《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 3 班
学 生 姓 名 ：	王若琪
学　　号 ：	18340166
时　　间 ：	2019　年　11　月　27　日

成 绩 ：

实验三 ： 单周期CPU设计与实现

一、 实验目的

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

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

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期。

CPU 在处理指令时，一般需要经过以下几个步骤：

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

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

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

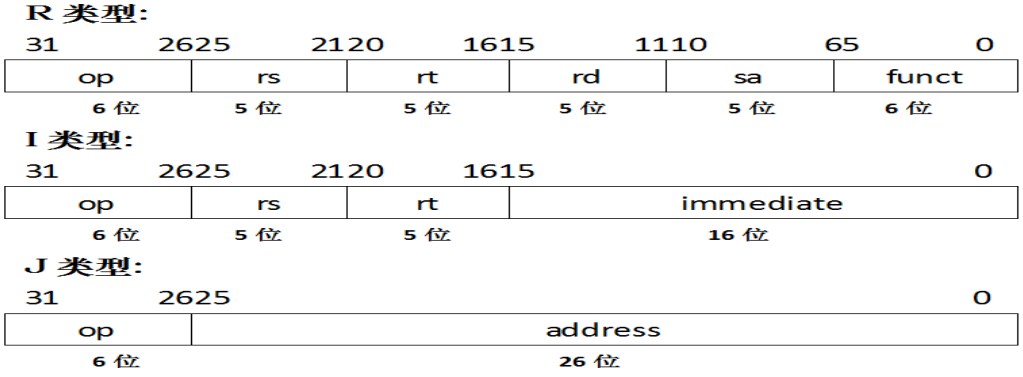
(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

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

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：



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。

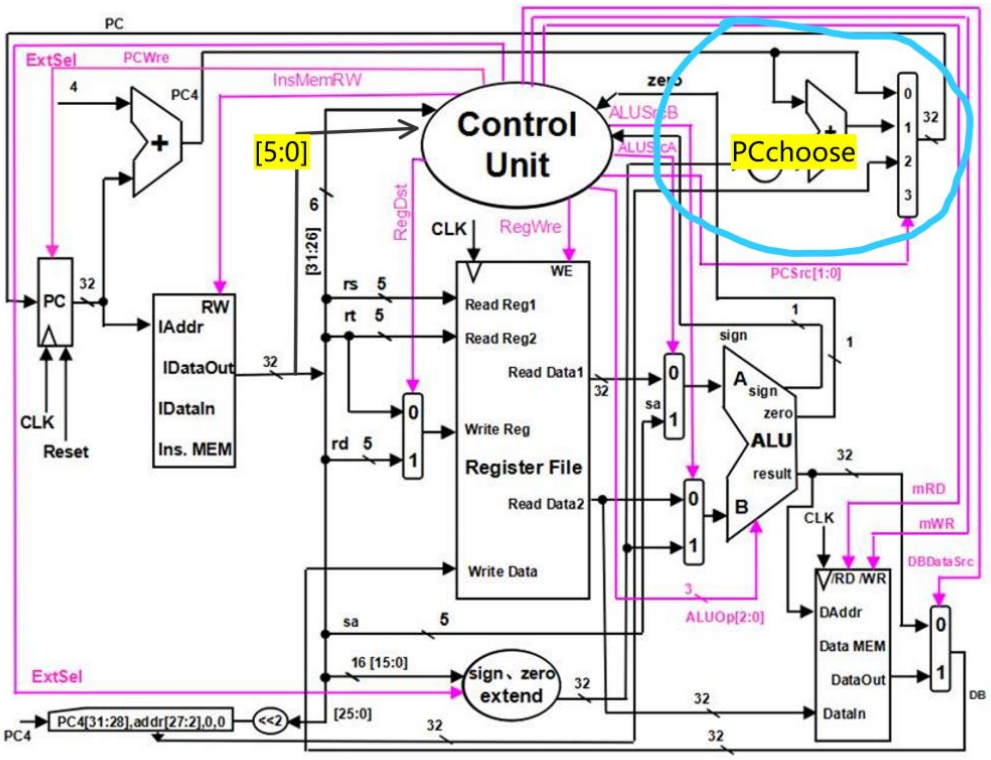


图 2　单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1 控制信号的作用

相关部件及引脚说明：

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、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)**immediate(0 扩展)**，相关指令： andi、ori	(sign-extend)**immediate (符号扩展)**，相关指令：slti、sw、lw、beq、bne、bltz、 addiu
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	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

Instruction Memory：指令存储器，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口
/RD，数据存储器读控制信号，为 0 读
/WR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口
Read Reg2，rt 寄存器地址输入端口
Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段
Write Data，写入寄存器的数据输入端口
Read Data1，rs 寄存器数据输出端口
Read Data2，rt 寄存器数据输出端口
WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU： 算术逻辑单元

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

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	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、实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五、实验过程与结果

1.确定主要模块：

首先先明确单周期CPU的整体设计思路，即为分模块设计，将CPU需要的各个模块分开设计，最后用一个顶层文件将它们连接起来。由于实验室提供的结构图有不完善或者过于

繁琐的地方，所以我并没有完全按照这个图来设计，我将我设计的CPU分为了这些模块：

（1）Control Unit 控制单元，负责根据指令和输入的zero、sign来决定输出控制其他各个单元的控制信号。

（2）ALU 算数逻辑单元，用作算数逻辑运算：根据控制信号从输入的数据中选取对应的操作数，并进行对应的运算操作并输出result、zero、sign。

（3）PC 程序计数器，用来存放当前执行指令的地址。接收PC Choose模块传进来的下一条PC并等待输出。

（4）Instruction Memory指令存储器，可以保存并输出指令。从文件读入所有的32位指令，并储存，根据输入的地址选择对应的指令输出。

（5）Data Memory 数据存储器，负责存储数据，在需要时输出数据。

（6）Register File 寄存器堆，储存程序所需要的临时数据。

（7）PC Choose 用来计算下一个指令的地址并传给PC。根据不同的PCSrc信号选择不同的跳转方式，输出正确的下一条指令PC。

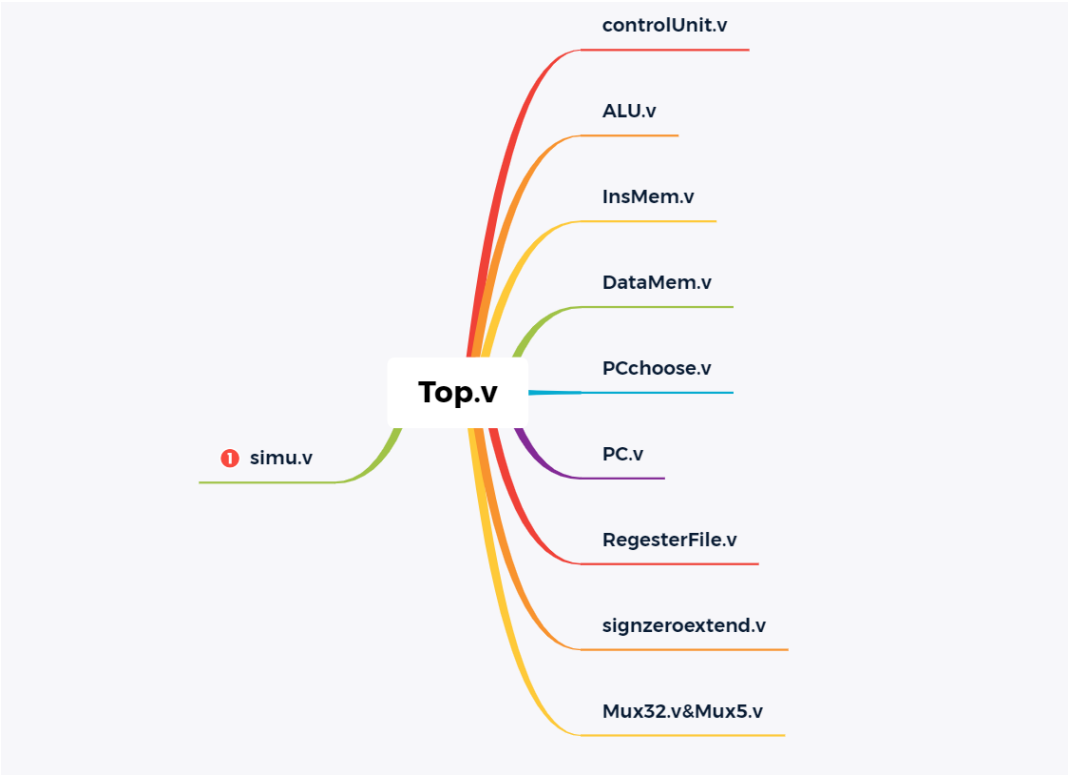
（8）Sign Zero Extension 位扩展，可以零扩展或符号扩展。

（9）Mux 多选器，用于从输入的多个数据中选一个输出。

（10）Top 顶层模块，用于将上述部分连接起来，构成真正的CPU。

（11）simu 仿真模块，激发CPU运行。

各个文件设计与关系如下图所示：



2.接着，按照老师要求，先画出了控制信号真值表

输出\输入	add	sub	addiu	andi	and	ori	or	sll	slti	sw	lw	beq	bne	bltz	j	halt
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	0	1	0	0	0	X	X
RegWre	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0
InsMemRW	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
mRD	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
mWR	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
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	1	1	1	1	1	1	1	X	X
ALUOp	000	001	000	100	100	011	011	010	110	000	000	001	001	001	X	X
PCSrc	00	00	00	00	00	00	00	00	00	00	00	00/01	01/00	00/01	10	11

这个真值表是根据要实现的指令内容来设计的，将每一条指令所对应的每一个控制信号都清楚地展示出来，是Control Unit模块实现的基础。在Control Unit模块中只需将真值表转化成对应的表达式即可。

3. Control Unit

模块说明：

Control Unit 控制单元，负责根据指令和输入的zero、sign来决定输出控制其他各个单元的控制信号。根据控制单元真值表列出表达式，达到信号控制的目的。

代码解释：

按照真值表来写出控制单元模块的代码，如下是我认为的核心部分，相当于用表达式翻译了一次真值表，当一个控制信号对应的指令0较多的时候，就用！和&&来写表达式，如RegWre的表达式，当1较多的时候，就用||来写，这样代码写起来比较方便。此外，这里有一点与实验室提供的结构图有所不同，因为我这里多加了一条输入，即为func，于是我在判断一些信号的时候，就会用到OP==Rtype&&func==and_类似的表达式来确定这条指令的目的。控制单元模块的代码的核心部分如下：

```
assign PCWre=(OP!=halt);
assign ALUSrcA=(OP==Rtype&&func==sll);
assign ALUSrcB=(OP==addiu||OP==andi||OP==ori||OP==slti||OP==sw||OP==lw);
assign DBDataSrc=(OP==lw);
assign RegWre=( OP!=sw && OP!=beq && OP!=bne && OP!=bltz && OP!=j && OP!=halt);
assign InsMemRW=1;
assign mRD=(OP==lw);
assign mWR=(OP==sw);
assign RegDst=(OP==Rtype);
assign ExtSel=(OP!=andi&&OP!=ori);
assign ALUOp[0]=(OP==Rtype&&func==sub||OP==Rtype&&func==or_||OP==ori||OP==beq||OP==bne||OP==bltz);
assign ALUOp[1]=(OP==Rtype&&func==or_||OP==ori||OP==Rtype&&func==sll||OP==slti);
assign ALUOp[2]=(OP==andi||OP==Rtype&&func==and_||OP==slti);
assign PCSrc[0]=(OP==beq&&zero==1||OP==bne&&zero==0||OP==bltz&&sign==1||OP==halt);
assign PCSrc[1]=(OP==j||OP==halt);
```

