《计算机组成原理实验》

实验报告

学 院 名 称 ：	计算机学院
专业（班级）：	计算机科学与技术(超算方向)
学 生 姓 名 ：	林天皓
学　　号 ：	18324034
时　　间 ：	2020 年 12 月 5 日

成 绩 ：

实　验 ： 单周期CPU设计与实现

一. 实验目的

1.理解 MIPS 常用的指令系统并掌握单周期 CPU 的工作原理与逻辑功能实现。

2.通过对单周期 CPU 的运行状况进行观察和分析，进一步加深理解。

二. 实验内容

1.利用 HDL 语言，基于 Xilinx FPGA basys3 实验平台，用 Verilog HDL 语言或 VHDL 语言来编写，实现单周期 CPU 的设计，这个单周期 CPU 能够完成 16 条 MIPS 指令，至少包含以下指令：

支持基本的内存操作如　lw，sw 指令

支持基本的算术逻辑运算如　add，sub，and，ori，slt，addi 指令

支持基本的程序控制如　beq，j 指令

2.掌握各个指令的相关功能并输出仿真结果进行验证，并最后在 FGPA 上实现，将其中的 alu 运算结果在开发板数码管上显示出来。

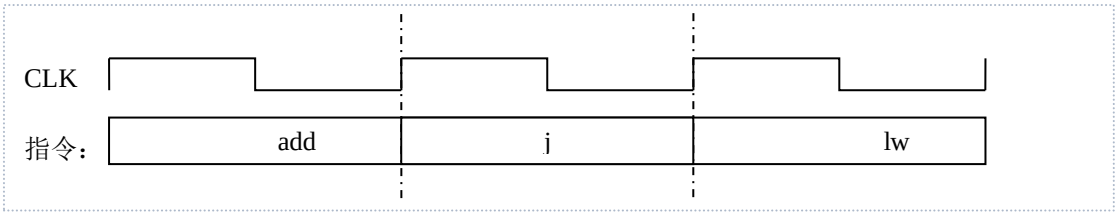
3.可拓展添加其他指令。

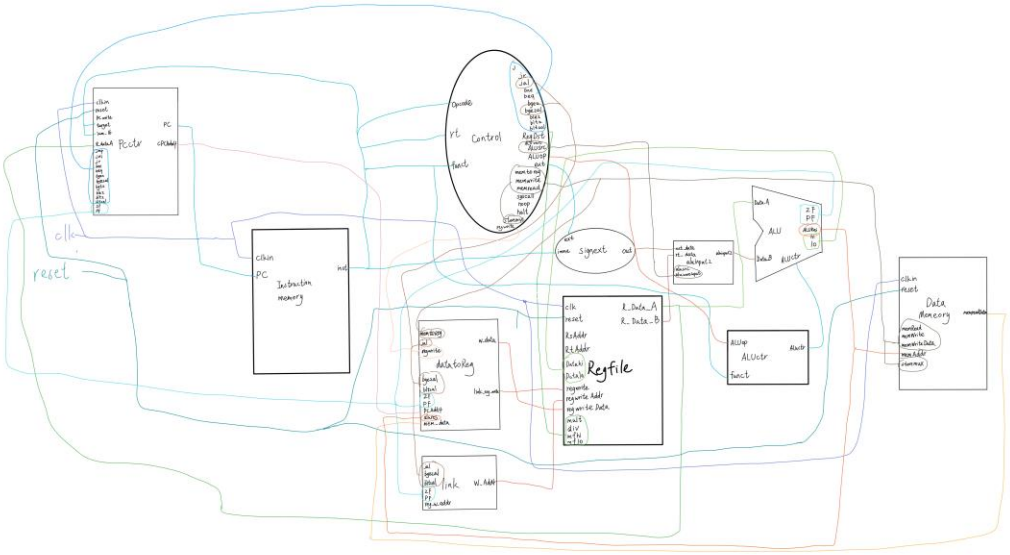
三. 实验原理

单时钟周期 CPU

单周期 CPU 的特点是每条指令的执行只需要一个时钟周期，一条指令执行完再执行下一条指令。再这一个周期中，完成更新地址，取指，解码，执行，内存操作以及寄存器操作。由于每个时钟上升沿时更新地址，因此要在上升沿到来之前完成所有运算，而这所有的运算除可以利用一个下降沿外，只能通过组合逻辑解决。这给寄存器和存储器 RAM 的制作带来了些许难度。且因为每个时钟周期的时间长短必须统一，因此在确定时钟周期的时间长度时，要依照最长延迟的指令时间来定，这也限制了它的执行效率。

