：halt

状态为1：PC更改，相关指令：除指令halt外

3、ALUSrcA：

状态为0：来自寄存器堆data1输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw

状态为1：来自移位数sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll

4、ALUSrcB：

状态为0：来自寄存器堆data2输出，相关指令：add、sub、or、and、beq、bne、bltz

状态为1：来自sign或zero扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw

5、DBDataSrc：

状态为0：来自ALU运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll

状态为1：来自数据存储器（Data MEM）的输出，相关指令：lw

RegWre：

状态为0：无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt

状态为1：寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw

6、InsMemRW：

状态为0：写指令存储器

状态为1：读指令存储器(Ins. Data)

7、mRD：

状态为0：输出高阻态

状态为1：读数据存储器，相关指令：lw

8、mWR：

状态为0：无操作

状态为1：写数据存储器，相关指令：sw

9、RegDst：

状态为0：写寄存器组寄存器的地址，来自rt字段，相关指令：addiu、andi、ori、slti、lw

状态为1：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

10、ExtSel：

状态为0：(zero-extend)immediate（0扩展），相关指令：addiu、andi、ori

状态为1：(sign-extend)immediate（符号扩展），相关指令：slti、sw、lw、beq、bne、bltz

11、PCSrc[1..0]：

00：pc<－pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)；

01：pc<－pc+4+(sign-extend)immediate，相关指令：beq(zero==1)、bne(zero==0)、bltz(sign==1);

10：pc<- {(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j；

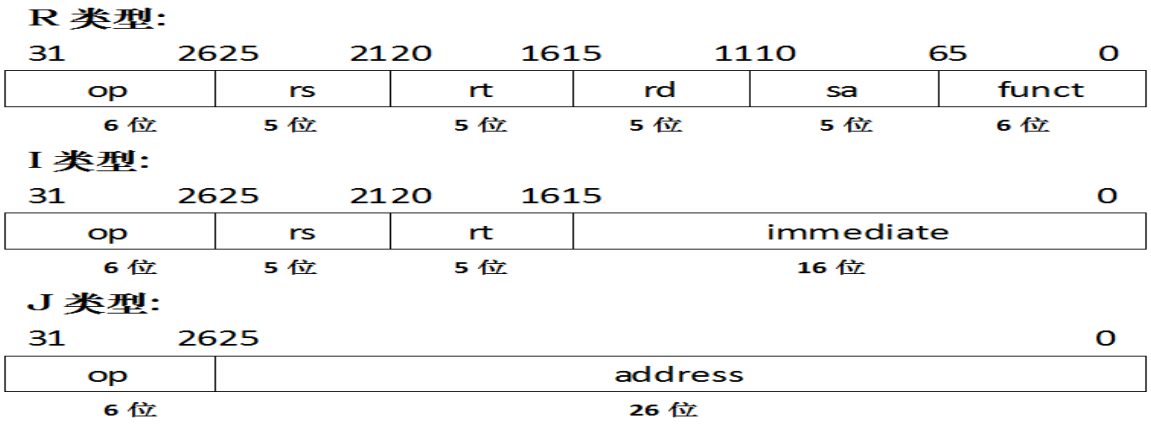
11：未用

12、ALUOp[2..0]:

ALU 8种运算功能选择(000-111)

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

（四）、MIPS指令格式化



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（Laod）/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。

待实现的指令中，add, sub, and, or, sll为R型指令；addiu, andi, ori, slti, sw, lw, beq, bne, bltz为I型指令，j为J型指令。

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

（一）、CPU设计的思想，方法

1、综述

CPU设计的总体思路是根据各个部件的输入，输出及其内部功能确定各个连线之间的内部逻辑，再用硬件描述语言将这个逻辑表达出来。CPU的设计我们将其分为各个功能部件分别设计，这样在不同的功能部件之间只需要考虑接口的设计，而不需要关心内部的连线实现与关系。

2、流程图的应用

根据流程图我们可以分析出各个部件之间的接口联系，根据这些接口我们可以确定

各个部件在设计的时候所需要的输入和输出。在设计好各个部件接口的输入和输出之后，再根据流程图将各个部件连接起来，就可以得到我们最终的CPU。

3、控制信号的应用

控制信号是根据各个指令的op和func部分得到的。对于R型指令，我们根据func部分推断出各个控制信号的取值；对于非R型指令，我们根据op部分推断出各个控制信号的取值。得到控制信号，我们就相当于得到了一张真值表，根据真值表的内容我们可以设计出ALU的逻辑，更新PC的数据来源，DataMemory的来源，RegisterFile的数据来源和目的寄存器的地址等等一些列的部件的控制。同时，在分析出控制信号的取值之后，控制单元也可以根据控制信号的取值逻辑来表示。所以本实验最重要的两个材料——数据通路和控制信号的取值，一点确定了下来，本实验就基本上成功了一半了。

	add	sub	addiu	andi	and	ori	or	sll	slti	sw	lw	beq	bne	bltz	j	halt
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
PCWre	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
ALUSrcA	0	0	0	0	0	0	0	1	0	0	0	0	0	0	X	X
ALUSrcB	0	0	1	1	0	1	0	0	1	1	1	0	0	0	X	X
DBDataSrc	0	0	0	0	0	0	0	0	0	X	1	X	X	X	X	X
RegWre	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	X
InsMemRw	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	X
mRD	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	X
mWR	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	X
RegDst	1	1	0	0	1	0	1	1	0	X	0	X	X	X	X	X
ExtSel	X	X	1	0	X	0	X	0	1	1	1	1	1	1	X	X
PCSrc[1:0]	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	zero=0->00 zero=1->01	zero=1->00 zero=0->01	sign=0->00 sign=1->01	1 0	X
ALUop[2:0]	0 0 0	0 0 1	0 0 0	1 0 0	1 0 0	0 1 1	0 0 1	0 1 0	1 1 0	0 0 0	0 0 0	0 0 1	0 0 1	1 1 0	XXX	XXX

▲控制信号的取值

4、各个部件的设计

（1）、符号扩展部件

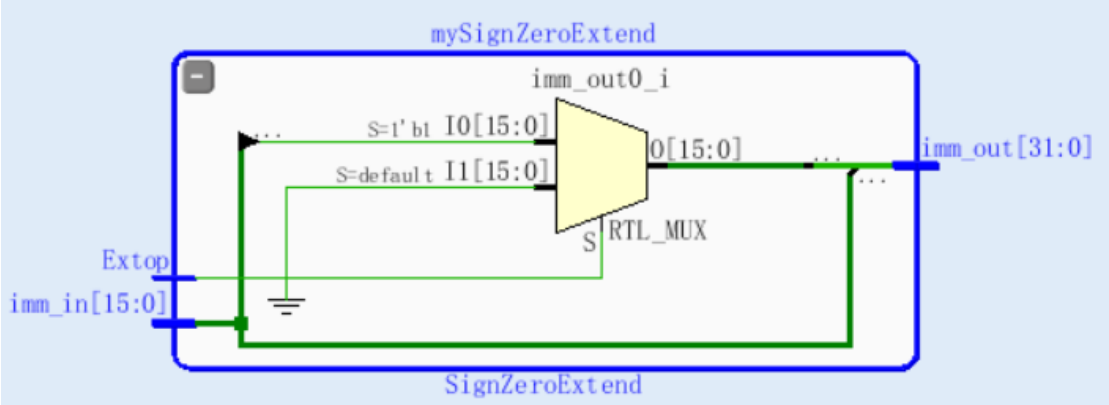
我们先从最简单的部件——符号扩展部件入手。

符号扩展部件就是决定对数据进行零扩展还是符号扩展。这个的确定要根据运算的不同，逻辑运算需要进行零扩展，而其他运算需要进行符号扩展。但是在具体实现的过程中，只需要根据控制单元内产生的控制信号就可以决定扩展方式。代码部分关于输入输出的设置我们暂且略去不谈，关于reg类型变量和wire类型变量的区别我们会在心得的部分再详细展开，对于符号扩展部分的主体部分，如下

```
always@(*)
begin
    imm_out[15:0] = imm_in;
    imm_out[31:16] = Extop?(imm_in[15]?16'hffff:16'h0000):16'h0000;
end
```

运用的语句还是在整个CPU设计过程中非常常用的 ？ ：语句。如果控制单元产生的控制信

号要求我们进行符号扩展，那么就根据待扩展的立即数首位进行扩展，否则就进行零扩展。

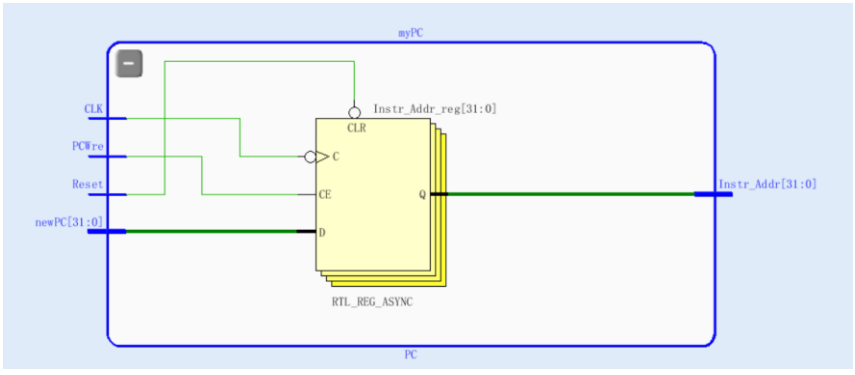


（2）、PC部分

我的CPU将PC设计成了两个部分，就是向PC里写入数据和更新PC分开成两个部件。在PC部分，只需要根PCWre和Reset的控制信号决定是否更改PC的值即可。需要注意的是Reset的控制信号是非常优先的，无论何时，只要Reset清零，那么PC的值都需要立刻清零。所以在触发的控制信号部分，Reset的下降沿也应该作为控制信号。在Reset为0时，PC值立刻清零，否则，根据PCWre的值来确定PC值是否更新。我们整个CPU的设计采用上升沿时数据写入PC，下降沿时对寄存器组和内存进行读写。

PC的写入部件主体内容如下所示

```
//Reset是很优先的，只要Reset变成了0，立刻清
always@(negedge Reset or posedge CLK)
begin
    if(Reset==0)//如果要清零的话，PC值全部赋为0
        Instr_Addr[31:0] = 32'h00000000;
    else
    begin
        if(PCWre==1)//如果PC的写使能为真，那么就要写入新的PC
            Instr_Addr[31:0] = newPC;
    end
end
```



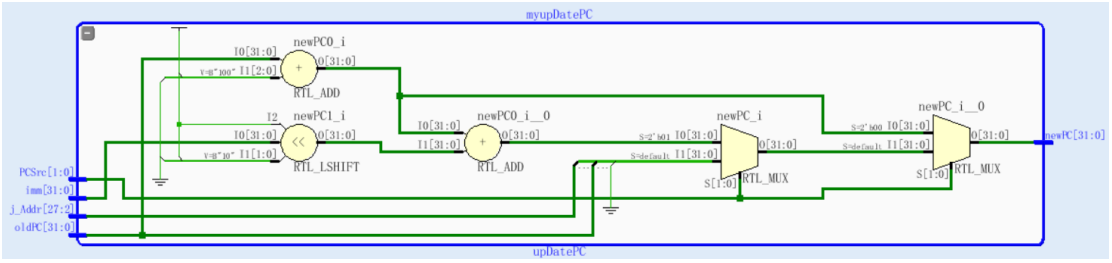
（3）、PC的更新部分

对于PC的更新，我们要根据控制单元产生的PCSrc来决定更新PC的值的来源，PC的来源可能是jump指令给出的伪直接寻址的地址，也可能是brench指令的偏移寻址的地址，还可能是原始PC值自增。这个具体的来源只需要根据控制单元的译码结果就可以确定。

如果PCSrc为00，则新的PC来自PC的自增，如果PCSrc为01，则新的PC来自brench的偏移寻址的结果，如果PCSrc为10，则新的PC来自jump指令。根据这个逻辑则可以得到PC更新这一部件的设计。

```
always@(*)
begin
    if(PCSrc==2'b00)
        newPC[31:0] = oldPC + 4;
    else if(PCSrc==2'b01)
        newPC[31:0] = oldPC + 4 + (imm<<2);
    else
        newPC[31:0] = {oldPC[31:28],j_Addr[27:2],2'b00};
end
```

PC更新这一部件的内部设计框图如下所示



（4）、指令存储器部分

指令存储器内需要保存我们的所有指令。在这个CPU的设计里面我们是用寄存器的数组来进行内存的模拟的。每个寄存器的位宽为8位，每一个数据的存取都需要的四个寄存器单元进行操作，用256个寄存器来模拟内存。我们整个CPU的设计采用大端方式设计，需要提一下的是，其实大端方式和小端方式究竟采用哪一个是没有关系的，但是对于整个项目我们应该采用同一种模式，即在内部应该相互一致，否则读取的结果是不正确的。

在这一部分我们将翻译好的指令保存在一个文件中，在初始化寄存器数组的时候将指令读入内存，这里采用的是verilog的系统函数$readmemh，这个函数是以16进制读入，所以在我们保存的指令的文件中，指令应该以十六进制保存，同时因为寄存器的位宽是八位的，所以我们的指令文件中每一行只能放两个十六进制数字。

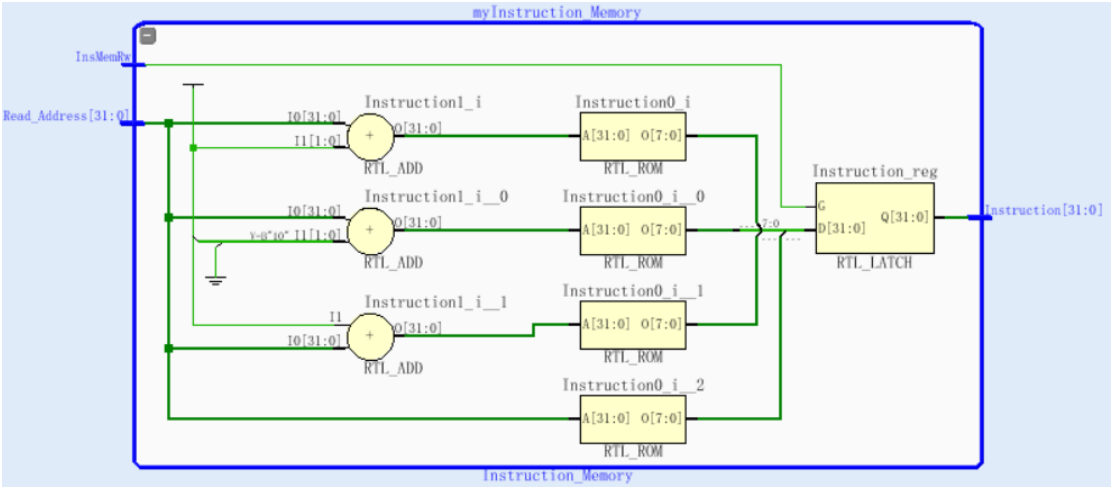
```
//从文件中读取指令
initial
    begin
        $readmemh("E:/Instruction.txt",memory);
        //$display(memory[0]);
    end
```