3.ALU

模块说明：

ALU 算数逻辑单元，用作算数逻辑运算：根据控制信号从输入的数据中选取对应的操作数，并进行对应的运算操作并输出result、zero、sign。

代码解释：

下一步我写了逻辑较为简单的ALU模块，在输入的所有值中任意一个变化时，就开始根据输入的ALUOp控制信号执行不同的运算，经老师和助教提醒，case部分需要有default，所以就写了default时result为0，但实际上正常情况下不会出现这种情况，只是满足语法要求而已。ALU模块核心部分如下：

```
assign sign=result[31];
assign zero=(result==0);
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<B)&&(A[31]==B[31]))||((A[31]==1&&B[31]==0)))?1:0;
        3'b111:result<=A^B;
        default:begin
            result=0;
        end
    endcase
end
```

4.Instruction memory

模块说明：

Instruction Memory指令存储器，可以保存并输出指令。从文件读入所有的32位指令，并储存，根据输入的地址选择对应的指令输出。输出后将指令传给其他部分进行处理。

代码解释：

指令存储器是将从文件里读入的指令，根据输入的PC的值来选择指令进行输出，因为助教说无法32位一起读入，所以只能切割成8位的，再拼接成32位的，当RW或Iaddr值改变时，并且RW控制信号为1时，读取指令。代码的核心部分如下：

```
reg[7:0] instruction[0:127];
initial begin
  $readmemh("input.txt",instruction);
end
always@(RW or IAddr)begin
  if(RW)begin
    IDataOut[31:24]=instruction[IAddr];
    IDataOut[23:16]=instruction[IAddr+1];
    IDataOut[15:8]=instruction[IAddr+2];
    IDataOut[7:0]=instruction[IAddr+3];
  end
end
```

5.Data Memory

模块说明：

数据储存器负责在时钟下降沿时，根据控制信号将传入的数据根据传入的地址写入并保存，当读取时根据控制信号和传入的地址输出正确的数据。

代码解释：

数据存储器主要负责数据的读写与储存。在我设计的模块中，当RD和DAddr中任何一个值有改变时，并且此时RD信号为1时，就开始读取。当时钟下降沿，并且WR信号为1时，就开始写。设计核心部分如下：

```
reg[7:0] data[0:127];
always@(RD or DAddr)begin
    if(RD)begin
        DataOut[31:24]=data[DAddr];
        DataOut[23:16]=data[DAddr+1];
        DataOut[15:8]=data[DAddr+2];
        DataOut[7:0]=data[DAddr+3];
    end
end
always@(negedge CLK)begin
    if(WR)begin
        data[DAddr]=DataIn[31:24];
        data[DAddr+1]=DataIn[23:16];
        data[DAddr+2]=DataIn[15:8];
        data[DAddr+3]=DataIn[7:0];
    end
end
```

6.PC

模块说明:

PC 程序计数器，用来存放当前执行指令的地址。接收PC Choose模块传进来的下一条PC并等待输出。

代码解释:

PC程序计数器模块先初始化了PC值为0，然后在时钟上升沿或者是Reset信号下降沿开始，如果Reset为0，则PC清零，如果Reset不为0，则下一个输出的PC就是经过PCchoose传进来的新PC值。PC模块核心部分如下：

```
initial begin
    nextPC=0;
end
always@(posedge CLK or negedge Reset)begin
    if(Reset==0)begin
        nextPC<=0;
    end
    if(Reset!=0&&PCWre==1)begin
        nextPC<=inPC;
    end
end
```

7. PCchoose

模块说明:

用来计算下一个指令的地址并传给PC。根据不同的PCSrc信号选择不同的跳转方式，输出正确的下一条指令PC。

代码解释:

PCchoose模块负责根据接收到的控制信号来计算下一个PC值，这一模块和图中的有较大的不同，相当于把图中右上角和左下角的几个模块合为一体，我认为这样更简洁一些。其中当控制信号PCsrc为00时，nextPC<=PC4；当控制信号PCsrc为01时，nextPC<=PC4+singedImmediate*4；当控制信号PCsrc为10时，nextPC<={PC4[31:28],addr[25:0],2'b00}。该模块设计的核心部分如下:

```
wire[31:0] PC4=curPC+4;
always@(*)begin
    if(PCSrc==2'b00)begin
        nextPC<=PC4;
    end
    if(PCSrc==2'b01)begin
        nextPC<=PC4+singedImmediate*4;
    end
    if(PCSrc==2'b10)begin
        nextPC<={PC4[31:28],addr[25:0],2'b00};
    end
end
```

8. Register File

模块说明:

通用寄存器组中存放数据都是32位的，有32个寄存器。负责存放程序需要用到的临时数据。根据控制信号和输入的寄存器号进行对应的读写操作。

代码解释:

通用寄存器组主要负责寄存器的读写和对数据的存储，一共有32个，这里需要注意一点就是0号寄存器的值是不能改变的，它一直为0，所以在写代码时需要特别注意一下0号寄存器。其余就是根据传入的地址去找到相应的寄存器，然后进行读写操作。还需注意就是写操作是需要沿时钟下降沿进行，这样比较稳定。这部分代码的核心部分如下:

```
reg[31:0] register[0:31];
integer i;
initial begin
  for(i=0;i<32;i=i+1)register[i]<=0;
end

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

always@(negedge CLK)begin
    if (WriteReg && WE) begin
        register[WriteReg]<=WriteData;
    end
end
```

9.零扩展和符号位扩展部分非常简单，就是根据传入的控制信号，对16位的数据进行零扩展或符号位扩展，成为32位的数据。代码核心部分如下：

```
always@(*)begin
  extended[15:0]<=immediate;
  if(ExtSel==0)begin
    extended[31:16]<=16'h0000;
  end
  else begin
    extended[31:16]<=immediate[15] ? 16'hffff : 16'h0000;
  end
end
```

10.Mux多选器我写了两种，一种是选择32位数据的，一种是选择5位数据的，思路很简单，就不赘述了，五位的代码如下，32位的同理：

```
module Mux5(
    input choice,
    input[4:0] in0,
    input[4:0] in1,
    output[4:0] out
);
    assign out=choice?in1:in0;
endmodule
```

11.顶层文件，要把所有的线声明一遍，然后将所有的模块进行实例化，再连接起来，这个过程和C语言程序设计中的函数比较相似，所以这个过程相对来说比较简单，但是需要非常细致，只需要把接口分配正确即可。部分核心代码如下：

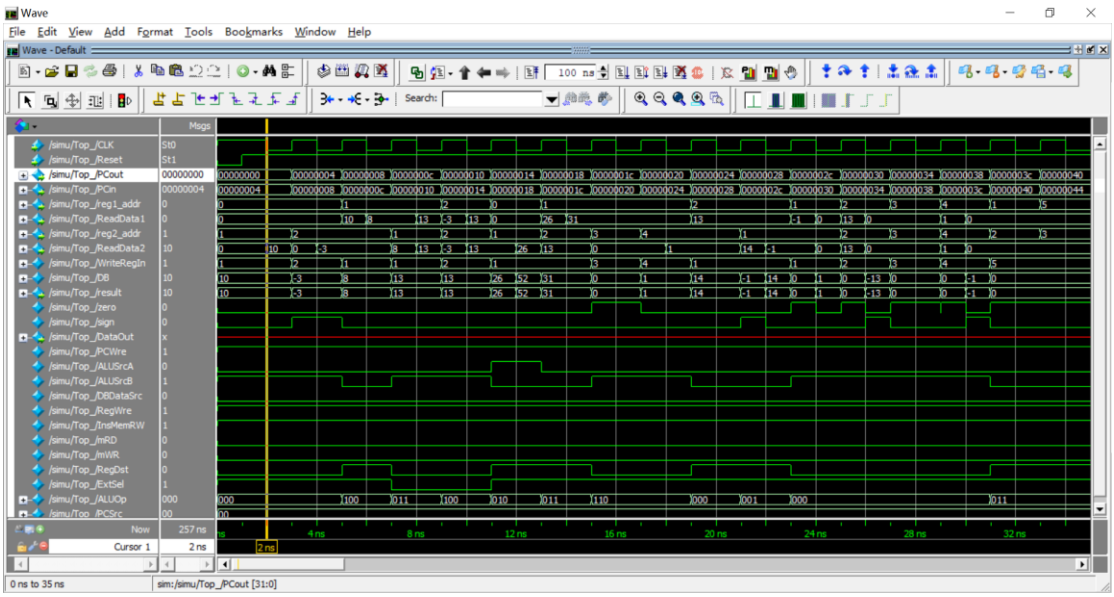
```
controlUnit controlUnit_(PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre,InsMemRW,mRD,mWR,RegDst,ExtSel,ALUOp,PCSrc,
                         zero,IDataOut[31:26],IDataOut[5:0],sign);
ALU ALU_(ALUAin,ALUBin,ALUOp,sign,zero,result);
PC PC_(PCWre,PCin,CLK,Reset,PCout);
PCchoose PCchoose_(PCSrc,ExtendOut,PCout,IDataOut,PCin);
signzeroextend signzeroextend_(IDataOut[15:0],ExtSel,ExtendOut);
RegisterFile RegisterFile_(CLK,RegWre,PCout,IDataOut[25:21],IDataOut[20:16],WriteRegIn,DB,ReadData1,ReadData2);
Mux32 Mux32_4(DBDataSrc,result,DataOut,DB);
Mux32 Mux32_2(ALUSrcA,ReadData1,{27'b000000000000000000000000000,IDataOut[10:6]},ALUAin);
Mux32 Mux32_3(ALUSrcB,ReadData2,ExtendOut,ALUBin);
Mux5 Mux5_1(RegDst,IDataOut[20:16],IDataOut[15:11],WriteRegIn);
DataMem DataMem_(CLK,mRD,mWR,result,ReadData2,DataOut);
insMEM insMEM_(InsMemRW,PCout,IDataOut);
```

12.这样整个CPU设计就完成了，但还需要一个仿真模块来进行仿真操作，可以理解为用仿真模块来激发这个CPU的运行，仿真模块中，我写的Reset一开始是0，隔一个时间单位后变为1，然后CLK是先为1，隔时间间隔之后开始进入0、1、0、1的变换循环。这样设计能够使保证第一条指令的完整进行。仿真模块代码如下：

```
module simu;
    reg CLK=1;
    reg Reset=0;
    reg[31:0] i;
    Top Top_(CLK,Reset,curPC,Reg1,Reg2,ALU);
    initial begin
    #1 Reset=1;
      for(i=0;i<128;i=i+1)begin
      # 1 CLK=0;
      # 1 CLK=1;
      end
    end
endmodule
```

六、指令对应波形分析

经过漫长的debug，终于显示出正确的波形图，总体效果如下：



下面以测试文件3中的指令为例来分析16种不同指令的波形是否正确:

测试程序段三

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	addiu $1,$0,10	001001	00000	00001	0000000000001010	=	24 01 00 0a
0x00000004	addiu $2,$0,-3	001001	00000	00010	1111111111111101		24 02 ff fd
0x00000008	and $1,$1,$2	000000	00001	00010	0000100000100100		00 22 08 24
0x0000000C	ori $1,$1,5	001101	00001	00001	0000000000000101		34 21 00 05
0x00000010	andi $2,$2,15	001100	00010	00010	0000000000001111		30 42 00 0f
0x00000014	sll $1,$1,1	000000	00000	00001	0000100001000000		00 01 08 40
0x00000018	or $1,$1,$2	000000	00001	00010	0000100000100101		00 22 08 25
0x0000001C	slti $3,$1,31	001010	00001	00011	0000000000011111		28 23 00 1f
0x00000020	slti $4,$1,32	001010	00001	00100	0000000000100000		28 24 00 20
0x00000024	add $1,$2,$4	000000	00010	00100	0000100000100000		00 44 08 20
0x00000028	sub $1,$2,$1	000000	00010	00001	0000100000100010		00 41 08 22
0x0000002C	addiu $1,$1,1	001001	00001	00001	0000000000000001		24 21 00 01
0x00000030	addiu $2,$2,-13	001001	00010	00010	1111111111110011		24 42 ff f3
0x00000034	addiu $3,$3,-0	001001	00011	00011	0000000000000000		24 63 00 00
0x00000038	addiu $4,$4,-1	001001	00100	00100	1111111111111111		24 84 ff ff
0x0000003C	or $5,$1,$2	000000	00001	00010	0010100000100101		00 22 28 25
0x00000040	or $5,$5,$3	000000	00101	00011	0010100000100101		00 a3 28 25
0x00000044	or $5,$5,$4	000000	00101	00100	0010100000100101		00 a4 28 25
0x00000048	addiu $0,$0,-2	001001	00000	00000	1111111111111110		24 00 ff fe
0x0000004C	beq $5,$0,1	000100	00101	00000	0000000000000001		10 a0 00 01
0x00000050	halt	111111	00000	00000	0000000000000000		fc 00 00 00

0x00000054	addiu $1,$0,0	001001	00000	00001	0000000000000000		24 01 00 00
0x00000058	addiu $2,$0,-1	001001	00000	00010	1111111111111111		24 02 ff ff
0x0000005C	sw $2,0($1)	101011	00001	00010	0000000000000000		ac 22 00 00
0x00000060	lw $1,0($1)	100011	00001	00001	0000000000000000		8c 21 00 00
0x00000064	bne $1,$2,5	000101	00001	00010	0000000000000101		14 22 00 05
0x00000068	bltz $1,1	000001	00001	00000	0000000000000001		04 20 00 01
0x0000006C	halt	111111	00000	00000	0000000000000000		fc 00 00 00
0x00000070	j 0x78	000010	00000	00000	0000000000011110		08 00 00 1e
0x00000074	halt	111111	00000	00000	0000000000000000		fc 00 00 00
0x00000078	halt	111111	00000	00000	0000000000000000		fc 00 00 00
0x0000007C	halt	111111	00000	00000	0000000000000000		fc 00 00 00

下面重点分析上图标黄的十六条指令，分别作为16种指令的代表，为了方便表述，先统一介绍一下我输出的波形的命名：

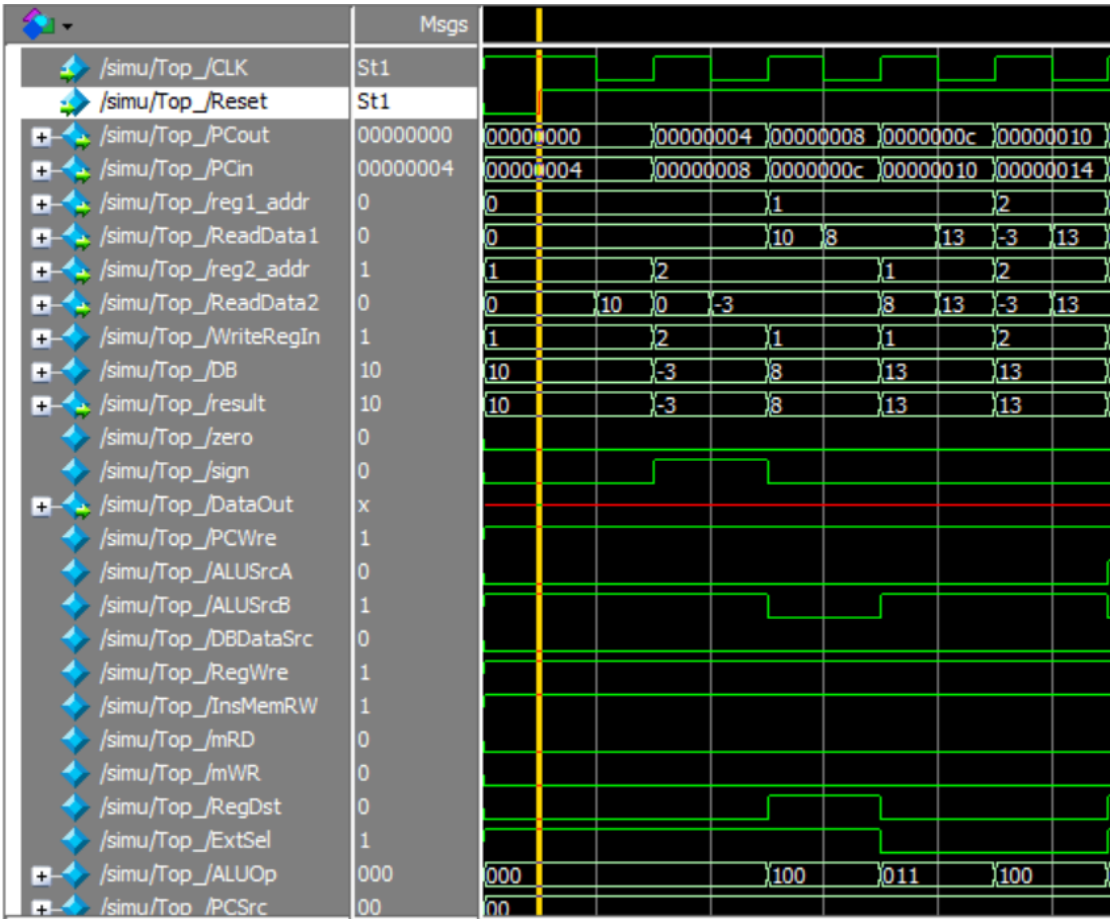
为了验证CPU设计的正确性，可以通过检查一些输入输出数据以及控制信号的波形来判断，我这里将要分析的数据及信号如下：

其中PCout、PCin分别指当前指令、下一条指令。reg1_addr，readData1，reg2_addr，readData2分别对应读寄存器组的两个寄存器的编号和读取结果：readreg1，readData1，readreg2，readData2。WrightRegIn指的是要写入的寄存器编号，DB指要写入的内容。Result指ALU运算的结果，zero、sign是ALU输出的判断是否为0和正负号的结果。DataOut指DataMemory的输出。接下来是一系列控制信号，命名与要求一致，不做赘述。

下面开始逐条分析：

（1）addiu

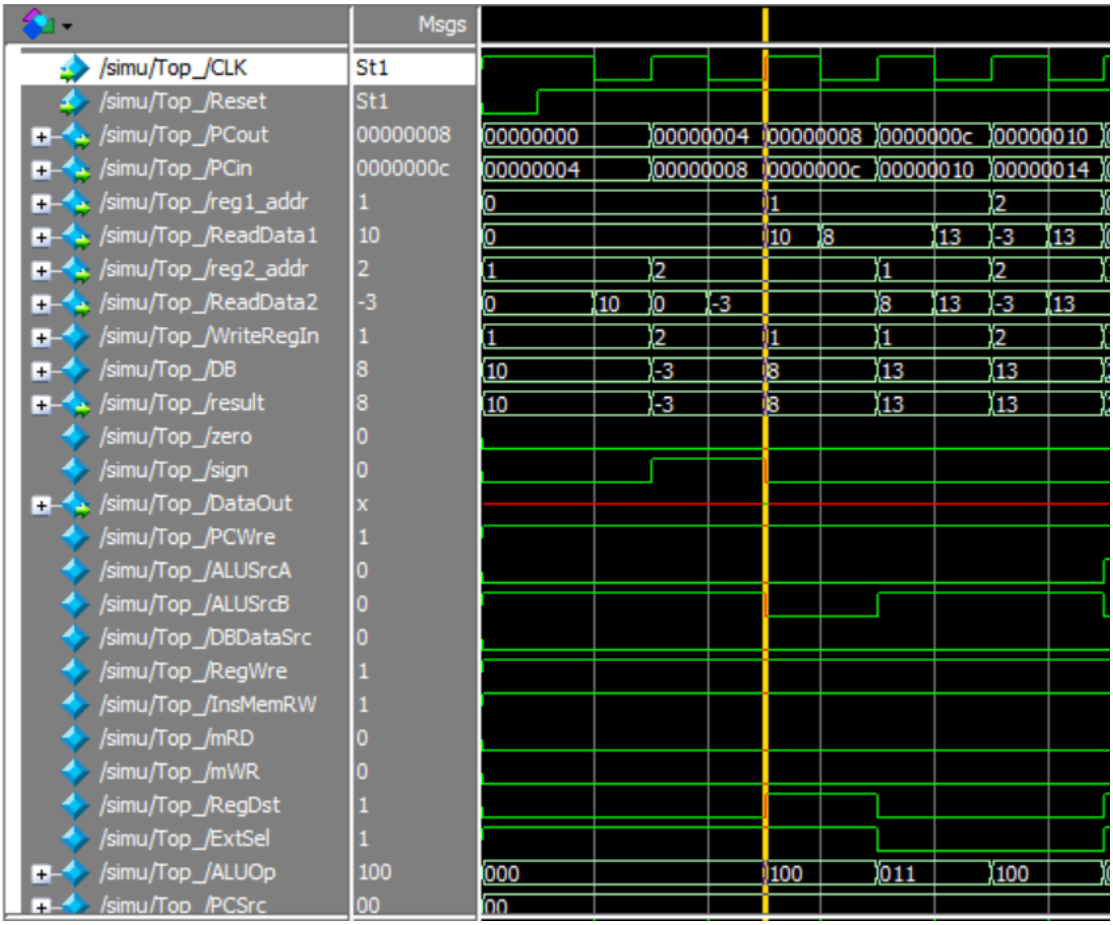
0x00000000	addiu $1,$0,10	001001	00000	00001	0000000000001010	=	24 01 00 0a



这条指令是要把$0中的值加上立即数10的结果写到1号寄存器中。如图所示，当前PC（PCout）为0，下一条指令PC（PCin）为4，顺序执行指令，符合实验要求和预期。st寄存器（reg1_addr）是0号寄存器，要写入的是一号寄存器（WrightRegIn），将0与10相加（ALUOp为000），写入的值（DB）和运算结果（result）都为10，并可见一号寄存器中的值在时钟下降沿由0变为10。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。ExtSel为1，立即数作符号位扩展。其他各个控制信号也都与之前我写的控制信号真值表对应正确。可验证这一条addiu指令在此CPU下正确执行。