单周期 CPU 在每个 CLK 上升沿时更新 PC，并读取新的指令。此指令无论执行时间长短，都必须在下一个上升沿到来之前完成。其时序示意如图Ⅰ。



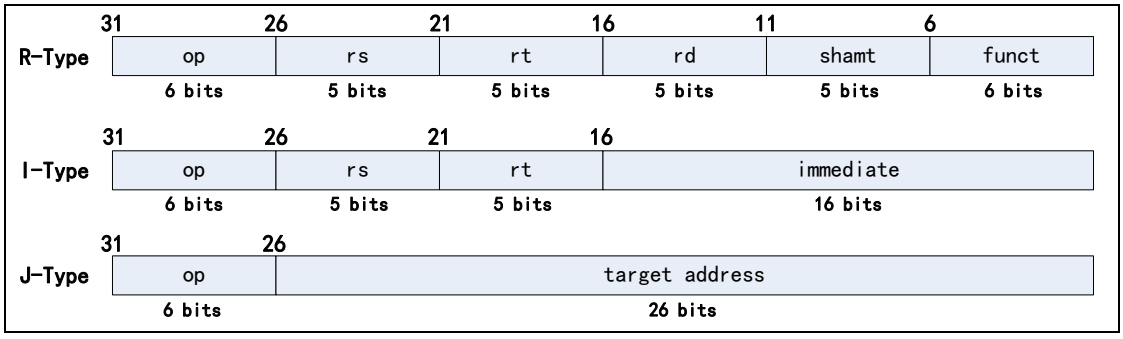


数据通路图如上，线路过多较复杂，可放大查看，尽可能把不同信号线使用不同颜色表示。

1. MIPS 指令集

本次实验共涉及三种类型的 MIPS 指令，分别为 R 型、I 型和 J 型，三种类型的 MIPS 指令格式定义如下：

- R（register）类型的指令从寄存器堆中读取两个源操作数，计算结果写回寄存器堆；
- I（immediate）类型的指令使用一个 16 位的立即数作为一个源操作数；
- J（jump）类型的指令使用一个 26 位立即数作为跳转的目标地址（target address）；



图III MIPS 指令集

一条指令的执行过程一般有下面的五个阶段，指令的执行过程就是这五个状态的重复

过程：

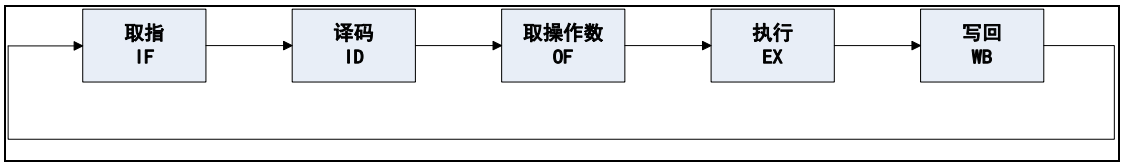


图 IV MIPS 指令集

在本次实验中，至少要完成 16 条指令的功能。

MIPS 的 31 种指令

助记符	指令格式						示 例	示例含义	操作及解释
BIT #	31..26	25..21	20..16	15..11	10..6	5..0			
R-类型	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2&S3	(rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2|S3	(rd)←(rs) | (rt); rs=$2,rt=$3,rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2^S3	(rd)←(rs)^(rt); rs=$2,rt=$3,rd=$1
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1= ~($2 | S3)	(rd)←~((rs) | (rt)); rs=$2,rt=$3,rd=$1
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1，无符号数
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	(rd)←(rt)<<shamt,rt=$2,rd=$1,shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (逻辑右移)
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (算术右移，注意符号位保留)
sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	(rd)←(rt)<<(rs), rs=$3,rt=$2,rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (逻辑右移)
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (算术右移，注意符号位保留)
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	(PC)←(rs)
I-类型	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2&10	(rt)←(rs)&(zero-extend)immediate,rt=$1,rs=$2

ori	001101	rs	rt	immediate	ori $1,$2,10	$1=$2|10	(rt)←(rs)|(zero-extend)immediate,rt=$1,rs=$2
xori	001110	rs	rt	immediate	xori $1,$2,10	$1=$2^10	(rt)←(rs)^(zero-extend)immediate,rt=$1,rs=$2
lui	001111	00000	rt	immediate	lui $1,10	$1=10*65536	(rt)←immediate<<16 & 0FFFF0000H，将 16 位立即数放到目的寄存器高 16 位，目的寄存器的低 16 位填 0
lw	100011	rs	rt	offset	lw $1,10($2)	$1=Memory[$2+10]	(rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2
sw	101011	rs	rt	offset	sw $1,10($2)	Memory[$2+10] =$1	Memory[(rs)+(sign_extend)offset]←(rt), rt=$1,rs=$2
beq	000100	rs	rt	offset	beq $1,$2,40	if($1=$2) goto PC+4+40	if ((rt)=(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2), rs=$1, rt=$2
bne	000101	rs	rt	offset	bne $1,$2,40	if($1≠$2) goto PC+4+40	if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2) , rs=$1, rt=$2
slti	001010	rs	rt	immediate	slti $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
sltiu	001011	rs	rt	immediate	sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
J-类型	op	address					
j	000010	address			j 10000	goto 10000	(PC)←((Zero-Extend) address<<2), address=10000/4
jal	000011	address			jal 10000	$31=PC+4 goto 10000	($31)←(PC)+4; (PC)←((Zero-Extend) address<<2), address=10000/4