▲从文件中读取指令到内存中

之后的步骤就是将指令读出，根据大端方式的连线只需要将对应数据的对应位宽进行连接即可。在我们所涉及到的指令中，所有的指令InsMemRw控制信号都是要取1的，但是考虑到程序的合理性，我们在进行读取时仍然对InsMemRw控制信号进行判断。

```
always@(*)
    begin
        //将内存中的地址读出
        if(InsMemRw)
            begin
                Instruction[7:0] = memory[Read_Address + 3];
                Instruction[15:8] = memory[Read_Address + 2];
                Instruction[23:16] = memory[Read_Address + 1];
                Instruction[31:24] = memory[Read_Address + 0];
            end
    end
```

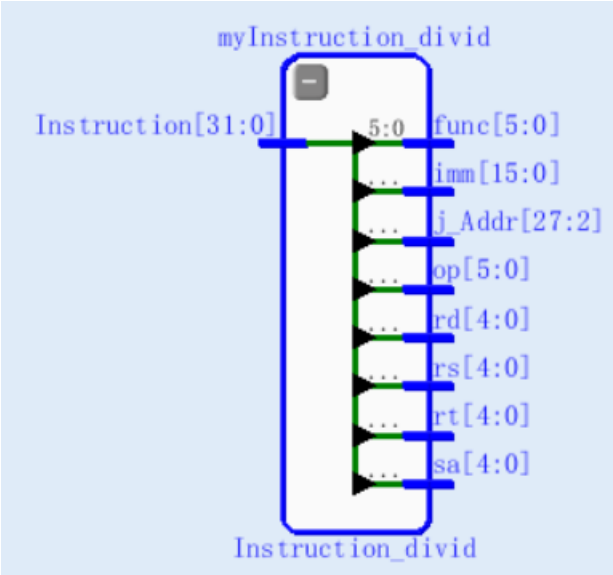
▲指令内存读取的部分



▲指令存储器的内部框图

（5）、指令分割

在CPU的设计中，我将指令的分割和存储器分离开来，而不是合在了同一个部件里，这样做的主要考量是方便调试阶段的处理。这里将指令分割出op, rs, rt, rd, sa, func, imm, j_Addr几个部分。再将对应的部分分别送入到各个部件里面去。这一部分的代码十分简单，只要根据MIPS的指令格式进行对应的连线即可。



▲指令分割器内部框图

（6）寄存器组部件

对于寄存器组部分输入为Read_reg1, Read_reg2, Write_reg, RegWre, CLK和Write_Data. 输出为Read_Data1, Read_Data2. 这里运用32个32位宽的寄存器数组来表示寄存器堆。在最开始，将寄存器中的数值全部赋为零。在读取寄存器组中的数值的时候不用等时钟信号，可以直接读取。需要注意的是，如果读取的目的寄存器是零号寄存器，则一定会读出数据0.

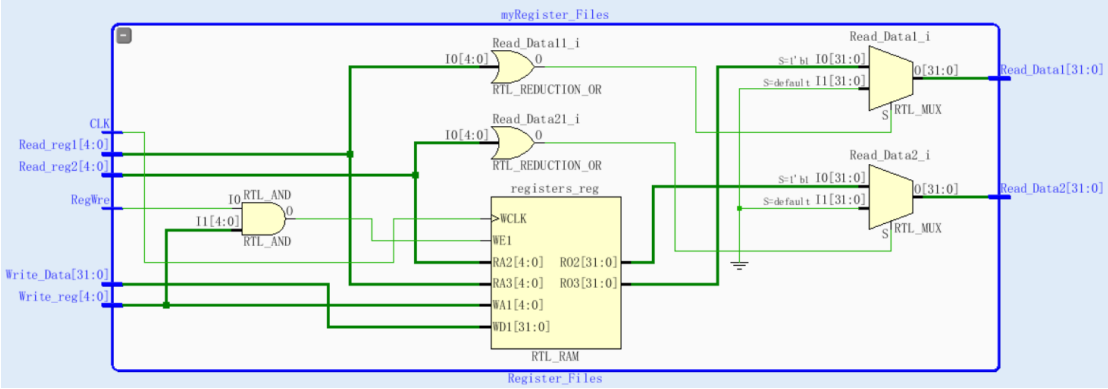
```
//不用等信号，可以直接读
assign Read_data1 = Read_reg1?registers[Read_reg1]:0;
assign Read_data2 = Read_reg2?registers[Read_reg2]:0;
```

▲读寄存器数值

在向寄存器数组里面写的时候，需要等时钟信号，当下降沿到来的时候，如果RegWre信号为1，且目的寄存器不是零号寄存器，那么就可以将数据写入寄存器组中。因为零号寄存器是不能写的，它的值永远为0.

```
//如果要写的话，要等时钟信号，这里用时钟的下降沿
always@(negedge CLK)
begin
    if(RegWre && Write_reg )//要注意零号寄存器是不能写的，所以条件判断的地方要加上Write_reg来判断目的寄存器是否为零号寄存器
        registers[Write_reg] <= Write_Data;//如果目的寄存器不是零号寄存器，就在时钟到来的时候，把数据写进寄存器
end
```

▲写寄存器



▲寄存器堆的内部框图

（7）、ALU部分

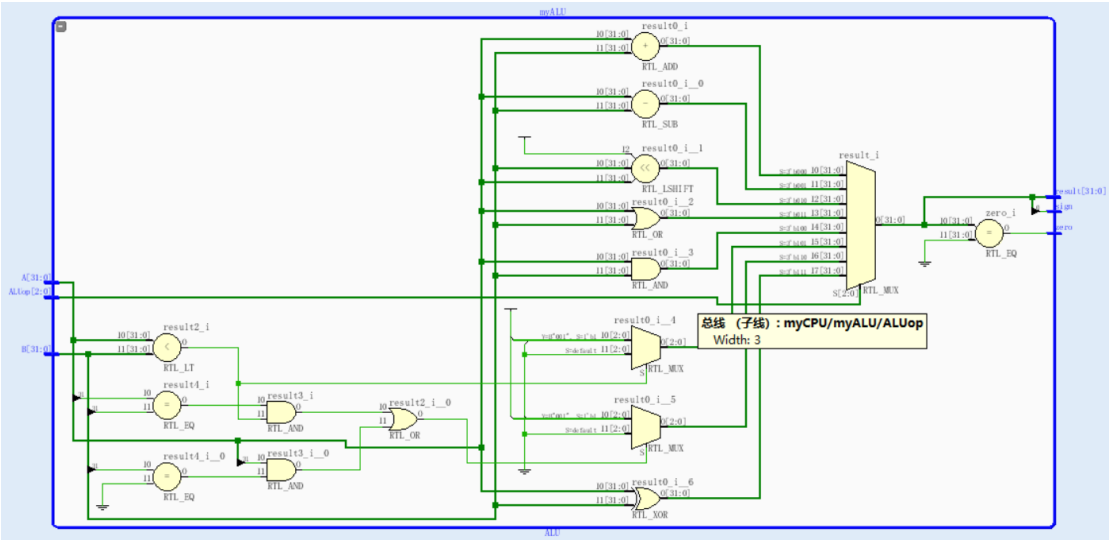
ALU部分就是根据控制单元的ALUop的译码结果，对输入的数据进行操作，上述已经列出了不同的ALUop的编码对应的算术运算，在这里只需要运用一个case语句就可以实现ALU功能。同时，为了方便后面bne, beq, bltz三条指令中PCSrc的判断，还附加了两个输出是zero，和sign。当两个操作数相减结果为零时，zero为1，否则zero为0；当寄存器中的值为负数时，sign为1，否则sign为0.所以zero只要根据result的结果来赋值即可，但是这里必须要提一下的是sign的赋值。我们的ALU操作中时候包括有符号数的比较的。对于bltz指令，我们是将第一个寄存器中的内容和零号寄存器，也就是数字零进行比较，当小于时，result结果赋为1，大于零时，result结果赋为0. 所以其实result的结果就是sign的结果，也就是说sign应该为result的第零位数据结果，而不是最高位数据结果。其实sign是不必要的，但是为了方便判断，我们还是定义了一个sign变量。

```
assign zero = (result==0);//零标志位标志结果是否为零
assign sign = result[0];
```

▲为zero和sign位赋值

```
always@(*)
begin
    case(ALUop)
        3'b000 : result = A + B;
        3'b001 : result = A - B;
        3'b010 : result = B << A;
        3'b011 : result = A | B;
        3'b100 : result = A & B;
        3'b101 : result = (A<B) ? 1 : 0;
        3'b110 : result = (((A[31]==B[31])&&(A<B))||(A[31]==1&&B[31]==0))?1:0;
        3'b111 : result = A ^ B;
        default: result = 32'h00000000;
    endcase
end
```

▲ALU运算



▲ALU部分的内部框图

（8）、数据存储器部分

和指令存储器类似，我们仍然是采用256个位宽为8的寄存器数组来模拟存储器。这里仍然采用大端方式存储。在最开始的时候我们仍然是将数据存储器中的值全部赋为零。在数据存储器的设计当中我们规定如果向寄存器中写入数据则需要等待时钟，如果从寄存器中读取数据，则不需要等待时钟，只要地址和读使能有效就可以读出。

```
//我们规定如果是写的话要等时钟，大端方式哦
always@(negedge CLK)
begin
    if(mWR==1)
    begin
    dataMemory[Addr+3] <=Data_in[7:0];
    dataMemory[Addr+2] <=Data_in[15:8];
    dataMemory[Addr+1] <=Data_in[23:16];
    dataMemory[Addr+0] <=Data_in[31:24];
    end
end
```

▲向存储器写入数值

```
always@(*)
begin
    if(mRD==1)
    begin
        Data_out[7:0] <= dataMemory[Addr + 3];
        Data_out[15:8] <= dataMemory[Addr + 2];
        Data_out[23:16] <= dataMemory[Addr + 1];
        Data_out[31:24] <= dataMemory[Addr + 0];
    end
end
```

▲从存储器读取数值

（9）、控制单元

接下来使我们的终极大boss ，控制单元。控制单元可以说是我们整个CPU设计最重要的一个部分，控制单元产生的控制信号，控制各个部件的运转。之前已经列出了各个控制信号的取值，其实只要根据上面列出的表格就可以得到各个控制信号的产生。为了方便代码的阅读，我们先定义一些常量来表示各个指令，如果是R型指令，则用func部分定义，如果不是R型指令，则用op部分定义。

```
parameter ADD = 6'b100000;
parameter SUB = 6'b100010;
parameter AND = 6'b100100;
parameter OR = 6'b100101;
parameter SLL = 6'b000000;
parameter ADDIU = 6'b001001;
parameter ANDI = 6'b001100;
parameter ORI = 6'b001101;
parameter SLTI = 6'b001010;
parameter SW = 6'b101011;
parameter LW = 6'b100011;
parameter BEQ = 6'b000100;
parameter BNE = 6'b000101;
parameter BLTZ = 6'b000001;
parameter J = 6'b000010;
parameter HALT = 6'b111111;
```

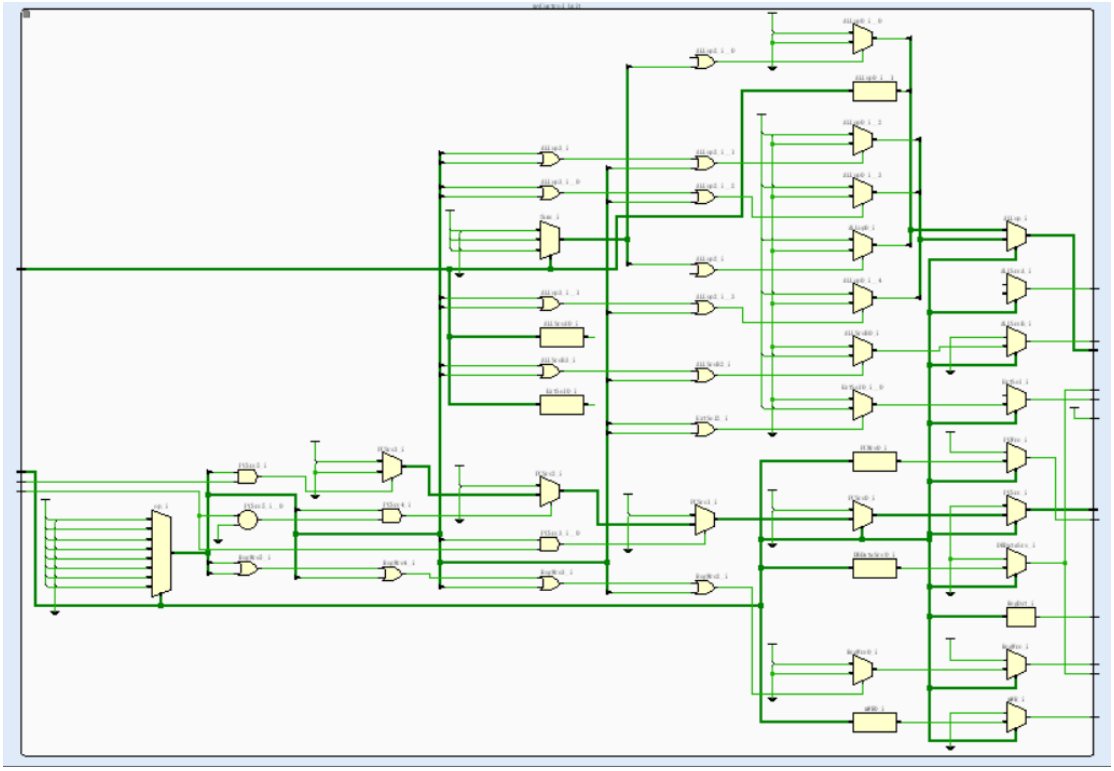
▲常量的定义

注意!!! 在译码之前，需要给一部分的信号先赋初始值，这些信号包括InsMemRw, PCWre,RegDst和ALUop.这一部分是必须的!! 因为在最开始没有读取指令的时候，也是没有指令可以来被译码的。所以一定要给这些控制信号先赋一个初始值，才能正确读取指令并向下进行。

```
initial begin
    InsMemRw = 1;
    PCWre = 1;
    RegDst = 1;
    ALUop = 3'b000;
end
```