(2) and

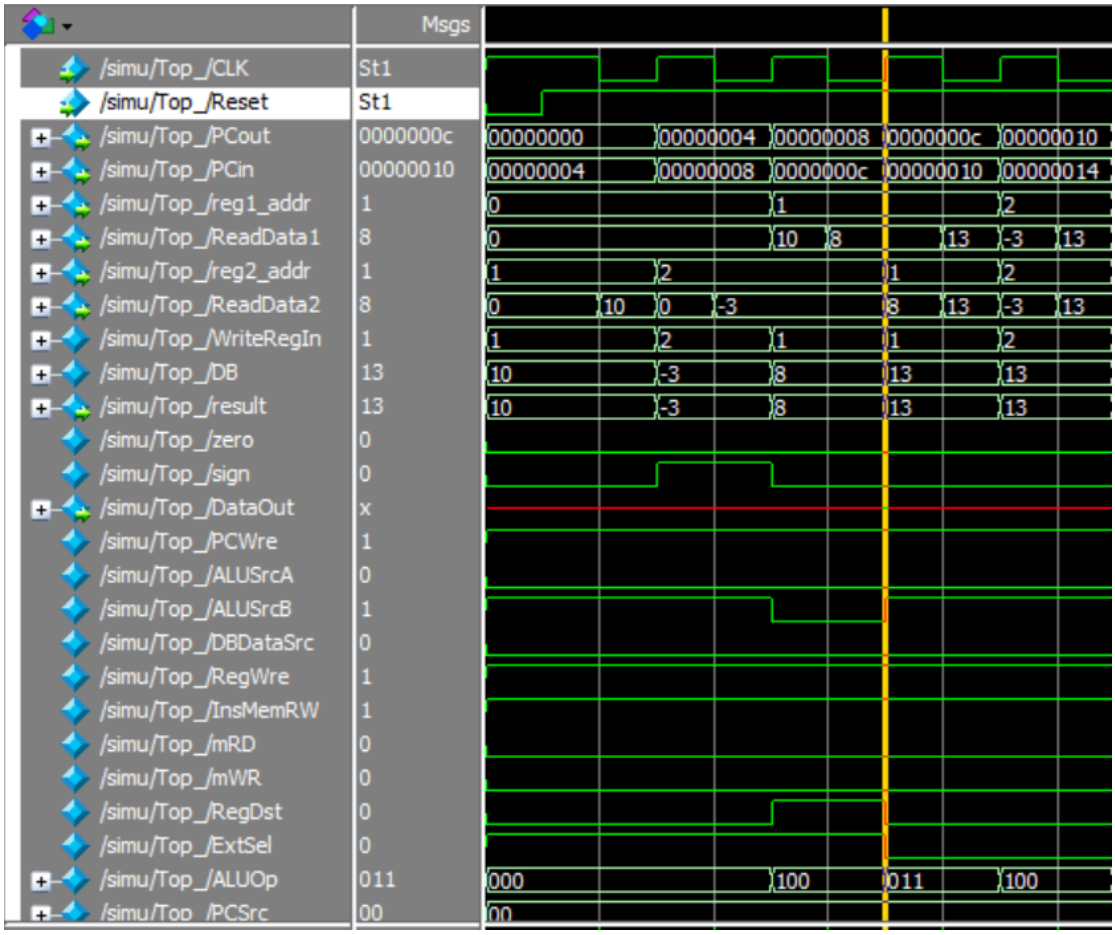
0x00000008	and $1,$1,$2	000000	00001	00010	0000100000100100		00 22 08 24



这一条指令是and，要把1号寄存器和2号寄存器中的值相与，结果写到1号寄存器中。如图所示，当前PC（PCout）值为08，下一条指令PC（PCin）为0C，顺序执行指令，符合实验要求和预期。1号寄存器中原来的值为10（ReadData1），2号寄存器中的值为-3（ReadData2），进行与运算（**ALUOp为100**）后结果为8，RegWre为1，在时钟下降沿将结果写入1号寄存器，这个变换过程也能从图中看出。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。其他控制信号和结果也都符合预期。可验证and指令在本CPU下可以正确执行。

（3）ori

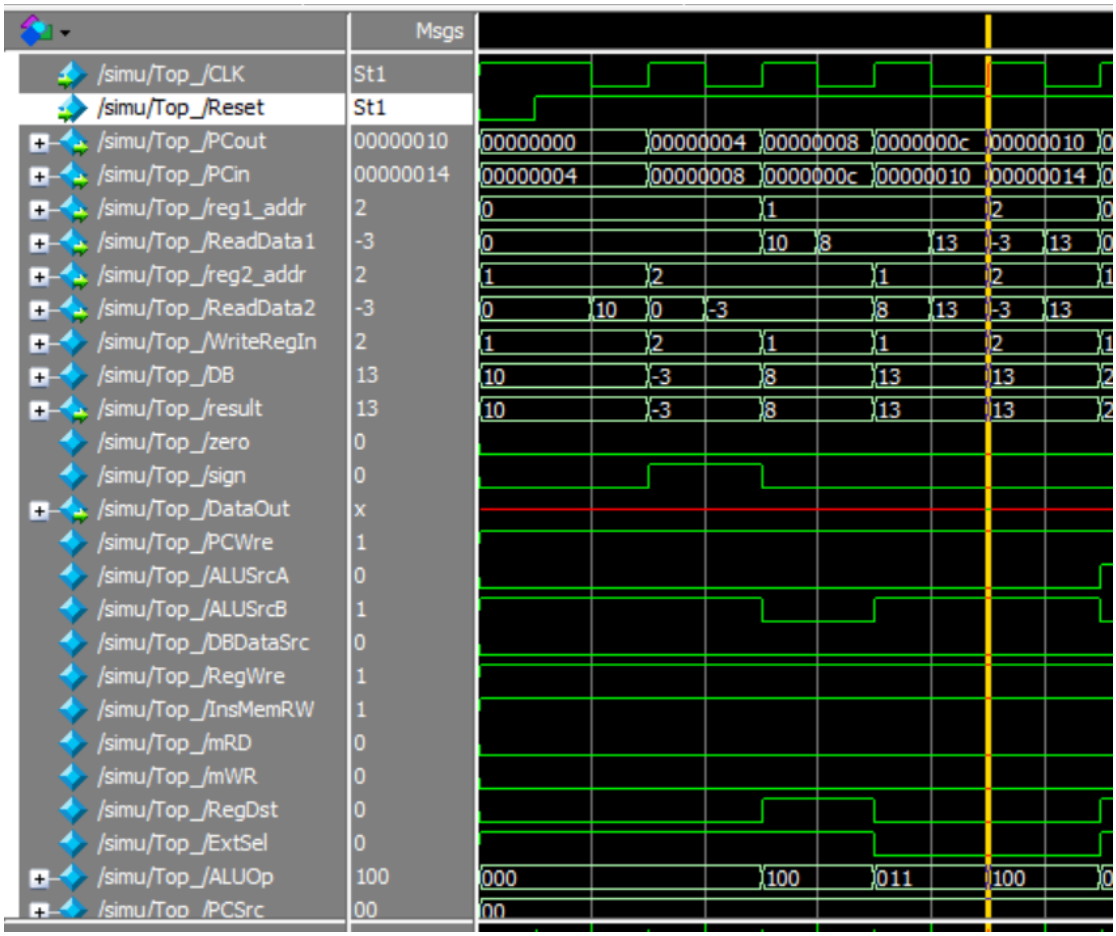
0x0000000C	ori $1,$1,5	001101	00001	00001	0000000000000101		34 21 00 05



这条指令是要把1号寄存器中的值与立即数5作或操作，将结果写入1号寄存器中，具体的控制信号参照实验报告第五部分的控制信号真值表。如图所示，当前PC（PCout）值为0C，PCsrc为00，下一条指令PC（PCin）为10，顺序执行指令，符合要求和预期。1号寄存器中原先值为8（ReadData1），与5作或操作（ALUOp为011）时，得到结果（result）为13，RegWre为1，将结果在时钟下降沿时写入寄存器1，这个变化过程可以从波形图中看出。ExtSel为0，立即数作0扩展。其他控制信号如图所示，经检查都符合之前写过的控制信号真值表。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。可验证ori指令在本CPU下可以正确执行。

（4）andi

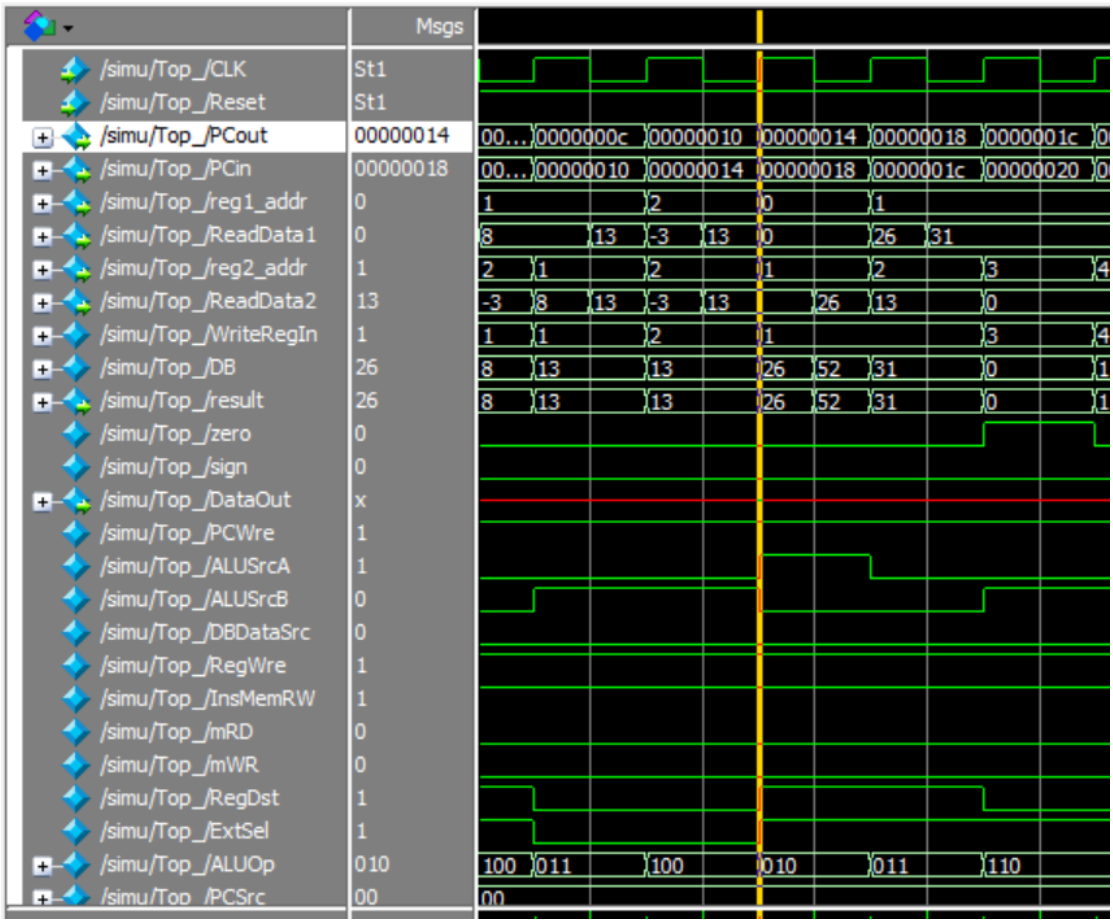
0x00000010	andi $2,$2,15	001100	00010	00010	0000000000001111		30 42 00 0f



这条指令是要把2号寄存器中的值与立即数15作与操作，将结果写入2号寄存器中，具体的控制信号参照实验报告第五部分的控制信号真值表。如图所示，当前PC（PCout）值为10，PCsrc为00，下一条指令PC（PCin）为14，符合实验要求和预期。2号寄存器中原先值为8，与15作与操作（ALUOp为100）时，得到结果（result）为13，RegWre为1，将结果在时钟下降沿时写入寄存器2，这个变化过程可以从波形图中看出。ExtSel为0，立即数作0扩展。其他控制信号如图所示，经检查都符合之前写过的控制信号真值表。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。可验证andi指令在本CPU下可以正确执行。