除了上述基本指令之外，本次实验还实现了其他的扩展指令(16 条)

I-类型	op	rs	rt	immediate			
Bgez	000100	Rs	00001	offset	bgez $s, offset	if($s>=0)pc+=4+(offset << 2)	if $s == $t advance_pc (offset << 2)); else advance_pc (4);
Bgezal	000001	rs	10001	offset	bgezal $s, offset	if($s>=0) $31=pc+4 pc+=4+(offset << 2)	if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
bgtz	000111	rs	00000	offset	bgtz $s, offset	if($s>0)pc+=4+(offset << 2)	if $s > 0 advance_pc (offset << 2)); else advance_pc (4);
blez	000110	rs	00000	offset	blez $s, offset	if($s<=0)pc+=4+(offset << 2)	if $s <= 0 advance_pc (offset << 2)); else advance_pc (4);
bltz	000001	rs	00000	offset	bltz $s, offset	if($s<0)pc+=4+(offset << 2)	if $s < 0 advance_pc (offset << 2)); else advance_pc (4);
bltzal	000001	rs	10000	offset	bltzal $s, offset	if($s<0) $31=pc+4 pc+=4+(offset << 2)	if $s < 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
sh	101001	rs	rt	offset	sh $t,offset($s)	mem[$s+8]=$t	MEM[$s + offset] = $t; advance_pc

								(4);
sb	101000	rs	rt	offset			sb $t,offset($s)	mem[$s+8]=$t	MEM[$s + offset] = $t; advance_pc (4);
lh	100001	rs	rt	offset			lh $t,offset($s)	$t =mem[$s+8]	$t = MEM[$s + offset]; advance_pc (4);
lb	100000	rs	rt	offset			lb $t,offset($s)	$t =mem[$s+8]	$t = MEM[$s + offset]; advance_pc (4);
syscall	000000	-----	-----	----------001100				no opreation	advance_pc (4)
nop	000000	00000	00000	0000000000000000			nop	no opreation	advance_pc (4)
R-类型	op	rs	rt	rd	shamt	func			
mult	000000	rs	rt	00000	00000	011000	mult $s,$t	$s*$t	$LO = $s * $t; advance_pc (4)
div	000000	rs	rt	00000	00000	011010	div $s,$t	$s\$t	$LO = $s / $t; $HI = $s % $t; advance_pc (4);
mfhi	000000	00000	00000	rd	00000	010000	mfhi $d	$d=hi	$d = $HI; advance_pc (4);
mflo	000000	00000	00000	rd	00000	010010	mflo $d	$d=lo	$d = $lo; advance_pc (4);