▲为部分控制信号赋初始值

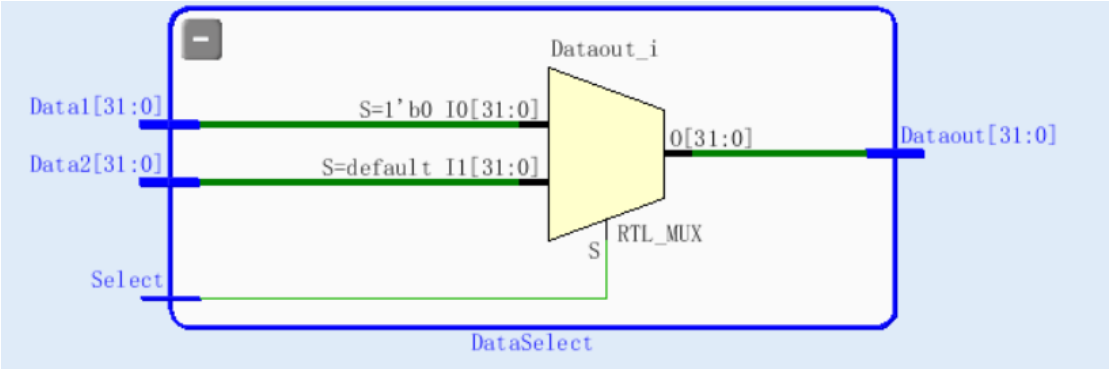
接下来只要根据之前列出的控制信号的真值表即可得到各个控制信号的取值。注意要分为R型指令和非R型指令两个部分进行讨论。



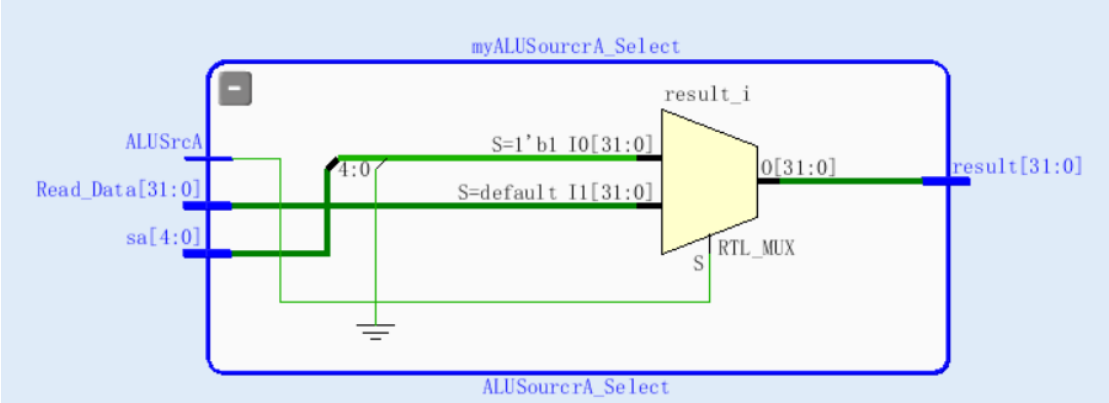
▲控制单元的内部框图

（10）、数据选择

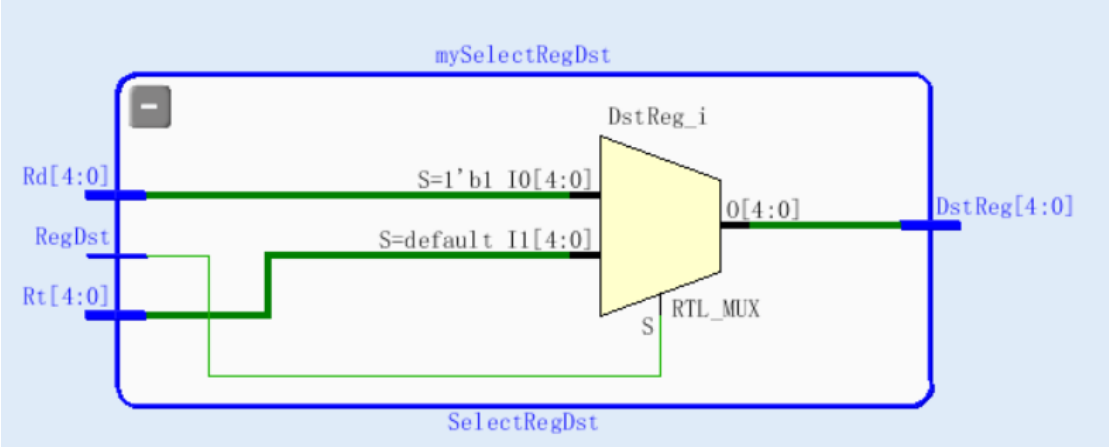
在CPU程序设计的过程中，涉及到了四个部分的数据选择。分别是ALUSrcA, ALUSrcB，RegDstSelect 和 DBDataSrcSelect四个部分。其中，ALUSrcB和DBDataSrcSelect是一个32位的二选一数据选择器。在这一部分的实现我们只需要实现一个32位二选一数据选择器的代码，之后实例化出两个实例即可。ALUSrcA是在移位量和寄存器中的数值进行选择。这里我们将移位量直接进行无符号的扩展，扩展成32位，然后在两者之间进行选择。而RegDstSelect是一个五位二选一数据选择器，对目的寄存器进行选择。数据选择的部分的代码都比较简单，只需要根据控制单元产生的控制信号对连线进行对应的连接就可以了。



▲32位二选一数据选择器内部框图

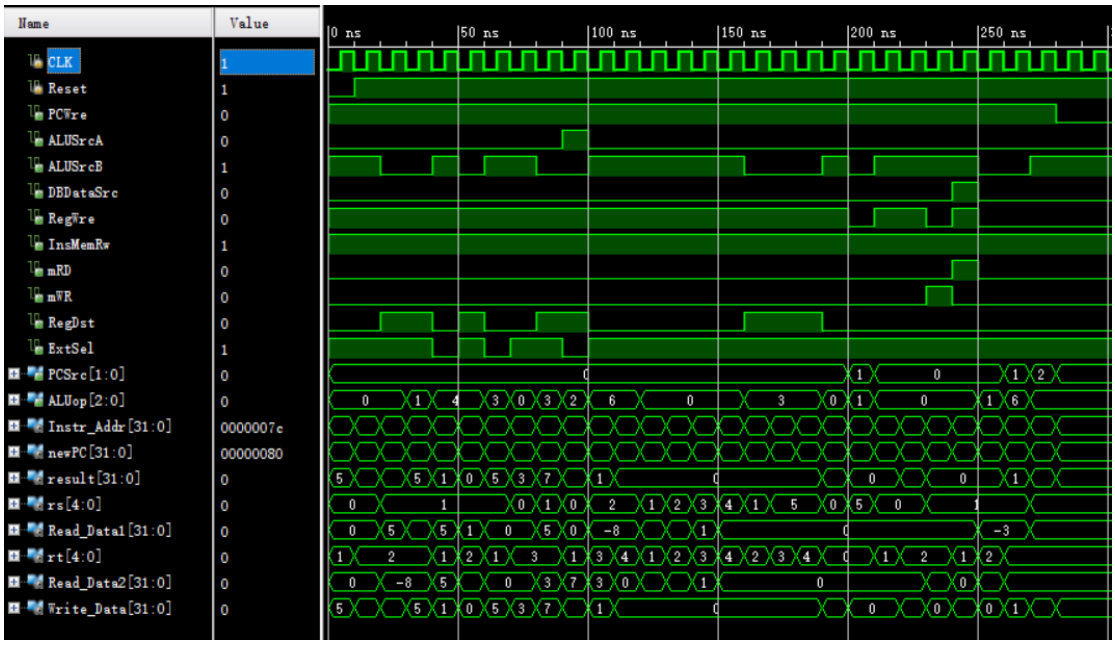


▲ALUSrcA数据选择器内部框图

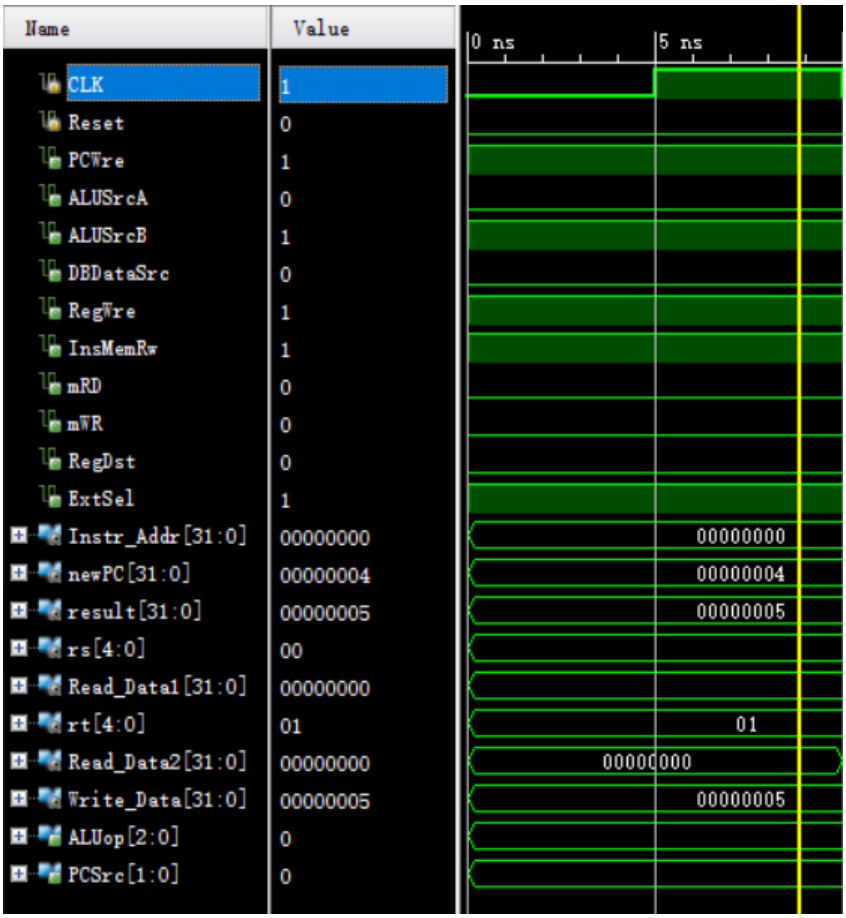


▲RegDstSelect内部框图

（二）、各条指令的波形图与信号解析



1、addiu指令



第一条指令为addiu指令。具体为：addiu $1,$0,5

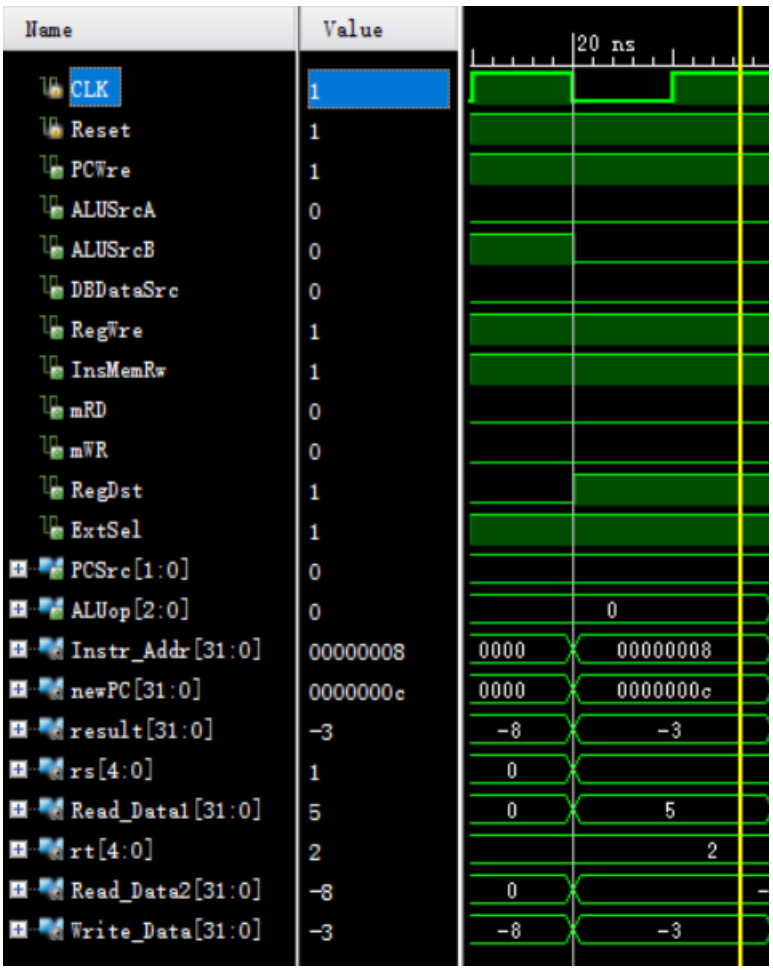
这一条指令为I型指令，即其中一个操作数为立即数。由于不是逻辑运算，所以要进行符号扩展。，因为结果要写入寄存器中，所以寄存器写使能信号为1，数据通路的内容为ALU运算结果；又由于下一条指令的地址需要写入PC中，所以PC的写使能应该为1；因为是要向指令存储器读入指令的，所以InsMemRw内容应该为1。不需要进行访问内存操作，所以数据存储器的写使能和读使能都是0。需要注意一下的是，这里ExtSel取值为1，因为这里做的是符号扩展。之前一直都有一个误区就是觉得addiu中u是代表unsigned的意思，但是去翻找MIPS的指令手册却发现手册上给出的是符号扩展，所以这次深入地了解了一下，在这里u并不是unsigned的意思，而是是否带有“自陷”或者溢出检测，而不是有无符号的加法。

仿真文件中时钟周期为10ns，所以时钟每隔5ns翻转一次。在最开始是Reset并没有置一，而是先置零，在一个时钟周期后才置一，因为程序的完成是规定只有Reset下降沿时刻才会PC清零，所以这里并不会对程序的运行造成影响。根据上述列出的表格，可以对照各个控制信号确定输出正确。

此条指令执行的操作是将5和零号寄存器即零相加的结果保存在一号寄存器中。所以rs

为0，rt为5. 由于零号寄存器中的值永远为零，所以Read_Data1为0，rt为1，此时还没有向其中写入数据，所以此时Read_Data2中的数据为0.指令没有经过跳转，此时顺序执行。下一条指令地址为00000004.指令进行的运算为0+5 = 5，所以result中保存的数值为5.