(5) sll

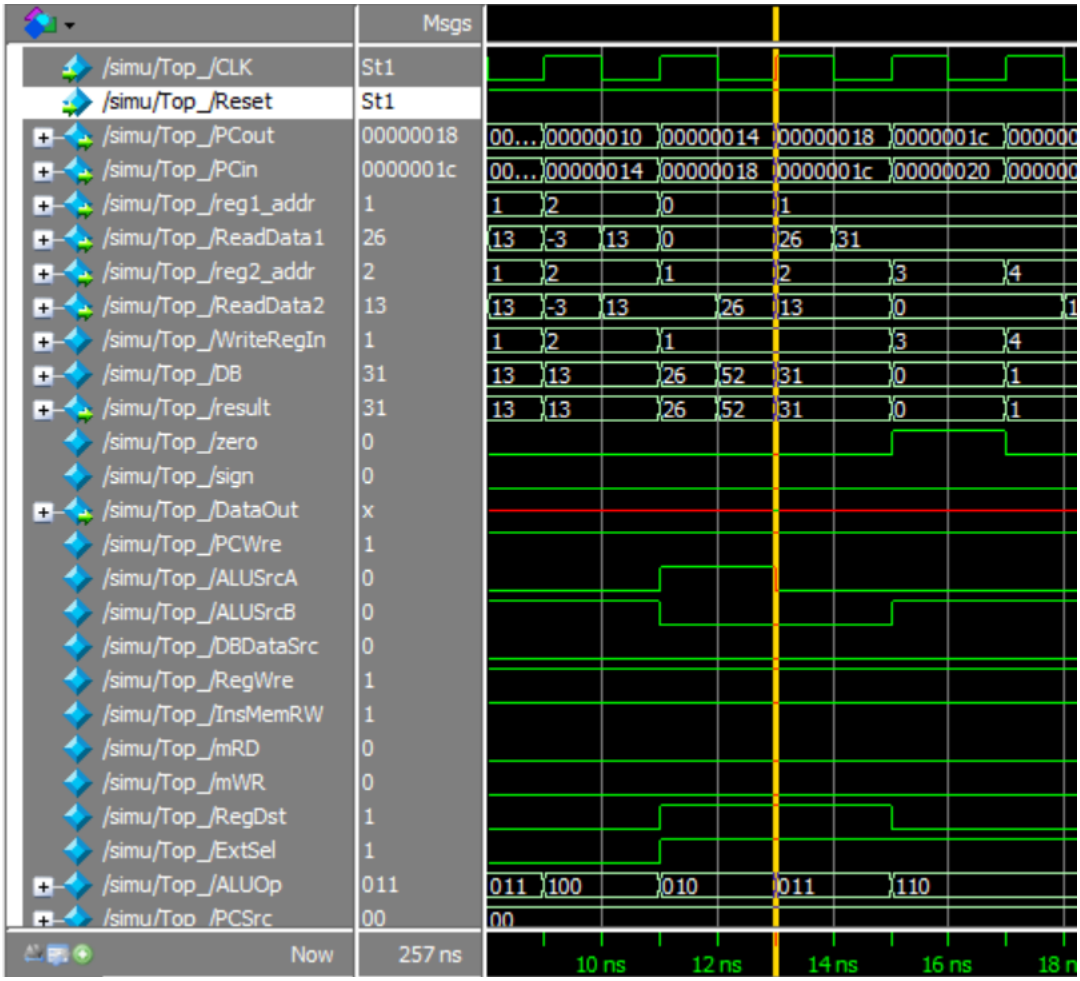
0x00000014	sll $1,$1,1	000000	00000	00001	0000100001000000		00 01 08 40



这条指令是要把1号寄存器中的值左移立即数1位，将结果写入1号寄存器中，具体的控制信号参照实验报告第五部分的控制信号真值表。如图所示，当前PC（PCout）值为14，PCsrc为00，下一条指令PC（PCin）为18，符合实验要求和预期。1号寄存器中原先值（ReadData1）为13，左移1位（ALUOp为010），得到结果（result）为26，RegWre为1，将结果在时钟下降沿时写入寄存器1，这个变化过程可以从波形图中看出。ExtSel为1，立即数作符号位扩展。其他控制信号如图所示，经检查都符合之前写过的控制信号真值表。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。可验证sll指令在本CPU下可以正确执行。

（6）or

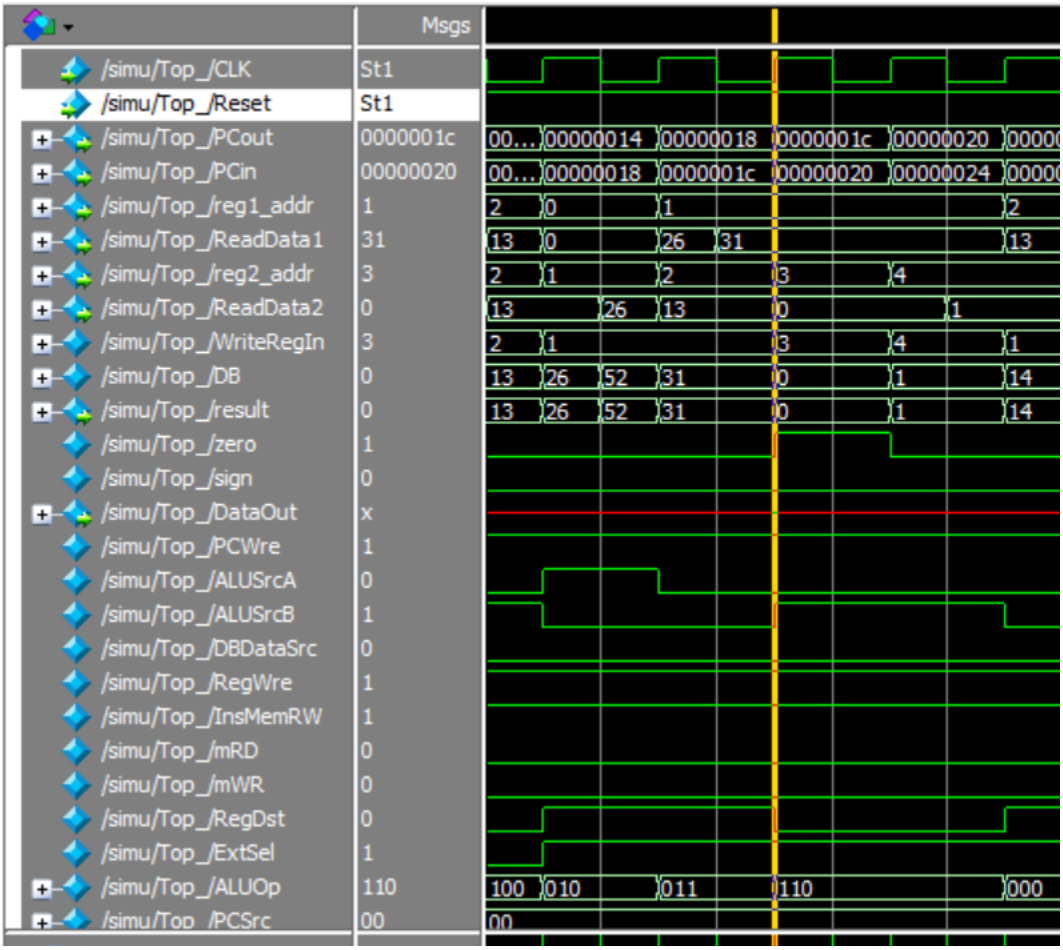
0x00000018	or $1,$1,$2	000000	00001	00010	0000100000100101		00 22 08 25



这一条指令是or，要把1号寄存器和2号寄存器中的值相或，结果写到1号寄存器中。如图所示，当前PC（PCout）值为18，下一条指令PC（PCin）为1C，顺序执行指令，符合实验要求和预期。1号寄存器中原来的值为26（ReadData1），2号寄存器中的值为13（ReadData2），进行或运算**（ALUOp为011）**后结果为31，RegWre为1，在时钟下降沿将结果31写入1号寄存器（WriteRegIn为1，DB为31），这个变换过程也能从图中在下降沿ReadData1值的改变看出。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。其他控制信号和结果也都符合预期。可验证or指令在本CPU下可以正确执行。

（7）slti

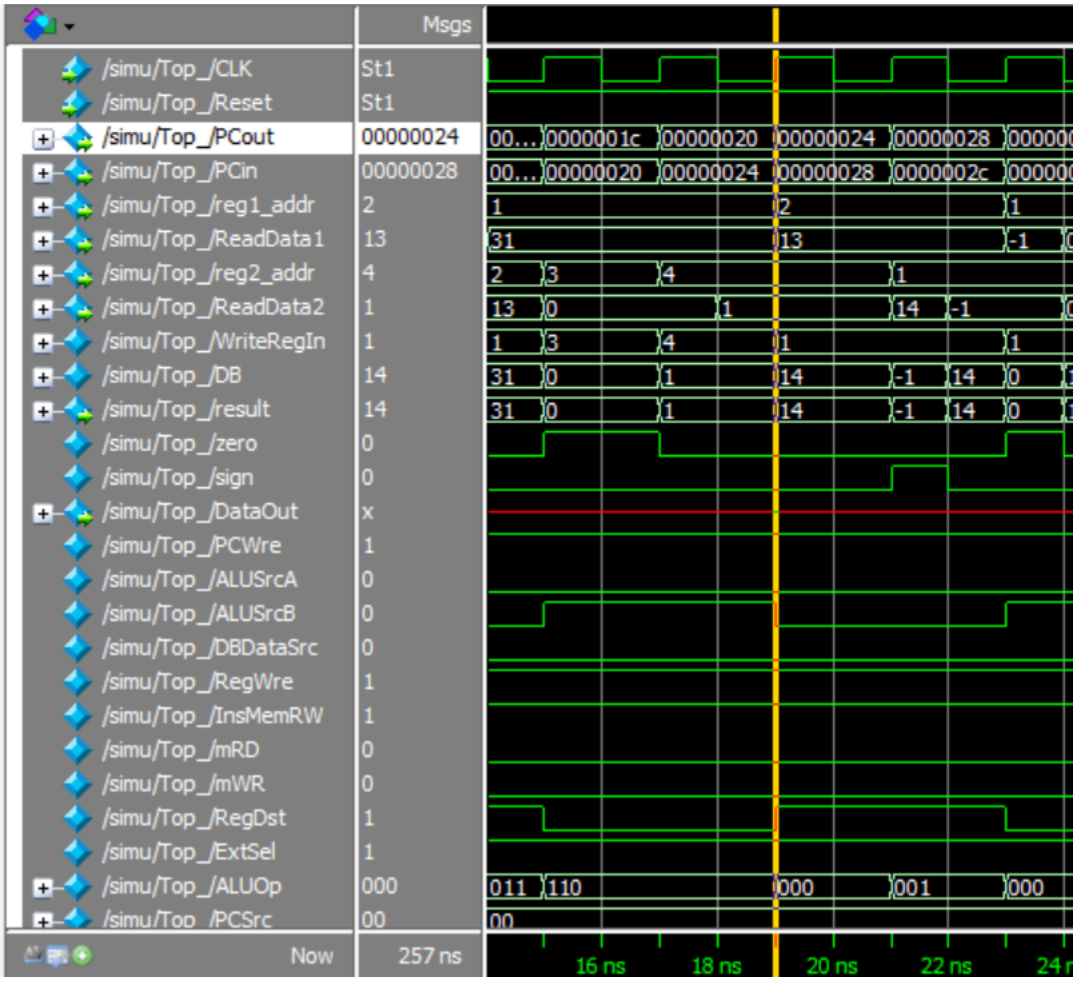
0x0000001C	slti $3,$1,31	001010	00001	00011	0000000000011111		28 23 00 1f