2、add指令



这一条指令为add指令。具体指令内容为：add $1,$1,$2

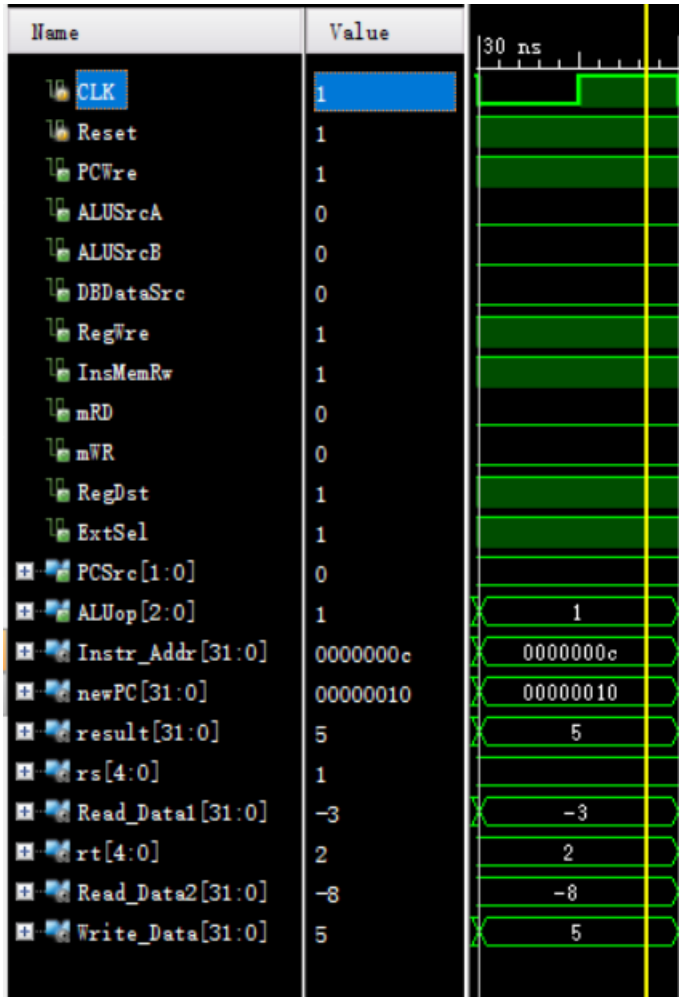
这是一条R型指令。由于两个操作数都来自寄存器，最终结果也要写入寄存器，所以寄存器写使能为1；由于最终的结果也要写入寄存器中，所以寄存器写使能信号也为1；这一条指令是不需要进行立即数扩展的，所以其实ExtSel无论是取1还是0都是可以的，为了使得程序的精简，我们令其取1；由于程序不需要对数据存储器进行读取和写入，所以数据存储器的读写控制信号都为0。因为是加法指令，所以ALU所做的运算为加法。

根据上述列出的表格，可以对照各个控制信号确定输出正确。

Rs为一号寄存器，里面存储的值为5，所以Read_Data1的值为5。Rt为二号寄存器，由于前面的操作，可以知道二号寄存器内部的值应该为-8，根据波形图也可以得到Read_Data2的内容为-8。寄存器内的值相加的结果为-3。根据波形图可以判断结果计算正确，即result和

Write_Data的结果都为-3，结果正确。

3、sub指令

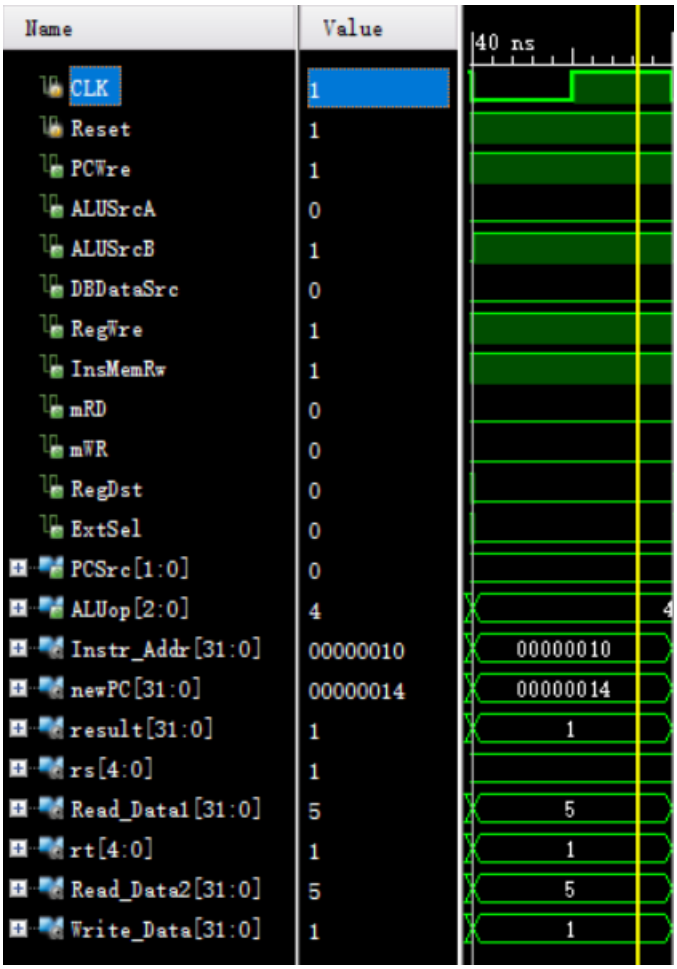


第三条指令为sub指令，具体的指令内容为：sub $1,$1,$2

同理，这也是一条R型指令，由于两个操作数都来自寄存器，最终结果也要写入寄存器，所以寄存器写使能为1；由于最终的结果也要写入寄存器中，所以寄存器写使能信号也为1；这一条指令是不需要进行立即数扩展的，所以其实ExtSel无论是取1还是0都是可以的，为了使得程序的精简，我们令其取1；由于程序不需要对数据存储器进行读取和写入，所以数据存储器的读写控制信号都为0。唯一与上一条指令不同的是这条指令ALU所作的运算是减法。

同理，控制信号的取值可以根据上面列出的表格对照为正确的取值。当前指令的地址为0000000C，由于指令时顺序执行的，所以指令地址直接加四即可，如上图所示指令地址正确。这条指令的意义是将一号寄存器和二号寄存器的值相减然后保存到一号寄存器中去。根据前面的运算，一号寄存器中的内容为-3，二号寄存器中的内容为-8. 由于-3 - （-8） = 5，将数值5存入一号寄存器中，即write_data为5.

4、andi

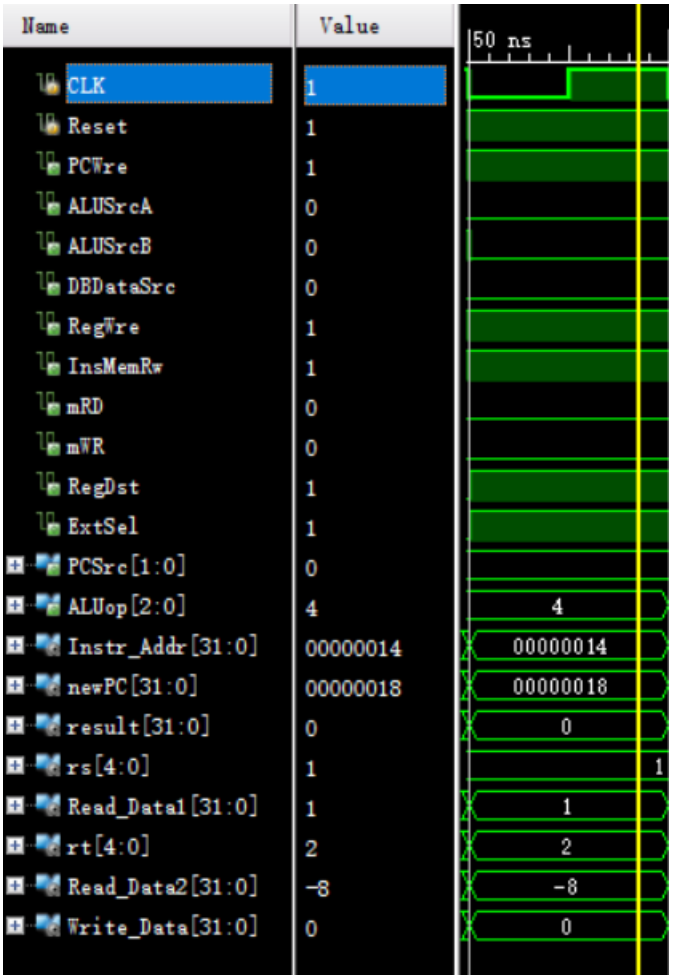


第四条指令为andi指令，具体指令内容为：andi $1,$1,9。

可知这一条指令为I型指令，即其中一个操作数为立即数，由于是逻辑运算，所以需要进行零扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；因为不需要对数据存储器进行存取，所以数据存储器的读写时能信号都为0.

这条指令的含义是将一号寄存器的内容和数字9按位相与，结果保存在一号寄存器当中。由于前面的计算，一号寄存器中的数据为5。即将1001和0101按位相与，结果应为0001，即1，所以result为1，结果正确。数据通路的内容应该为ALU运算结果，所以Write_Data内容为1，结果正确。

5、and指令

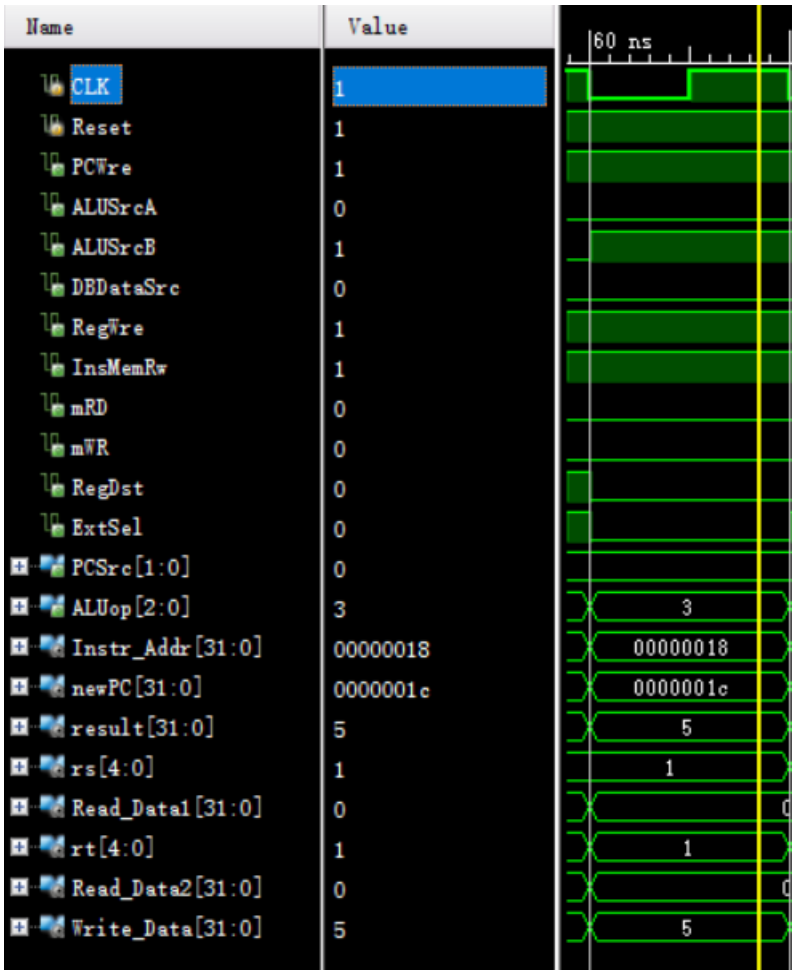


这一条指令为and指令，指令的具体内容为：and $1,$1,$2

这是一条R型指令，由于两个操作数都来自寄存器，最终结果也要写入寄存器，所以寄存器写使能为1；由于最终的结果也要写入寄存器中，所以寄存器写使能信号也为1；这一条指令是不需要进行立即数扩展的，所以其实ExtSel无论是取1还是0都是可以的，为了使得程序的精简，我们令其取1；由于程序不需要对数据存储器进行读取和写入，所以数据存储器的读写控制信号都为0。ALU所做的运算为将一号寄存器的内容和二号寄存器的内容按位相与。

由于之前的运算，寄存器1中的内容为1，寄存器2中的内容为-8.也就是将1111111111111000和0001按位相与，结果为0.即result为0，结果正确。由于指令是顺序执行的，所以下一条指令地址即为当前PC值直接加四。结果正确。

6、ori

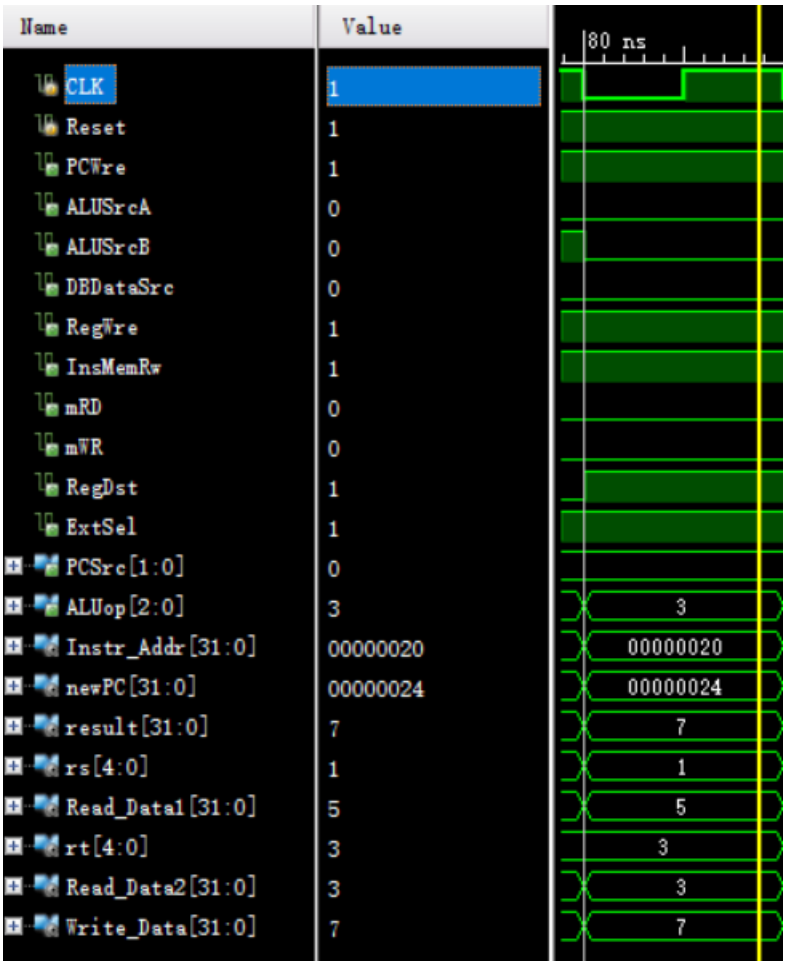


这一条指令是ori指令，具体指令内容为ori $1,$1,5

可知这一条指令为I型指令，即其中一个操作数为立即数，由于是逻辑运算，所以需要进行零扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；因为不需要对数据存储器进行存取，所以数据存储器的读写时能信号都为0.

这一条指令的含义是将一号寄存器中的内容和立即数5进行按位或，然后结果保存到一号寄存器当中。由于之前的计算，一号寄存器中此时保存的数据内容为0，所以本条指令就是将0101和0000按位或，即结果为0101，用十进制表示为5。结果写到一号寄存器中。

7、or

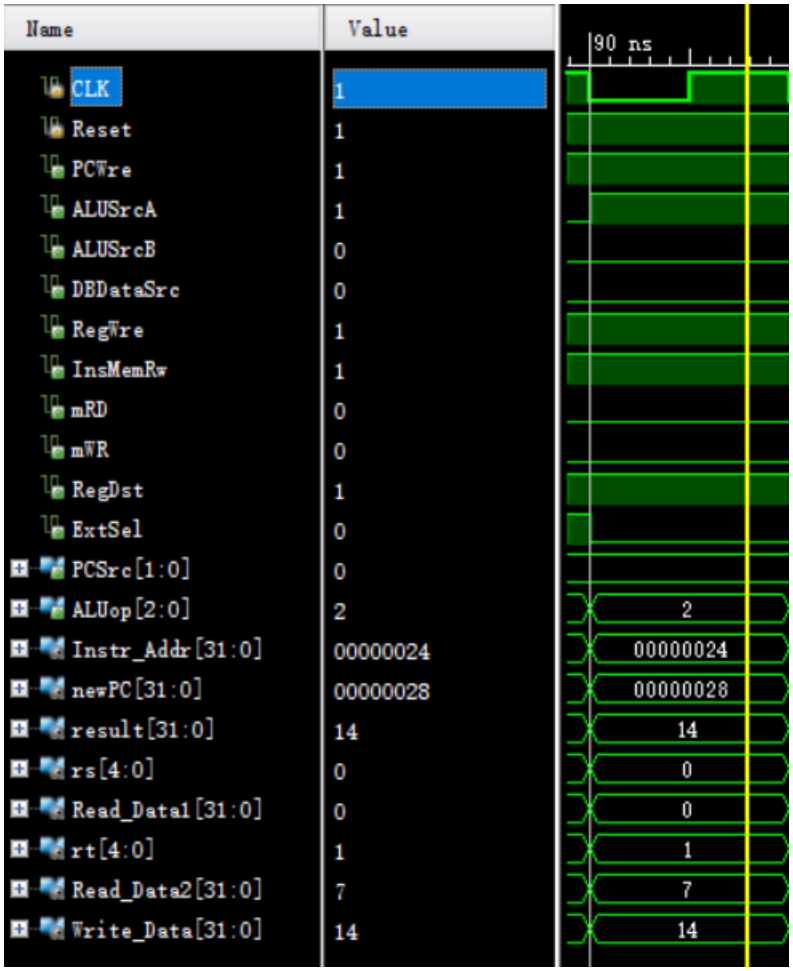


本条指令为or指令，指令的具体内容为or $1,$2,$3

这是一条R型指令，由于两个操作数都来自寄存器，最终结果也要写入寄存器，所以寄存器写使能为1；由于最终的结果也要写入寄存器中，所以寄存器写使能信号也为1；这一条指令是不需要进行立即数扩展的，所以其实ExtSel无论是取1还是0都是可以的，为了使得程序的精简，我们令其取1；由于程序不需要对数据存储器进行读取和写入，所以数据存储器的读写控制信号都为0。ALU所做的运算为将一号寄存器的内容和三号寄存器的内容按位相或。

这一条指令和上一条指令是很相似的，区别在于上一条指令对立即数进行操作，而本条指令没有。由于前面指令的计算，一号寄存器中的内容为5，三号寄存器内容为3，即将0101和0011按位或，结果为0111，即7.所以result结果为7，结果正确。

8、sll

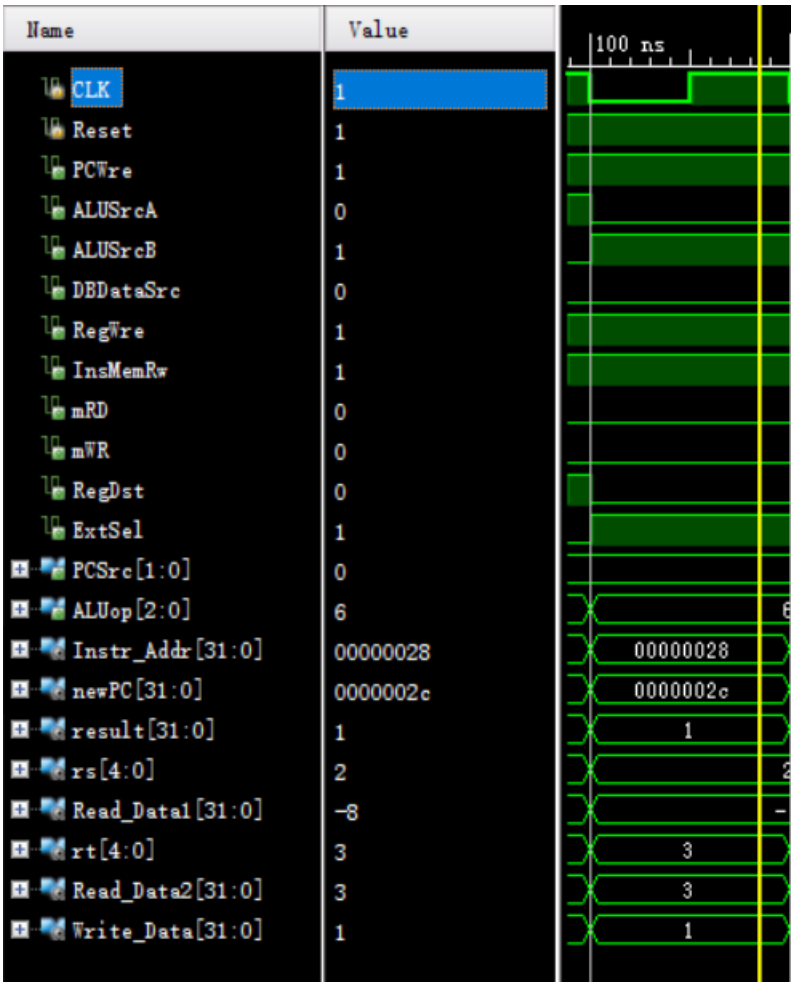


本条指令为sll指令，具体指令内容为sll $1,$1,1。

这条指令的含义是移位指令。操作数分别是一号寄存器中的内容和指令中sa段的内容。波形图中显示出了rs寄存器的编号和其中的内容，但其实我们并不需要这个值，因为真正的操作数并不是他。在指令的编写中，我们直接将sa段进行零扩展而没有经过扩展器，所以在这里ExtSel的取值是不重要的，但是为了程序的规整器件还是将其赋为0.由于ALU操作数分别来自于寄存器和Sa段的数值，所以ALUSrcA, ALUSrcB的值分别为1和0. 最终数据通路的内容为ALU运算的结果，所以DBDataSrc的取值应当为0.不需要对存储器进行读写操作，所以存储器的写使能和读使能信号都是0.因为结果最终要写回寄存器堆中，所以寄存器写使能应该为1。

这条指令的含义是将一号寄存器中的内容向左移一位。由于之前的运算，一号寄存器中的数据为7，对于二进制来说，移位就相当于扩大一倍，所以结果为14。由于指令时顺序执行的，所以下一条指令的地址即为当前指令地址直接加四。

9、slti

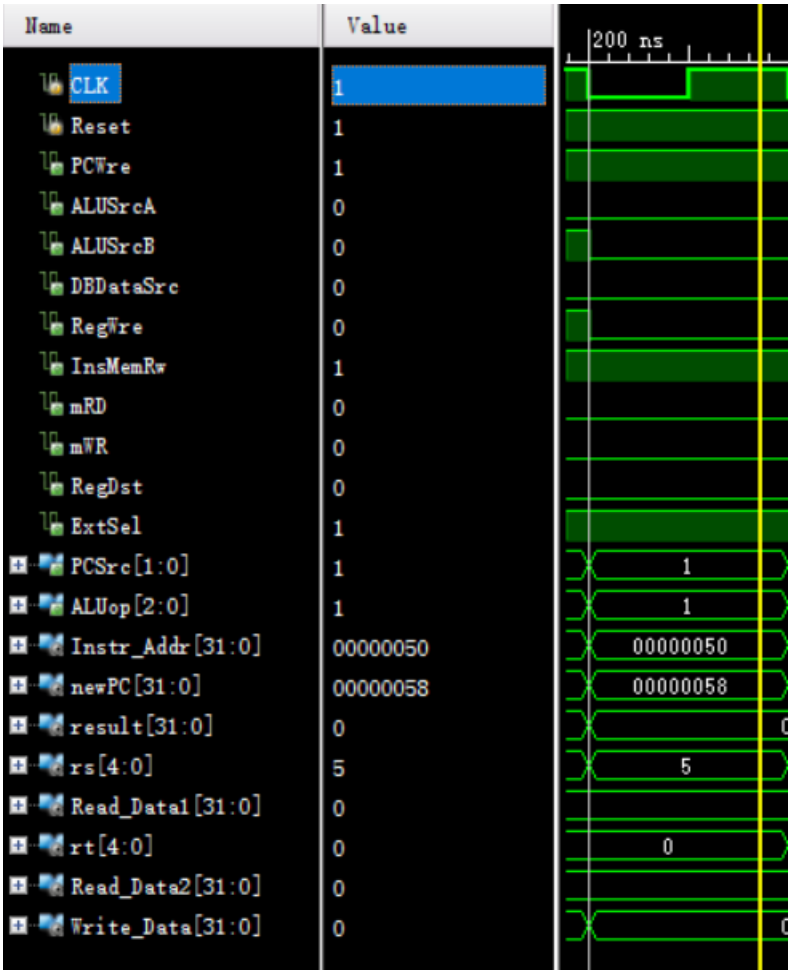


本条指令是slti，具体指令内容为slti $3,$2,-7

可知这一条指令为I型指令，即其中一个操作数为立即数，由于进行的是有符号的比较，所以应该进行符号扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；因为不需要对数据存储器进行存取，所以数据存储器的读写时能信号都为0.

这条指令的含义是将二号寄存器中的内容和立即数-7进行有符号的比较。因为二号寄存器中的内容是-8，-8< -7，所以比较结果应该置一。所以result的内容为1.由于指令是顺序执行的，所以下一条指令的地址为当前PC值加四

10、beq



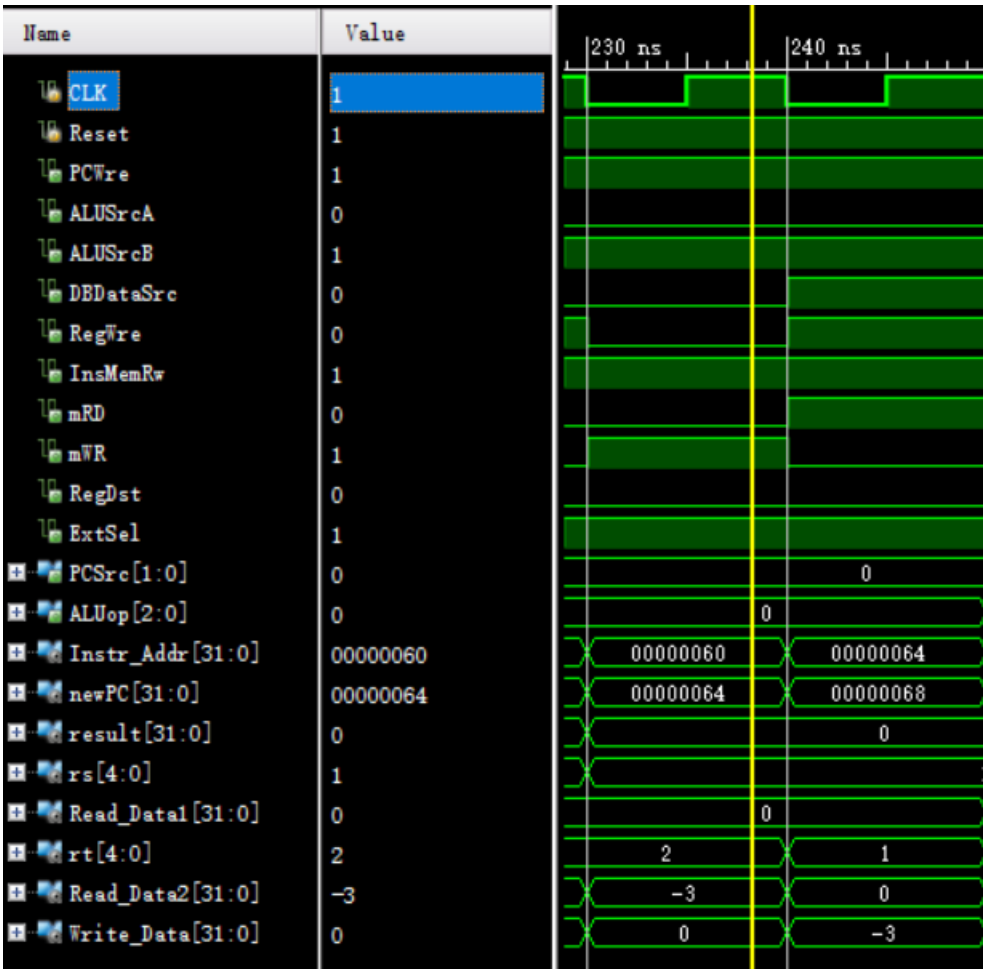
这一条指令是beq指令，指令的具体内容为beq $5,$0,1

可知这一条指令为I型指令，即其中一个操作数为立即数，由于进行的是有符号的比较，所以应该进行符号扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；因为不需要对数据存储器进行存取，所以数据存储器的读写时能信号都为0.

这条指令的含义是比较五号寄存器和零号寄存器中的内容，如果相等就跳转，在CPU内真正的运算过程是将五号寄存器中的内容和零号寄存器中的内容做差，然后将结果和零比较。所以ALU所做的运算为减法运算。

由于五号寄存器中的数据为0，零号寄存器中的数据也永远为0，所以两者相等，程序将进行跳转。跳转地址为PC + 4 + 4.所以下一条指令的地址为当前地址加8，即PC由00000050变为00000058.