Slti，小于则置位，本条指令目的是比较1号寄存器中的值与立即数31，如果1号寄存器中的值小于31，则将结果1写入3号寄存器，否则，将结果0写入3号寄存器。如图所示，当前PC（PCout）值为1c，下一条指令PC（PCin）为20，顺序执行指令，符合实验要求和预期。1号寄存器中的值为31（ReadData1），等于立即数31，所以ALU作带符号比较（ALUOp为110）结果（result）为0，RegWre为1，在时钟下降沿将结果0写入3号寄存器（writeRegIn为3，DB为0）。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。其他控制信号和结果也都符合预期，如图。可验证slti指令在本CPU下可以正确执行。

（8）add

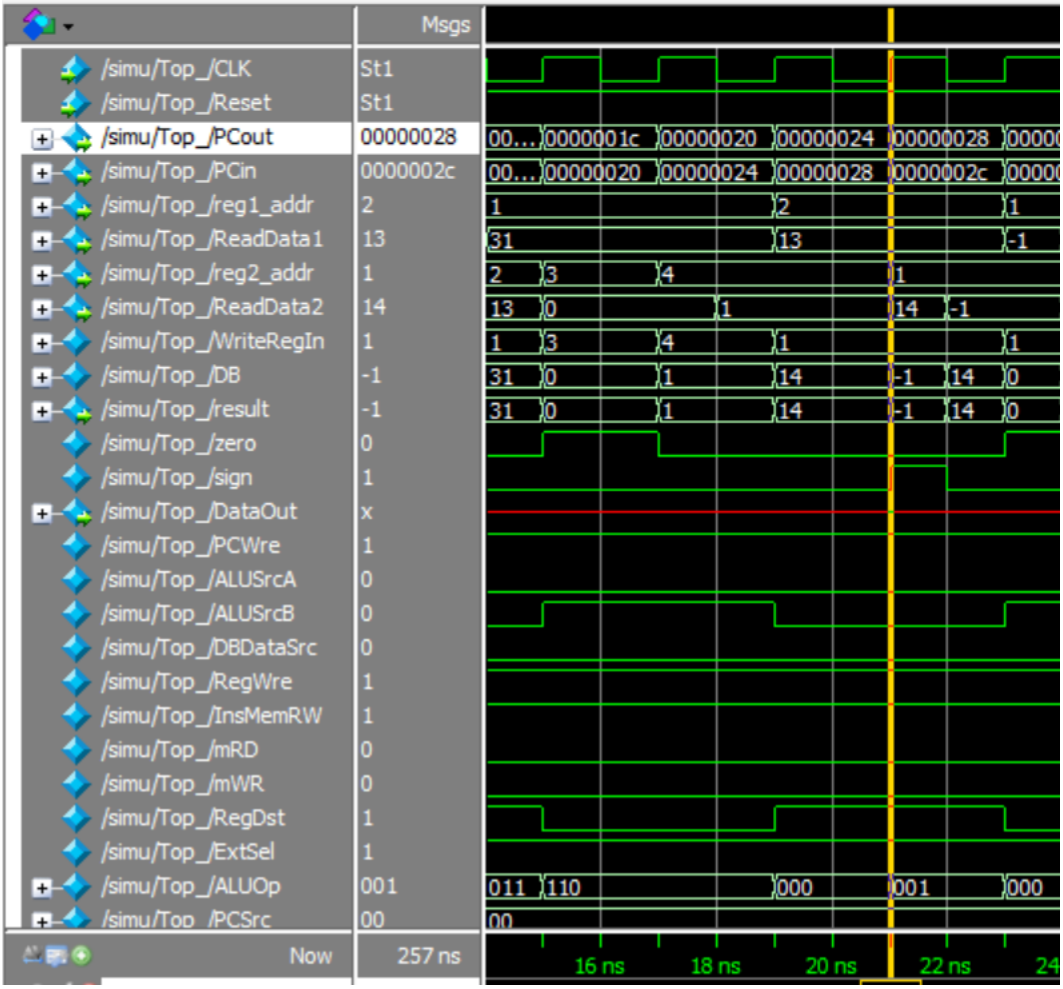
0x00000024	add $1,$2,$4	000000	00010	00100	0000100000100000		00 44 08 20



这一条指令是add，要把2号寄存器和4号寄存器中的值相加，结果写到1号寄存器中。如图所示，当前PC（PCout）值为24，下一条指令PC（PCin）为28，顺序执行指令，符合实验要求和预期。2号寄存器中的值为13（ReadData1），4号寄存器中的值为1（ReadData2），进行加运算（**ALUOp为000**）后结果为14，RegWre为1，在时钟下降沿将结果14写入1号寄存器（WriteRegIn为1，DB为14）。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。其他控制信号和结果也都符合预期。可验证add指令在本CPU下可以正确执行。

（9）sub

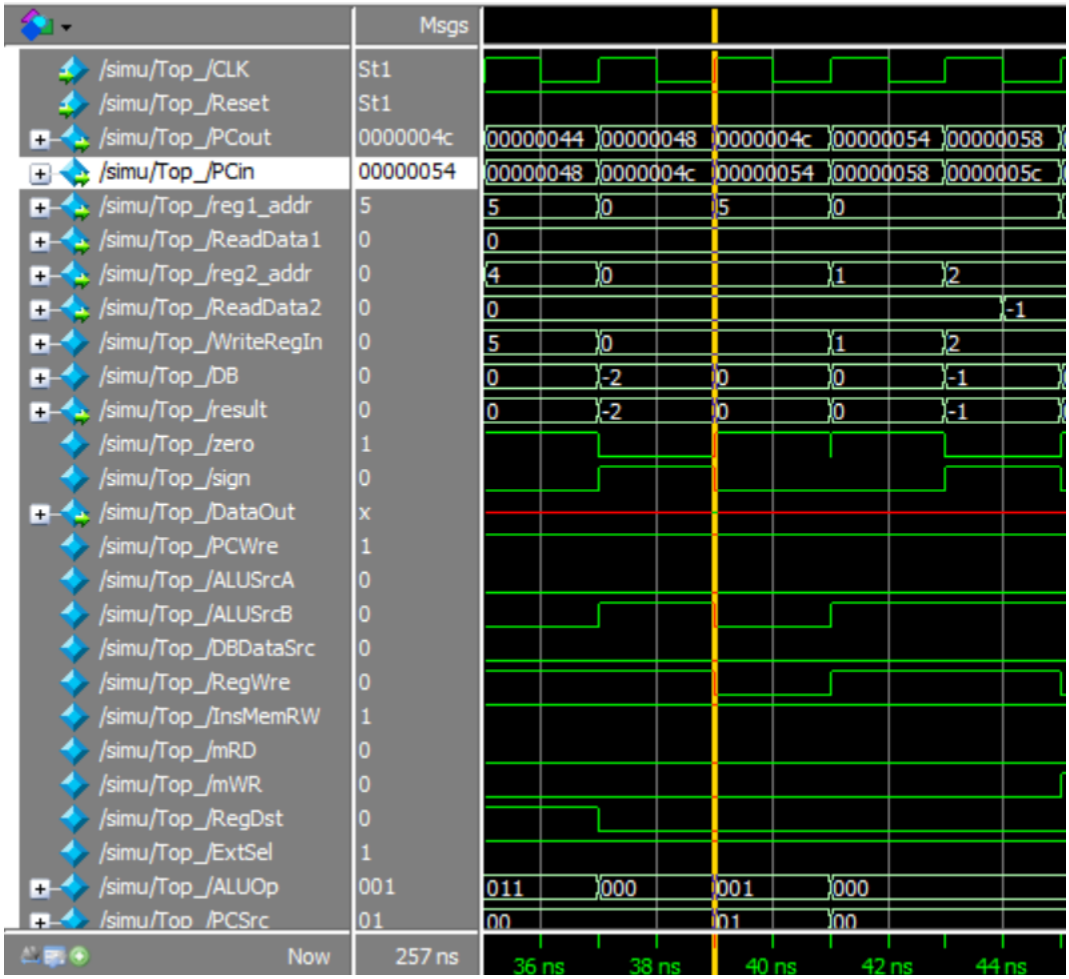
0x00000028	sub $1,$2,$1	000000	00010	00001	0000100000100010		00 41 08 22



这一条指令是sub，要把2号寄存器的值减去1号寄存器中的值，结果写到1号寄存器中。如图所示，当前PC（PCout）值为28，下一条指令PC（PCin）为2c，顺序执行指令，符合实验要求和预期。1号寄存器中原来的值为13（ReadData1），2号寄存器中的值为14（ReadData2），相减后（ALUOp为001）后结果为-1。RegWre为1，在时钟下降沿将结果-1写入1号寄存器（WriteRegIn为1，DB为-1），这个变换过程也能从图中在下降沿ReadData2值的改变看出。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。其他控制信号和结果也都符合预期。可验证sub指令在本CPU下可以正确执行。

（10）beq

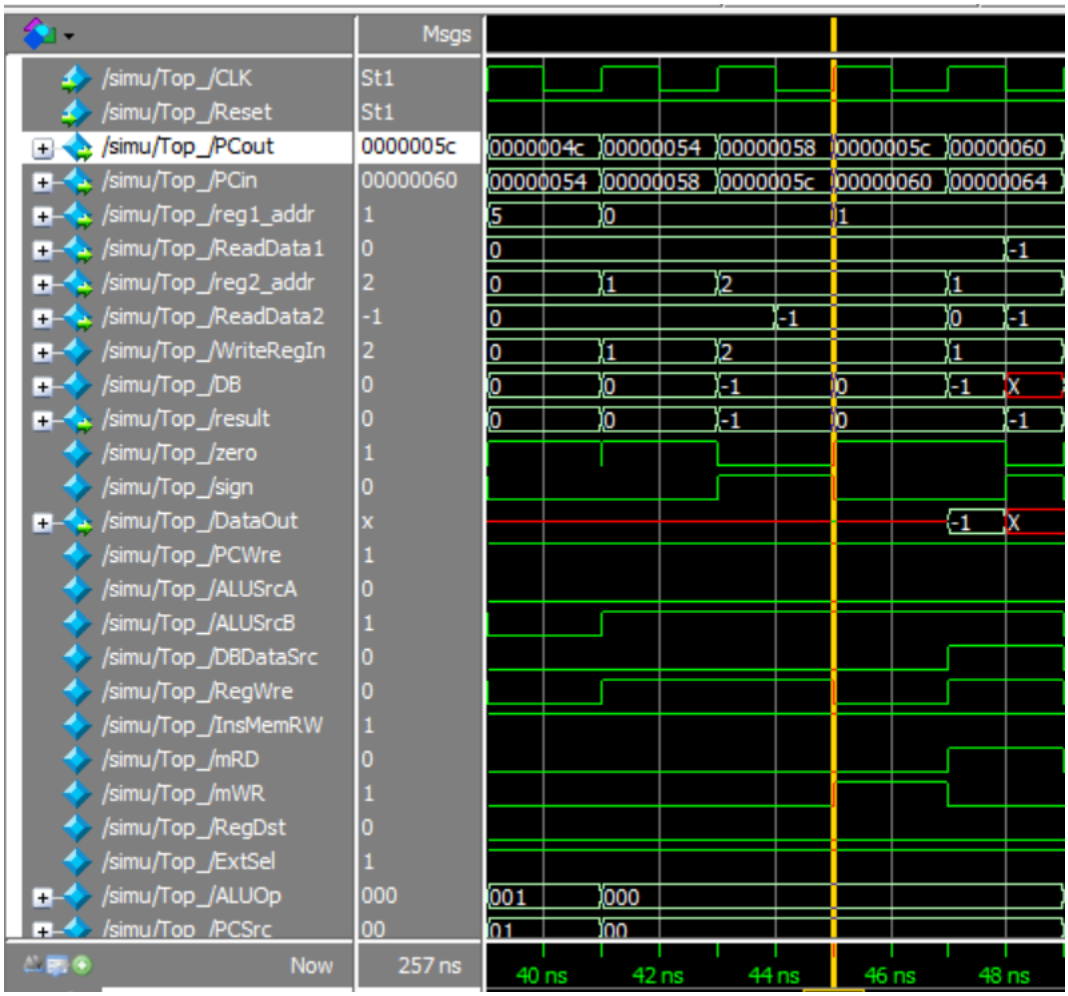
0x0000004C	beq $5,$0,1	000100	00101	00000	0000000000000001		10 a0 00 01