11、sw



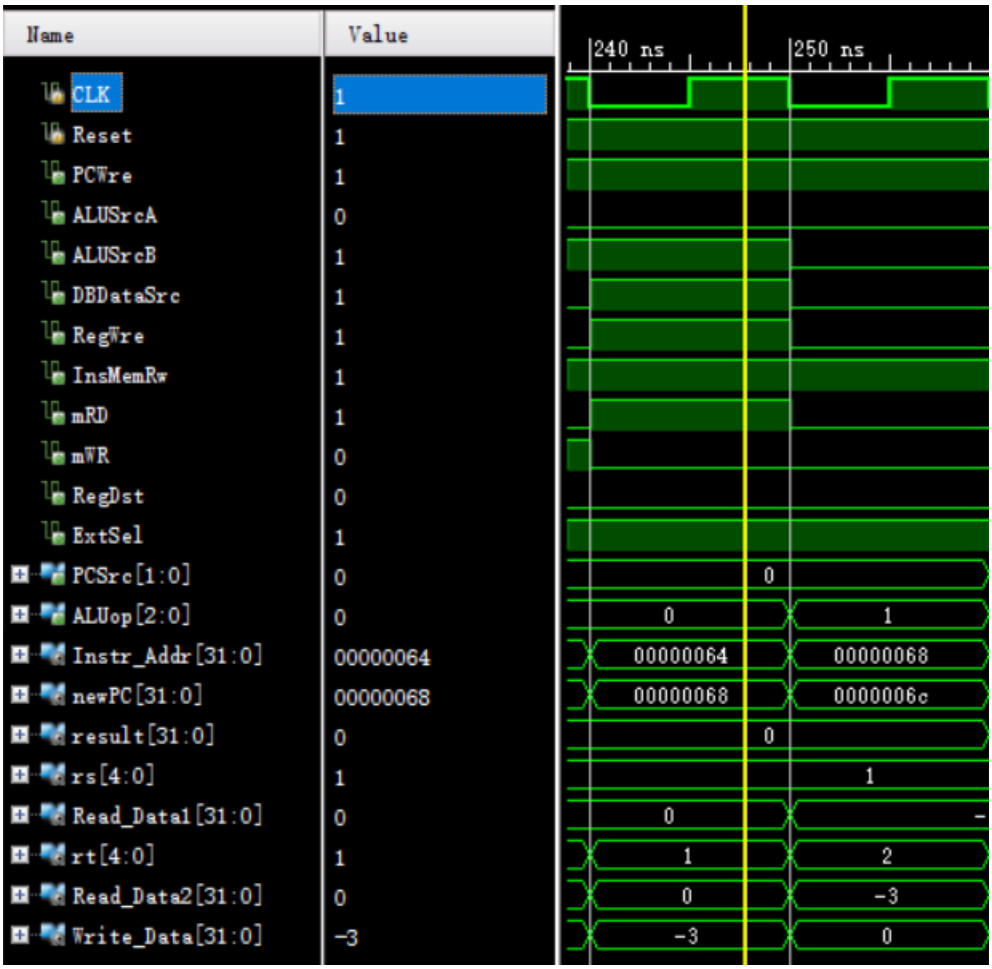
这条指令为sw指令，指令的具体内容为sw $2,0($1)

可知这一条指令为I型指令，即其中一个操作数为立即数，由于进行的是有符号的比较，所以应该进行符号扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；由于S指令需要向数据存储器中写入数据，所以数据存储器的写使能信号为1而读使能信号为0；

这条指令的含义是，一号寄存器中的数据表示地址，该地址作为基址偏移0个单位得到数据存储器的地址，然后将二号寄存器中的内容新进存储器中。所以ALU所作的运算为加法。由于前面的运算，一号寄存器中的内容为0，二号寄存器中的内容为-3.所以这条指令就是将-3写入数据存储器的零地址处。

由于指令时顺序执行的，所以指令直接自增。

12、lw



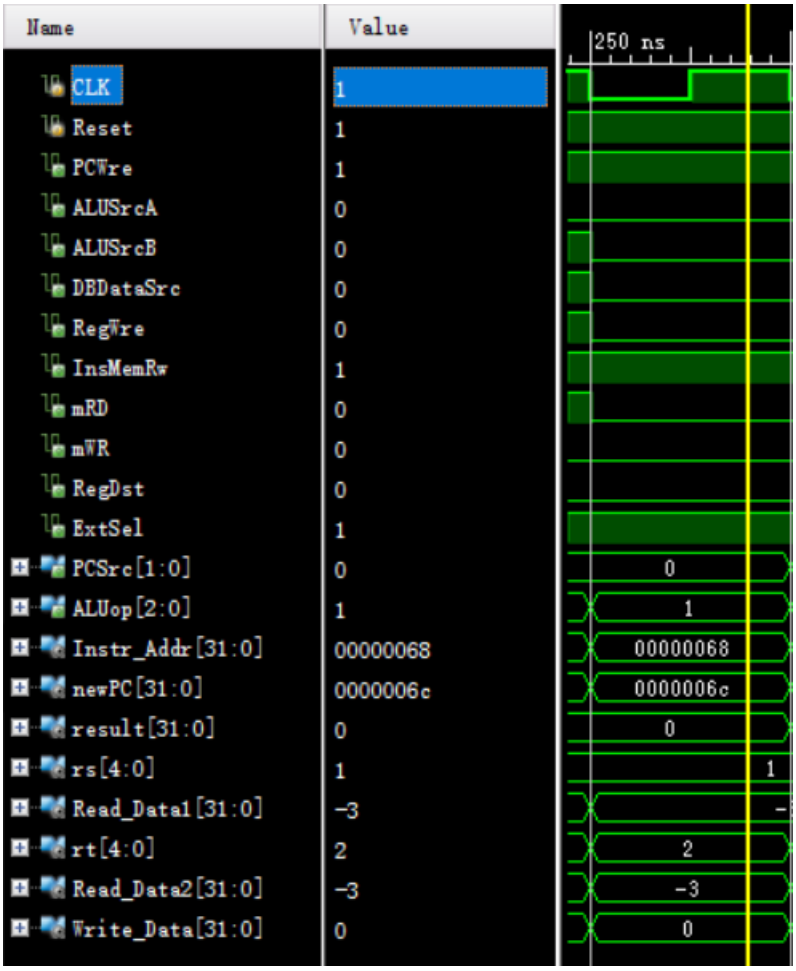
这条指令是lw指令，指令的具体内容是lw $1,0($1)

可知这一条指令为I型指令，即其中一个操作数为立即数，由于进行的是有符号的比较，所以应该进行符号扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；由于S指令需要向数据存储器中读取数据，所以数据存储器的写使能信号为0而读使能信号为1；

这条指令的含义是，一号寄存器中的数据表示地址，该地址作为基址偏移0个单位得到数据存储器的地址，然后将数据存储器再该地址处的内容加载到一号寄存器中。所以ALU所作的运算为加法。由于前面的运算，一号寄存器中的内容为0，而存储器中位与该地址处的数据为-3.所以这条指令就是将-3写入一号寄存器中。

由于指令时顺序执行的，所以指令直接自增。

13、bne

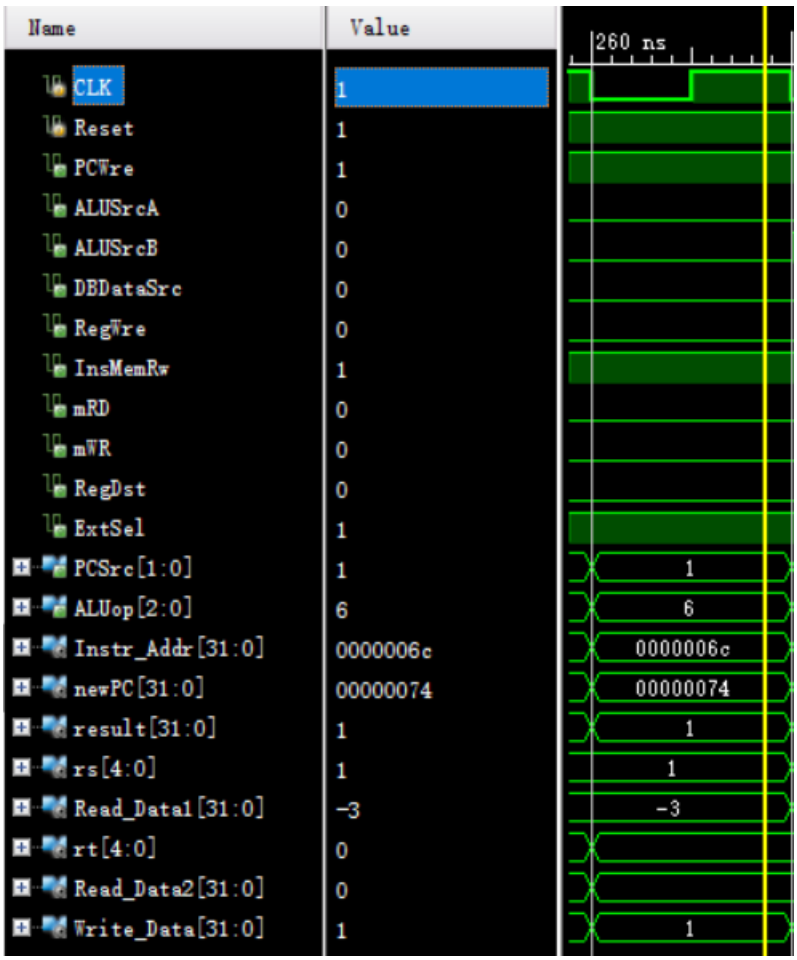


这条指令是bne指令，指令的具体内容为bne $1,$2,5

这条指令的含义是将一号寄存器中的内容和二号寄存器中的内容比较，如果不相等就跳转。根据前面的操作我们可以知道一号寄存器和二号寄存器中的内容全部为-3，所以两个寄存器中的内容相等，程序会发生跳转。PC = PC + 4 + 4 * 5；

可知这一条指令为I型指令，即其中一个操作数为立即数，由于跳转指令可以向前跳也可以像后跳，所以应该进行有符号扩展，所以应该进行符号扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；因为不需要对数据存储器进行存取，所以数据存储器的读写时能信号都为0.

14、bltz

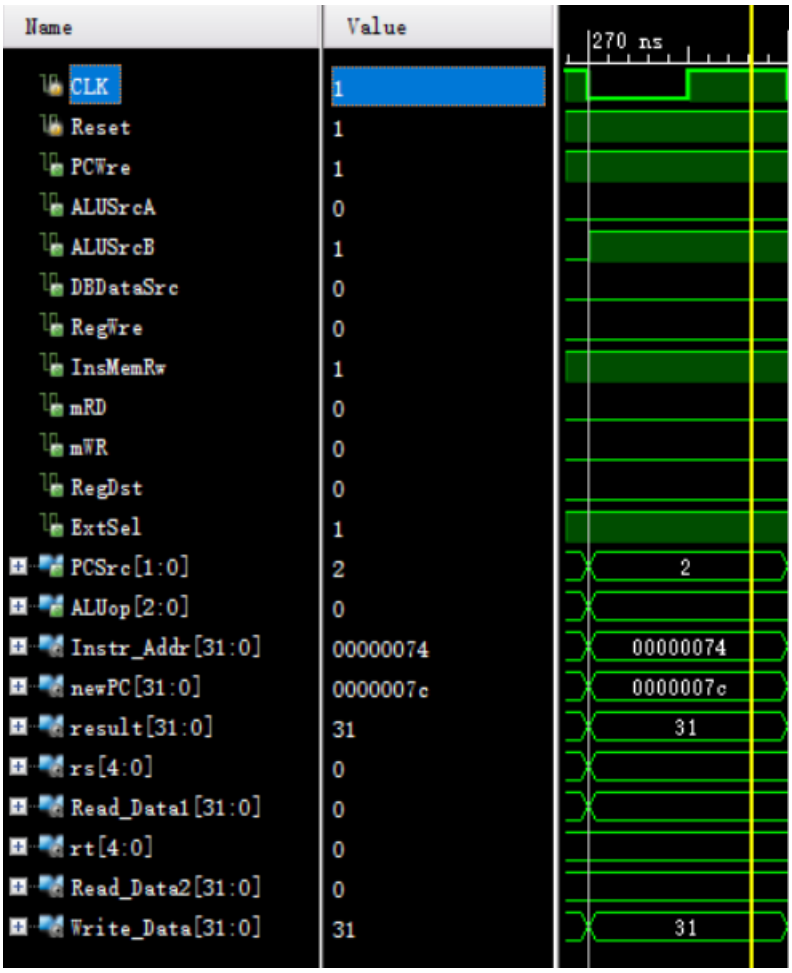


这条指令是bltz指令，指令的具体内容是bltz $1,1

这一条指令的含义是如果一号寄存器中的内容小于0就进行跳转。由于前面的操作一号寄存器中的内容为-3，小于零，所以需要进行跳转。PC = PC + 4 + 4*1；

可知这一条指令为I型指令，即其中一个操作数为立即数，由于跳转指令可以向前跳也可以像后跳，所以应该进行有符号扩展，所以应该进行符号扩展。ALU的操作数一条来自于寄存器，另一个为立即数，所以ALUSrcA、ALUSrcB分别为0和1；最终的数据通路来自ALU运算结果，所以DBDataSrc为0；由于运算结果最终是要写入一号寄存器中的，所以寄存器写使能信号为1；因为不需要对数据存储器进行存取，所以数据存储器的读写时能信号都为0.

15、j

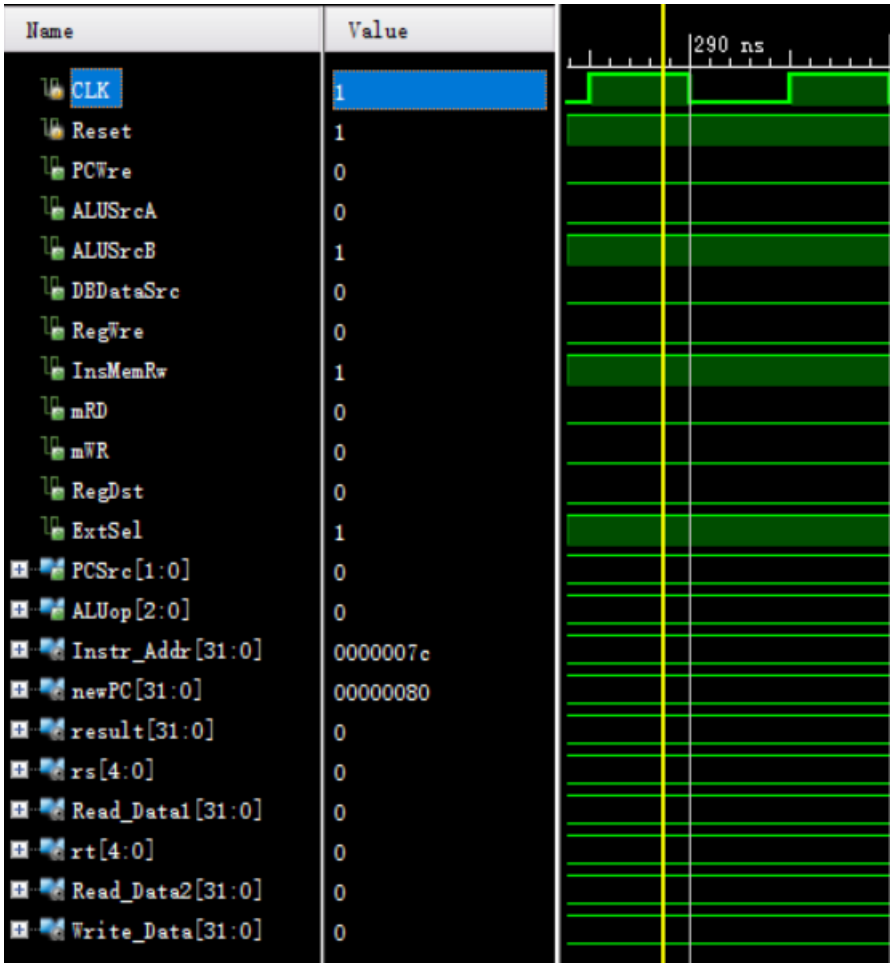


这条指令是无条件跳转指令，指令的具体内容为 j 0x7C

这条指令的含义是直接跳转到指令中给出的地址。由于要对 PC 做出更改，所以 PC 的写使能为 1.因为不需要对寄存器内容进行更改，所以寄存器写使能信号为 0.同样的不需要对存储器进行读写操作，所以存储器的读写使能信号都为 0.由于需要对指令存储器进行写入指令所以 InsMemRw 取值那个应该为 1.对于其他的控制信号，都是取一和取零都可以的，但是为了程序的规整性，我们还是为其赋予了一定的数值。

由于是无条件跳转指令，所以 PC 值直接更改为 0x7C。

16、halt



Halt为停机指令，指令经过译码所有的控制信号全部为0，PC也不会再更改，程序结束。

（三）、Basys3实验板

本实验的文件组织如下所示

▲项目的文件组织

也就是说在烧板的部分，还单独添加了一个数码管的译码文件和按键消抖的文件，同时在顶层文件中将这几个模块实例化。

那么在烧板的过程中我们需要解决一些问题，其中包括：

（1）、如何实现数码管的位选

（2）、如何实现拨动开关控制数据的显示

（3）、如何实现按键的消抖

（4）、各个模块之间是如何连接的

我们接下来将分别讨论这四个问题的解决方法。

1 、如何实现数码管的位选

数码管的显示我们仍然采用上个学期所学习过的扫描式显示。这就需要对时钟进行分频操作，防止时钟频率过高，数码管不能正确显示。同时，由于数码管扫描式显示利用的人眼的视觉暂留效果，如果时钟频率过低仍然不能达到同时显示的视觉效果，所以时钟分频要选取一个是适当的分频时间。

之前的分频操作都是直接用封装好的分频器 IP，这次自己实现其实原理也很简单。设置一个常量表示 CLK_DIV 表示分频后得到的每个时钟周期的内部时钟周期数，这里我们选取的数值是 10000。对于位选部分具体的实现，就是每次一个机器时钟的下降沿到来，计数变量就自增，当计数变量和 CLK_DIV 相等时，就是到了我们理想的分频周期，此时数码管的位选改变。对于数码管位选的改变，只需要一个 case 语句，使得数码管的位选信号顺序变化即可，事实上位选是不需要顺序变化的，只要是每一位都会平均的取到就可以了，不过我们又何必自找麻烦呢～

```
//这一部分是位选
always@(negedge CLK) begin
    if(Reset==1) begin//如果不是清零信号
        cnt <= cnt + 1;//来一个CLK记一次数
        if(cnt==CLK_DIV) begin
            cnt <= 0; //当来到了CLK_DIV这么多的始终的时候，
            case(AN)
                4'b0111: AN <= 4'b1011;
                4'b1011: AN <= 4'b1101;
                4'b1101: AN <= 4'b1110;
                4'b1110: AN <= 4'b0111;
                default: AN <= 4'b1111;
            endcase
        end
    end
end
```

▲位选部分代码

2 、如何实现拨动开关控制数据的显示

我们使用两个拨动开关来控制四组数据的显示，所以表示拨动开关的变量要设置成两位。在拨动开关控制数据显示这一部分，要注意需要两层 case 语句嵌套，这两层 case 语句分别控制位选和数据。即在位选选到第 K 位时根据波动开关的取值选取第 K 位应该输出的

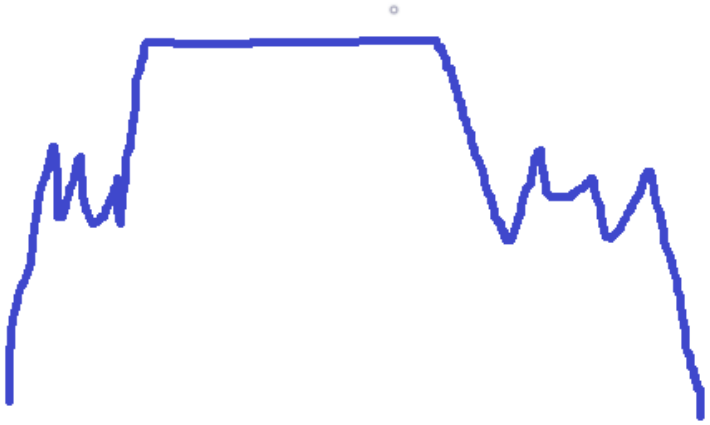
数据。这一部分拨动开关的显示不是独立自主确定的，是要结合位选所选通的位置的。这里是位选 case 在外层还是拨动开关 case 在外层是没有关系的，我们的程序选择了位选 case 在外层

```
always@( key_en ) begin
    if(Reset==1) begin
        case(AN)
            4'b0111:begin
            case(SW_in)
                2'b00: display_data = Instr_Addr[7:4];
                2'b01: display_data = {3'b000,rs[4]};
                2'b10: display_data = {3'b000,rt[4]};
                2'b11: display_data = result[7:4];
            endcase
            end
            4'b1011:begin
            case(SW_in)
                2'b00: display_data = Instr_Addr[3:0];
                2'b01: display_data = rs[3:0];
                2'b10: display_data = rt[3:0];
                2'b11: display_data = result[3:0];
            endcase
            end
            4'b1101:begin
            case(SW_in)
                2'b00: display_data = newPC[7:4];
                2'b01: display_data = Read_Data1[7:4];
                2'b10: display_data = Read_Data2[7:4];
                2'b11: display_data = result[7:4];
            endcase
            end
            4'b1110:begin
            case(SW_in)
                2'b00: display_data = newPC[3:0];
                2'b01: display_data = Read_Data1[3:0];
                2'b10: display_data = Read_Data2[3:0];
                2'b11: display_data = result[3:0];
            endcase
            end
        endcase
        end
    end
```

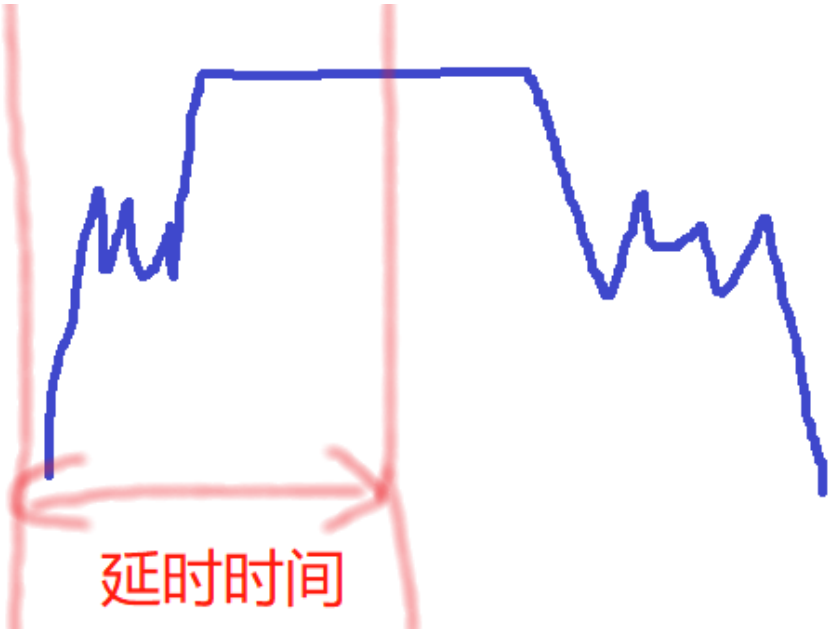
▲拨动开关控制数据输出部分代码

3、按键消抖部分

按键消抖部分其实是为了提高CPU的稳定性，经过资料的查询，按键消抖的方法是有很多的，我们这里采用最简单的一种按键消抖方法，就是延时的方式。根据Basys3实验板手册，可以知道Basys3板子的按键是高电平有效，但是按键按下时电平信号如下所示



如上所示，在按键刚刚按下时，信号是不稳定的，那么我们用延时的方法，等到信号稳定的时候再选取信号，即如下所示



和数码管的位选一样，这个延时时间不能过长也不能过短，如果过短信号还没有稳定就捕捉信号，如果过长可能捕捉不到信号。对于延时的实现思路，就是定义一个DURATION 的常量，当时钟周期走过DURATION这么多的个数时，捕捉信号。这就相当于将信号捕捉时间与按键真正按下的时间延迟了5000个时钟周期。在每个时钟到来的时候，如果计数变量和

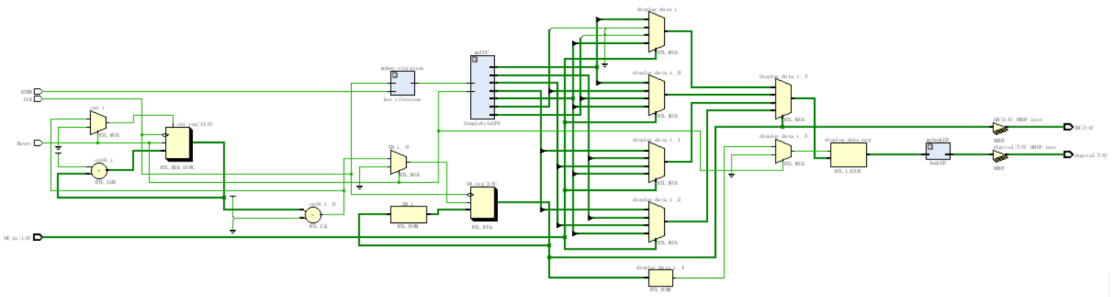
DURATION－1 相等，那么就采集键使能信号，否则不采集。具体实现代码如下：

```
always@(posedge CLK)
begin
    if(key==1) begin//表示如果按键按下去了
        if(cnt==DURATION)
            cnt <= cnt;
        else
            cnt <= cnt + 1;
    end
    else
        cnt <= 16'b0;
end

always@(posedge CLK) begin//每个时钟来的时候都判断现在是否键使能了
    if(key) key_en <= (cnt==DURATION-1'b1)?1'b1:1'b0;
end
```

▲按键消抖部分主体代码

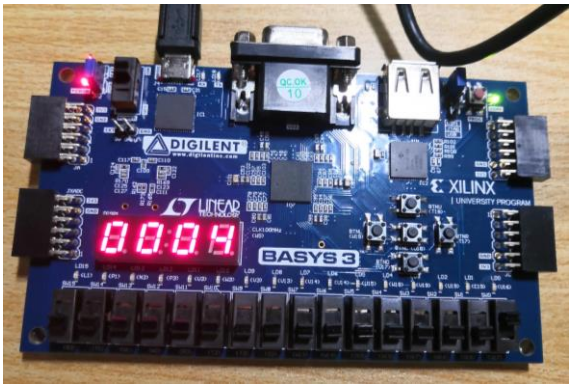
4 、各个模块之间的连接



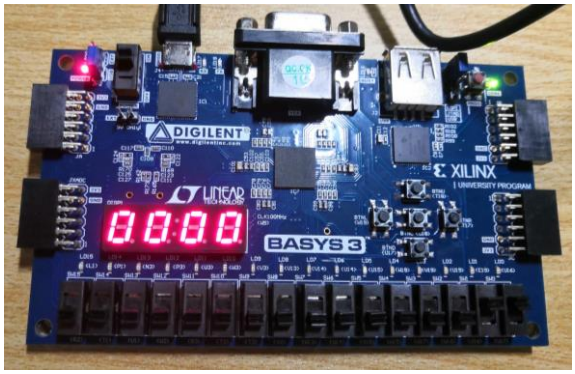
▲各个器件的连接图

在Basys3顶层模块中实例出CPU，按键消抖，数码管译码部分，由于数码管的位选部分在顶层文件中直接实现而没有再设计模块，所以接口连接中存在很多的数据选择器。对于整个CPU的设计，只包含四个输入和两个输出端口。