这条Beq指令目的是比较5号寄存器中的值与0号寄存器中的值，如果相等，则跳到这条指令的下一条指令的下一条，即这条指令往后的第2条指令，如果不相等则顺序执行。如图所示，可见5号寄存器中的值为0（ReadData1），0号寄存器中的值为0（ReadData2），通过ALU减法运算（ALUOp为001）得到zero为1，所以PCSrc为01，所以nextPC<=PC4+singedImmediate*4，故当前PC（PCout）值为4c，下一条指令PC（PCin）为54，符合预期。此外因为没有设计过储存器读写操作，所以DataOut的值未定义，是红线。其他控制信号和结果也都符合预期。可验证beq指令在本CPU下可以正确执行。

（11）sw

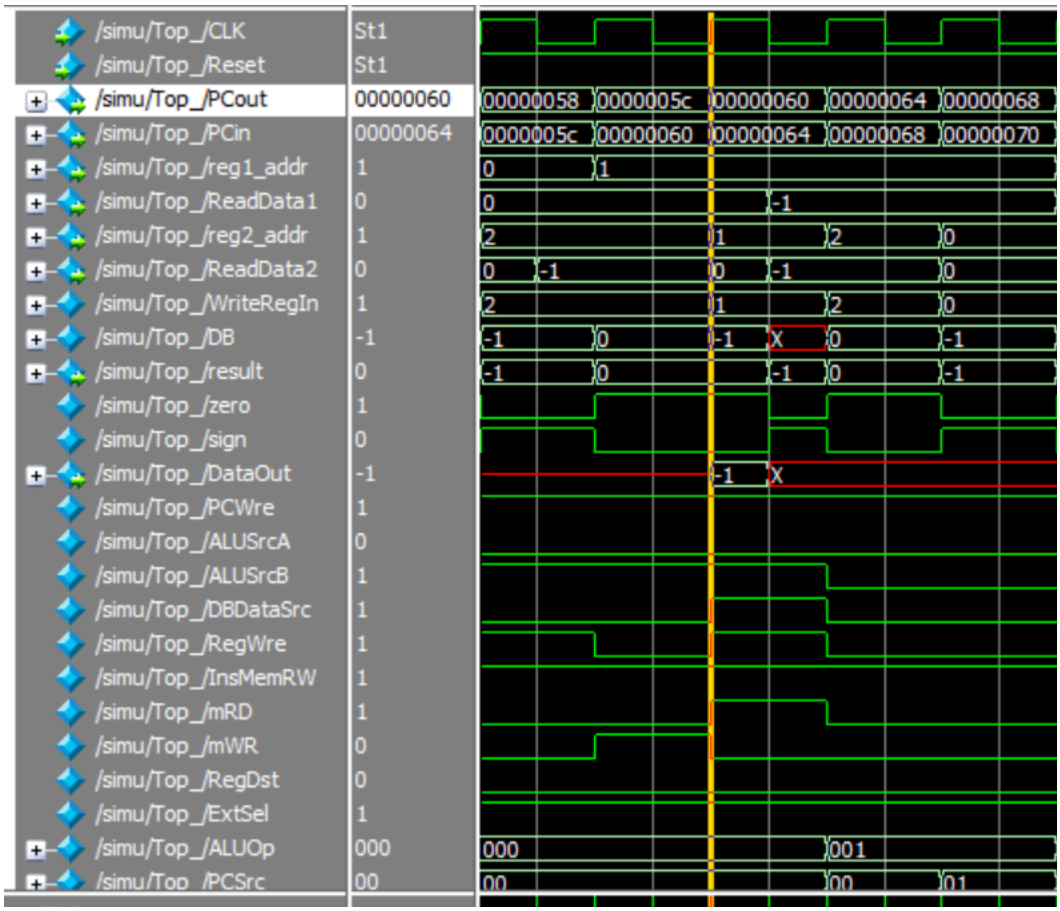
0x0000005C	sw $2,0($1)	101011	00001	00010	0000000000000000		ac 22 00 00



本条指令目的是将1号寄存器中的值加上偏移量0，求得一个地址，并将2号寄存器中的内容存到以这个地址开始的一个字的空间里。如图所示，当前PC（PCout）值为5c，下一条指令PC（PCin）为60，顺序执行指令，符合实验要求和预期。可见1号寄存器中的值为0（ReadData1），2号寄存器中的值为-1（ReadData2）。ALU加运算（ALUOp为000）的结果（result）为0，即要将-1存到存储器中0地址的位置。mWR为1，在时钟下降沿写入。写入结果的正确性将在下一条指令得到验证。此外，该指令其他控制信号如图都符合要求，正确。

（12）lw

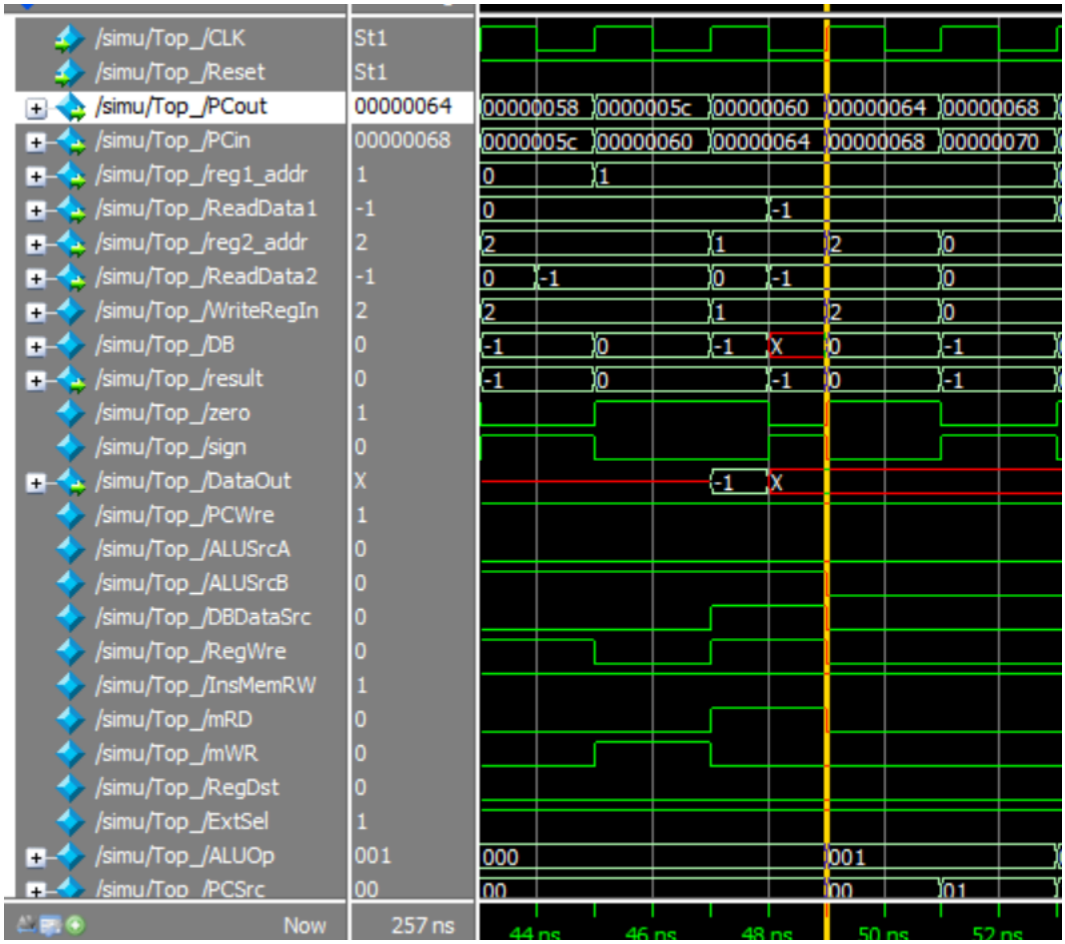
0x00000060	lw $1,0($1)	100011	00001	00001	0000000000000000		8c 21 00 00



本条指令目的是将1号寄存器中的值加上偏移量0，求得一个地址，并读取此地址开始的一个字中的内容，将其存到1号寄存器中。如图所示，当前PC（PCout）值为60，下一条指令PC（PCin）为64，顺序执行指令，符合实验要求和预期。可见1号寄存器中的值一开始为0（ReadData1），ALU加运算（ALUOp为000）的结果（result）为0，即要将存储器中0地址的数取出，写入1号寄存器中，由上一条指令可知此地址中存着-1，如图所示，DataOut的值为-1，说明取出的数正确。RegWre为1，在时钟下降沿写入，这一过程可以在图中一号寄存器的值的下降沿改变看出，由0变为-1。这样同时证明了上一条指令sw的正确性。此外，该指令其他控制信号如图都符合要求，正确。

（13）bne

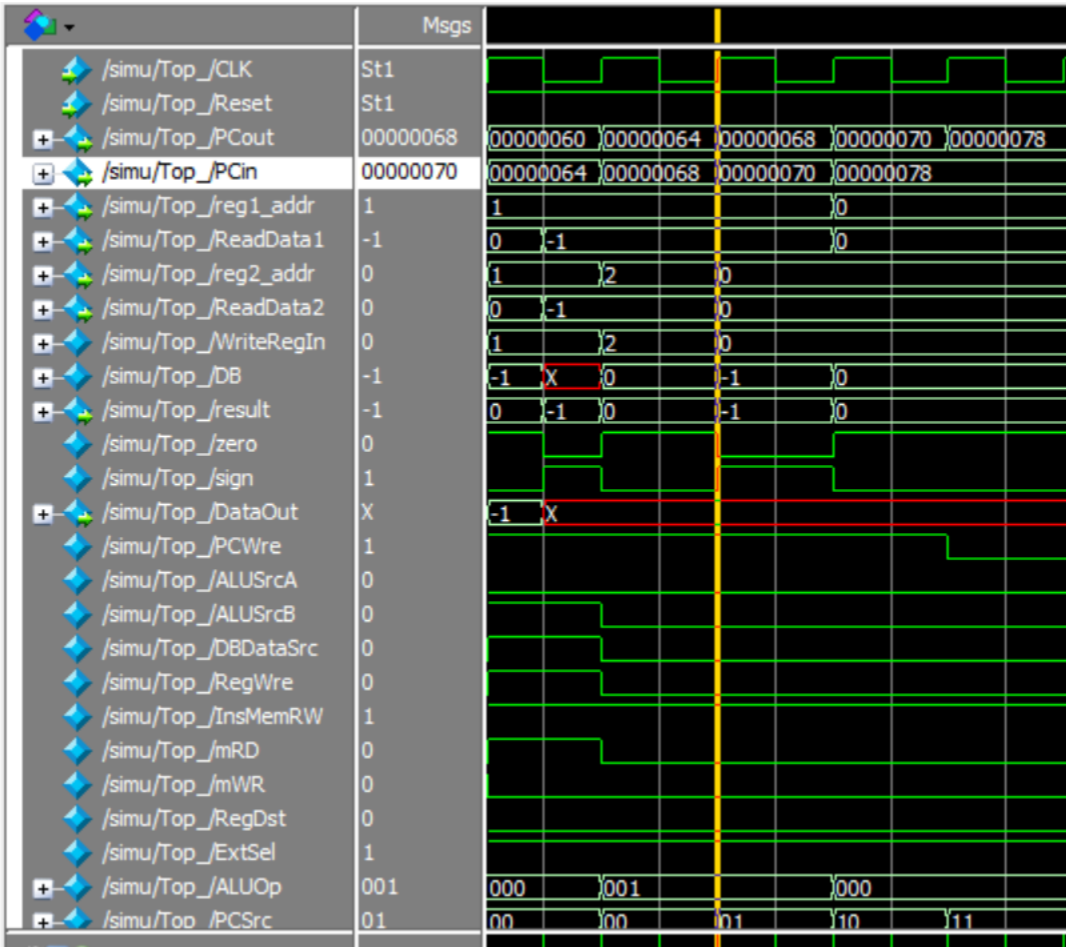
0x00000064	bne $1,$2,5	000101	00001	00010	0000000000000101		14 22 00 05