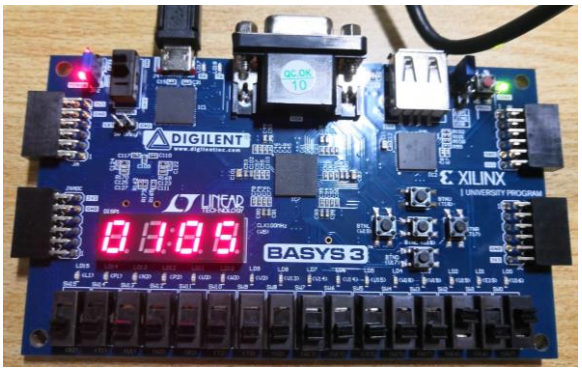
Basys3上连续的指令：



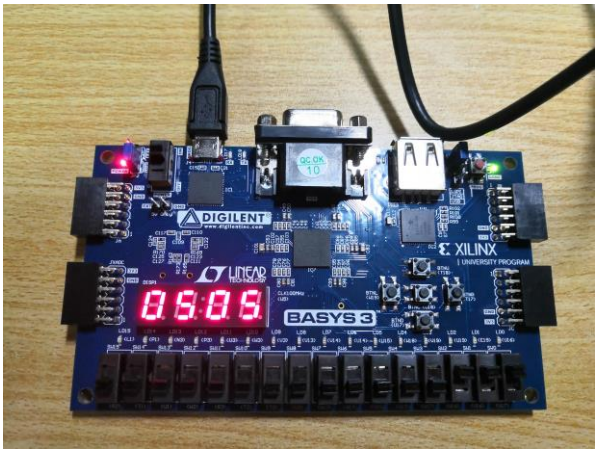
▲第一条指令addiu $1,$0,5-> 当前PC：下一条PC



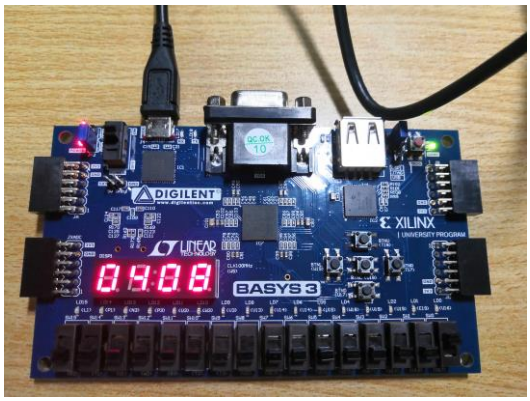
▲第一条指令addiu $1,$0,5-> rs寄存器地址：rs寄存器数据



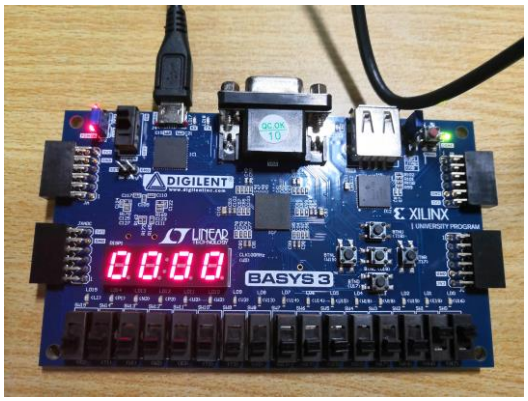
▲第一条指令addiu $1,$0,5-> rt寄存器地址：rt寄存器数据



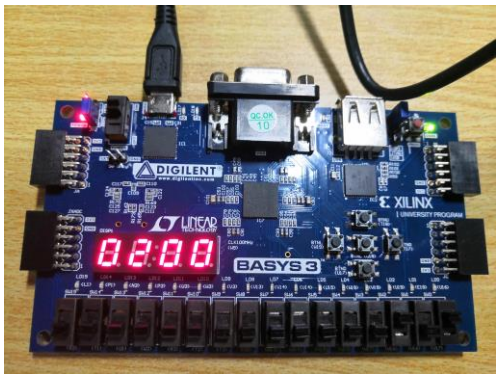
▲第一条指令addiu $1,$0,5->ALU运算结果：DB总线数据



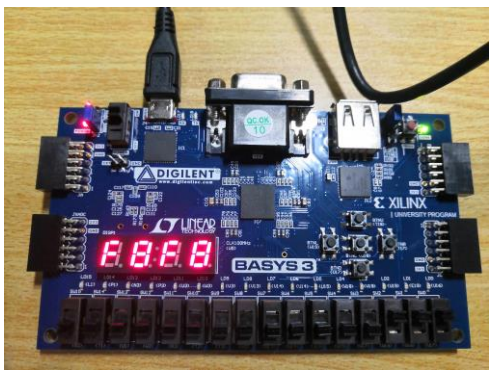
▲第二条指令addiu $2,$0,-8-> 当前PC：下一条PC



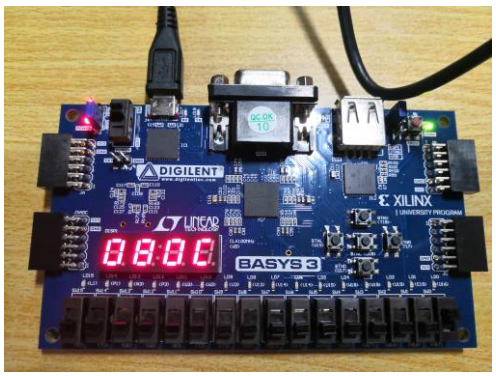
▲第二条指令addiu $2,$0,-8-> rs寄存器地址：rs寄存器数据



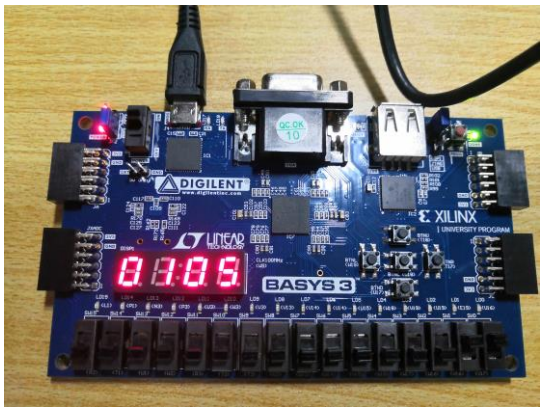
▲第二条指令addiu $2,$0,-8-> rt寄存器地址：rt寄存器数据



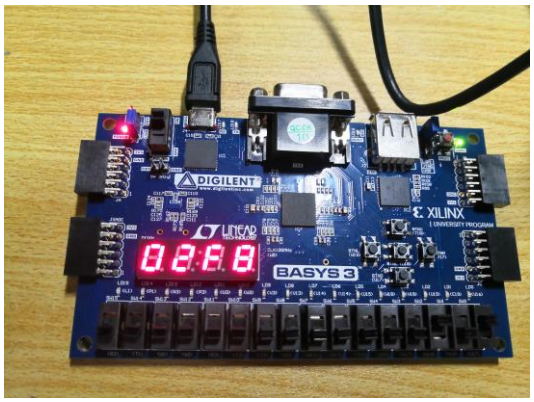
▲第二条指令addiu $2,$0,-8->ALU运算结果：DB总线数据



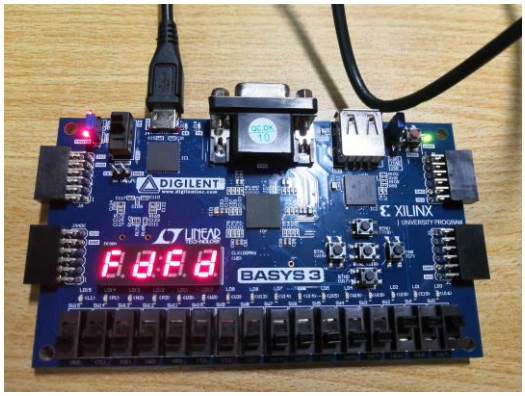
▲第三条指令add $1,$1,$2-> 当前PC：下一条PC



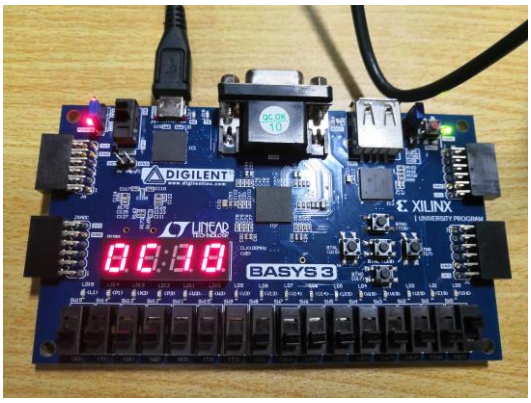
▲第三条指令add $1,$1,$2-> rs寄存器地址：rs寄存器数据



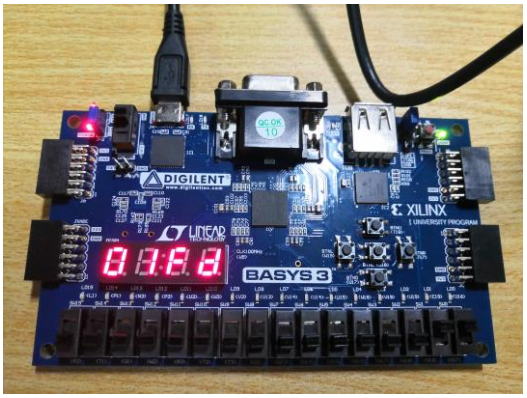
▲第三条指令add $1,$1,$2-> rt寄存器地址：rt寄存器数据



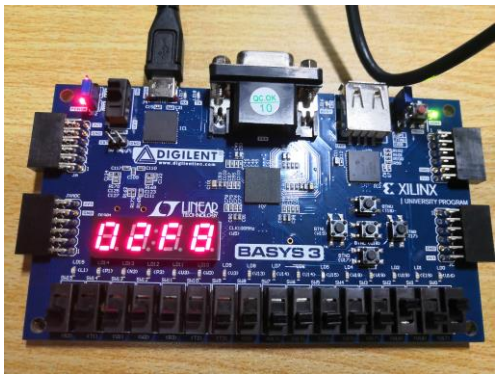
▲第三条指令add $1,$1,$2->ALU运算结果：DB总线数据



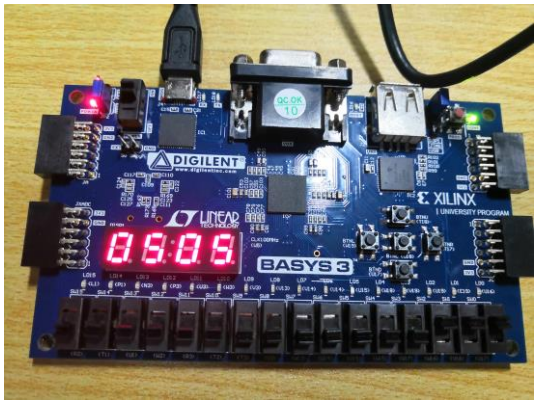
▲第四条指令sub $1,$1,$2-> 当前PC：下一条PC



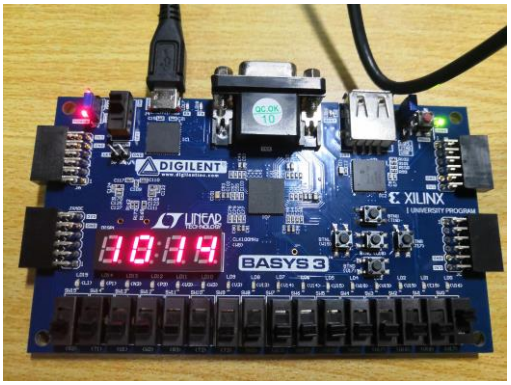
▲第四条指令sub $1,$1,$2-> rs寄存器地址：rs寄存器数据



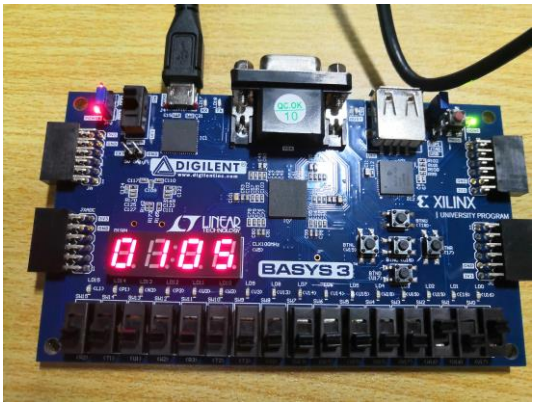
▲第四条指令sub $1,$1,$2-> rt寄存器地址：rt寄存器数据



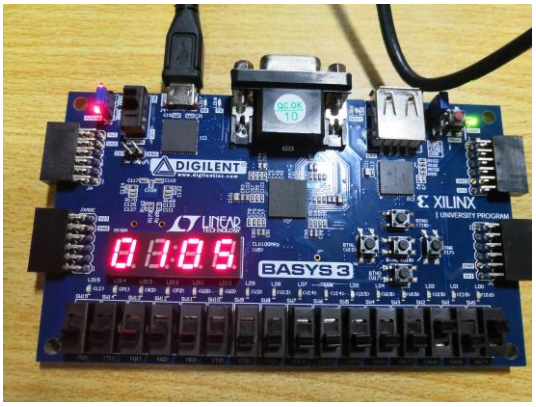
▲第四条指令sub $1,$1,$2->ALU运算结果：DB总线数据



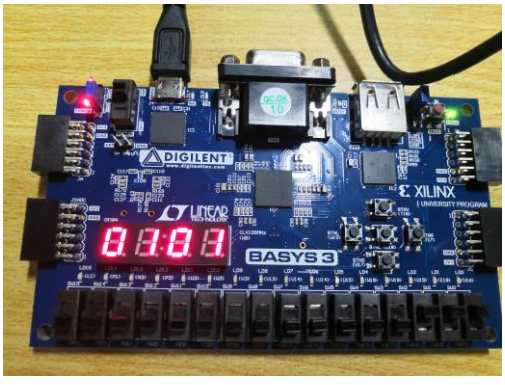
▲第四条指令andi $1,$1,9-> 当前PC：下一条PC



▲第四条指令andi $1,$1,9-> rs寄存器地址：rs寄存器数据



▲第四条指令andi $1,$1,9-> rt寄存器地址：rt寄存器数据



▲第四条指令andi $1,$1,9->ALU运算结果：DB总线数据

六. 实验心得

体会和建议。

这次因为还有其他学科的任务，所以实验完成的时间非常紧迫，不过一周写完单周期CPU也是非常非常的有成就感啊！！！！实验的过程中也遇到了很多问题，好在到后来都一一解决了，接下来就一点点来回顾一下遇见的问题吧~

1、 开始着手写代码时，遇见的第一个问题就是关于如何设置输入输出变量。开始时对于位宽的设置没有概念，也不记得要设置位宽这件事情。对于输入变量的类型和输出变量的类型也很懵懂。不过经过摸索，可以得知输入和输出的变量类型默认都是wire型的变量，如果要在always和initial语句中对其进行更改，就需要手动把它表明定义的是reg型变量。对如输入型变量是不可以定义成reg的，只能是wire。

2、 一个模块的代码中，是不可以对输入进行赋值的。也就是说assign语句不可以用在输入变量上。

3、 关于wire类型和reg类型的变量的区别：

   a) 从仿真角度来说，wire对应于连续赋值，如assign；而reg对应于过程赋值，如always和initial。

   b) 从综合的角度来说，HDL语言面对的是综合器，要从电路的角度来考虑。Wire型变量综合出来是一根导线，reg变量在always中综合出来可能是组合逻辑电路也可能是时序逻辑。

4、 关于数据存储器和指令存储器的实现：程序的数据存储器和指令存储器是需要用八位宽的寄存器来模拟的。开始的时候一直都不知道这个要怎么实现，对于这个语句的内容含义也一直都不是非常的理解。

```
Reg [7:0] Data_Memory [0:255];
```

经过学习，了解到这个语句其实就相当于定义了一个寄存器数组，然后我们把它当作内存来用，前面的[7:0]代表每个寄存器单元的数据位宽位8，后面的[0:255]表示整个寄存器数组由256个单元。在查阅资料的时候，发现有[255:0]和[0:255]两种写法，一直分不太清楚，然后按照定义位宽的习惯就写成了[255:0]，但是在仿真的时候发现数据存储并不正确，所有还是改成了[0:255]的形式。

5、 关于ALU中sign符号的设置。其实这个符号是并不必须设置的，因为我们是有带符号数比较这一运算的。那么比较完的结果就会直接保存在result里面呢，所以sign

的比较位就不应该是result的最高位，而应该是result的最低位。其实严谨的实现应该使用ALU做减法，然后用result的最高位来表示符号位的，因为在实现其他指令的时候也有可能用到sign位，如果我们简单粗暴地用result的最低位，那么这样得到的程序兼容性是不好的。

6、 关于addiu的符号扩展问题。之前一直将addiu理解成无符号的加法，而实际上并不是这样的。对16位立即数imm，在执行加法指令之前都符号扩展成32位数。ADDI和ADDIU的最大区别是有没有溢出标志（overflow），而非有符号无符号加法的区别。

7、 在第一次仿真的时候，无论是输入还是输出都是高阻态，也就是说程序根本就没有跑起来，这个令我自闭了很久，找了很久的错误也没有找到，最后发现原因是实例化的时候少打了一个字母！！！！ 乀（＇□＇）ノ 所以认真真的很重要啊啊啊！！

8、 对于控制信号初始化的部分。在开始的时候没有将控制信号初始化，所以所有的输出都是高阻态。因为在最开始的时候还没有指令读到内存中，这样的话内存中就是没有指令的，那么也就没有指令可以去译码，这样永远不会有新的指令写道PC中，也不会有数据的访问和运算，所以所有的输出都是高阻态，在最开始时要对一定的控制信号进行初始化，之后他们才能自主正确地运行下去。

9、 烧板部分：在烧板时遇到的最大的问题应该就算是分频和分配引脚的问题。在开始给时钟分频的过程中，把周期设置的太长了，导致数码管不能正确的显示，找了很久代码的错误都没有找到，之后更改过分频周期后才能正确的显示。对于引脚分配部分在最开始的时候数码管的引脚分配反了，所以显示也不正确，所以还是要认真啊啊啊啊！！！

这次单周期CPU实验就此告一段落啦~休整一下要开始准备多周期CPU了~要好好加油呀！