这条Bne指令目的是比较1号寄存器中的值与2号寄存器中的值，如果不相等，则跳到这条指令的下一条指令的下一条，即这条指令往后的第2条指令，如果相等则顺序执行。如图所示，可见1号寄存器中的值为-1（ReadData1），2号寄存器中的值为-1（ReadData2），通过ALU减法运算（ALUOp为001）得到zero为1，所以PCSrc为00，所以顺序执行下一条指令，故当前PC（PCout）值为64，下一条指令PC（PCin）为68，符合预期。其他控制信号和结果也都符合预期。可验证bne指令在本CPU下可以正确执行。

（14）bltz

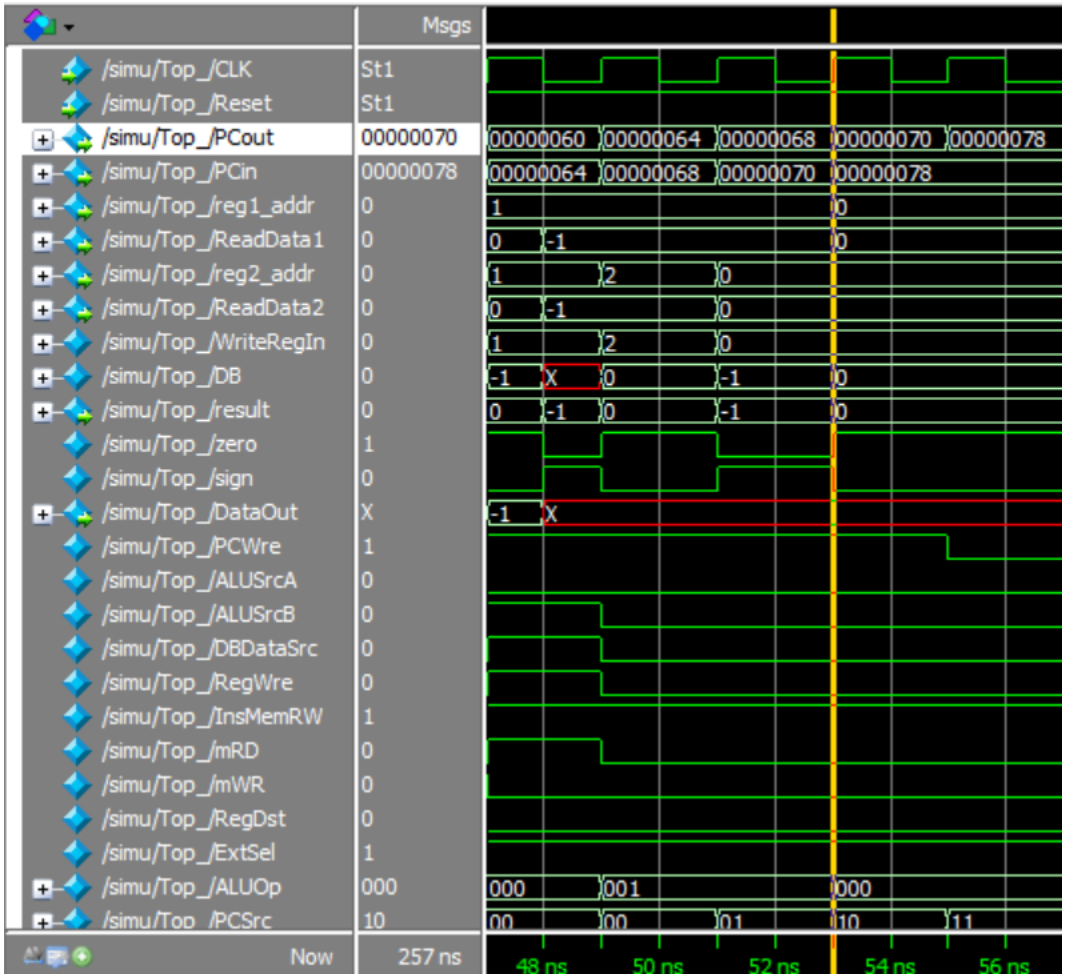
0x00000068	bltz $1,1	000001	00001	00000	0000000000000001		04 20 00 01



这条bltz指令的目的是，比较1号寄存器中的值与0，如果小于0，则pc←pc + 4 + sign_extend (**offset**) <<2，如果大于等于0，则pc ←pc + 4，顺序执行。如图所示，当前PC (PCout) 为68。图中可见1号寄存器中的值是-1，ALU中进行减法运算 (ALUOp为001)，将其与0相减，结果（result）为-1，sign为1，说明1号寄存器中的值小于0。此时PCSrc为01，所以下一个PC（PCin）为68+4+4*1=70（16进制），符合预期。其他控制信号和结果也都符合预期。可验证bltz指令在本CPU下可以正确执行。

（15）j

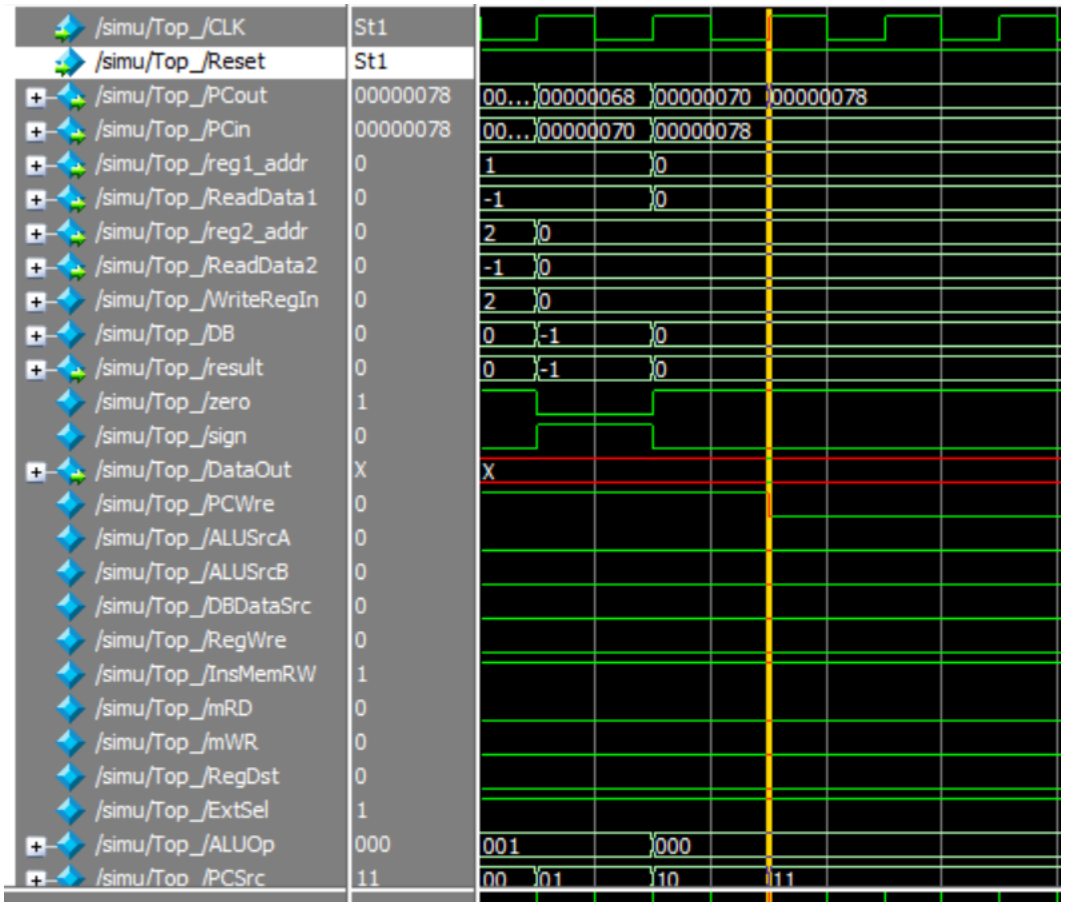
0x00000070	j 0x78	000010	00000	00000	0000000000011110		08 00 00 1e



本条J指令的目的是让它跳转到0x78的位置，如图所示，当前PC（PCout）值为70，下一条指令PC（PCin）为78，PCSrc为10。符合实验要求。

（16）halt

0x00000078	halt	111111	00000	00000	0000000000000000		fc 00 00 00



停机指令，不改变PC值，所以当前PC（PCout）和下一个PC（PCin）都是78。其他各种信号和数据都不再改变了。所以可验证halt指令在本CPU下可以正确执行。

七、实验问题与心得

这个实验是一个非常考验耐心的实验，因为从写控制信号真值表，到最后设计顶层文件，都有许多需要注意到的细节问题。当我开始着手写代码时，先是遇到了verilog语言使用上的很多问题，比如说分不清assign、=、<=等赋值方法，还有分不清reg和wire的用法，导致每次使用modelsim编译时，都会出现很多语法报错，只能硬着头皮一行行去改。

后来终于改完了语法问题，可以仿真了，却又是遇到了仿真波形与理想不同的问题，后来经检查，原来是控制信号真值表有的地方出错了，有一处把bltz写成了bne，于是就又需要去改真值表和控制单元模块。后来又发现了寄存器文件中也有关于写入数据的问题，都进行了修改。这一过程对我来讲是一个不小的挑战，因为同时有很多个文件，很多代码，有些小错误都是牵一发而动全身的，所以很难定位出bug的地方，只能一行行看代码，一行行检

查分析，有时候找出来由于自己粗心造成的问题甚至想打自己哈哈!

当我用刚开始发给我们的测试指令进行测试时，没有发现其他问题了，但新的测试指令发下来时，我又用新的测试，发现在第一条指令中，因为没有下降沿导致寄存器写入出错了，而旧的指令中恰巧没有遇到这个问题。于是我又修改simu文件，一开始设置了一个时间单位的暂停，使第一条指令能够正常执行。

总而言之，写出所有文件只花了3天时间，而找bug却花了整整一周。由于对verilog语言不熟悉，debug真的花费了太长时间，导致最后没有太多时间烧板子了，又不想继续熬夜，也不想拖到第三周验收，索性提前验收了。

虽然这次实验我完成的比较吃力，但是它让我得到了很好的锻炼。在实验设计的过程中让我明白了细心和静心的重要性，做实验的过程中我渐渐地变得更加平和，不会像刚开始那样，遇到一点bug就心浮气躁，而是学会了冷静地去处理解决。感谢这次实验。

所以希望我的多周期实验能够顺利一些，争取能够在两周之内烧完板子，有所进步。