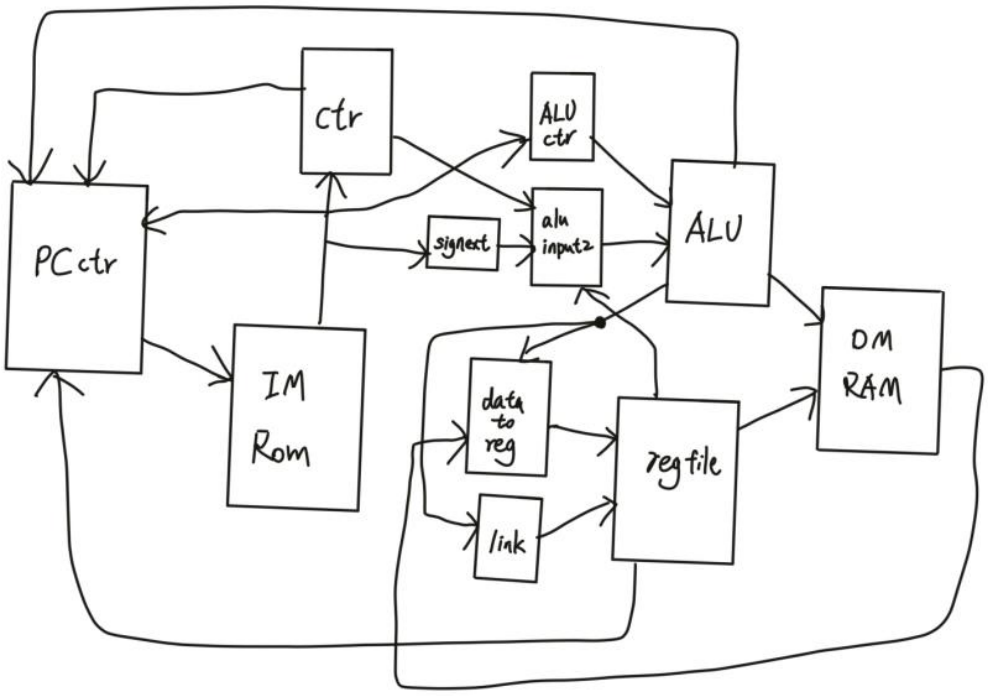
综上，在本次实验中，一共实现了47条指令。

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

CPU设计整体架构图：



为了在本次实验中扩展指令，对整体实现的架构进行了较大的修改，例如，对与各种对$ra进行操作的jal，bgtzal，blezal指令中，需要根据信号选择对应的写入寄存区和写入数据，对于blez，bgez，等等和零有关的branch指令，alu的运算数的来源除了来自立即数和寄存器读取值之外，还需要额外选择来自0来和零做比较，同时也需要在alu模块中加入表明运算结果正负号的PF信号来判断是否要执行这些分支指令，在这些扩展指令的实现过程中，对整个原始的设计架构进行了较大的改变。

接下来分模块分析：

一、PCctr模块

PCctr模块实现了下一步PC的逻辑，通过分析可以得知，下一PC可能有以下几种状态，

1.正常执行程序，PC=PC+4

2.执行无条件跳转指令，PC=跳转地址

3.执行jr指令时候PC=寄存器读取数据

4.执行分支指令条件成立时，PC=PC+4+offset<<2

PCctr模块实现代码如下：

```
module PCctr
(
    input clkin,
    input reset,
    //input [31:0] CPC,
    input PCWrite,
    input [25:0] target,
    input [15:0] imm_16,
    input [31:0] R_Data_A,
    input jmp,
    input jal,
    input jr,
    input bne,
    input beq,
    input bgez,
    input bgezal,
    input bgtz,
    input blez,
    input bltz,
    input bltzal,
    input ZF,
    input PF,
```

```
    output reg [31:0] PC,
    output [31:0]CPCadd4
);
wire[31:0] NPC,jumpAddr,branchAddr,jrAddr,extimm,nextPC,muxjump,muxbran
ch,muxjr,muxPCWrite;
assign NPC=PC;
assign CPCadd4=PC+4;
assign extimm = {{14{imm_16[15]}},imm_16[15:0], {2{1'b0}}};//branch 偏移
量
assign jumpAddr={PC[31:28],target, {2{1'b0}}};//jump 地址 target<<2
assign branchAddr = CPCadd4+extimm;//branch 跳转地址
assign jrAddr=R_Data_A;
assign muxjump=(jmp | jal)?jumpAddr:CPCadd4;
assign muxbranch=((bne & (~ZF))|(beq & ZF)|((bgez|bgezal) & (PF | ZF))
| (blez & (~PF))| (bgtz & PF) | ((bltz | bltzal) & (~PF & ~ZF)))?branch
Addr:muxjump;
assign muxjr = jr?jrAddr:muxbranch;
assign muxPCWrite = PCWrite?muxjr:NPC;
assign nextPC = muxPCWrite;
// initial begin
//     PC=32'h00000000;
// end

always @(negedge clkin ,posedge reset)
    begin
        if(~reset)
            PC<=nextPC;
        else
            PC<=32'h00000000;
    end
endmodule  //PCctr
```

二、指令储存器rom模块

使用自带的romIP核心，使用一层top文件，由于该rom是由时钟控制的，所以在我们的地址改变后，需要手动设置一个脉冲，以便能及时读取储存器中的指令。

实现代码如下：

```
`timescale 1ns / 1ps
module rom_top(
    input Clk,//系统时钟
    input Rst,//高电平复位
    input [31:0] PC,//rom 地址
    output [31:0] data//输出数据
);
```

```
wire [7:0]addr;
reg clk;
assign addr[7:0] = PC[9:2];
initial begin
    clk=0;
    #1;
    clk=1;
    #1
    clk=0;
    #1;
    clk=1;
end
always @(addr)begin
    clk=0;
    #1;
    clk=1;
    #1;
    clk=0;
    #1;
    clk=1;
end
blk_mem_gen_0 rom (
                  .clka(clk),    // input wire clka
                  .ena(1'b1),      // input wire ena 数据输出允许
                  .addra(addr),  // input wire [7 : 0] addra
                  .douta(data)  // output wire [15 : 0] douta
              );

endmodule
```

三、控制器模块

首先根据指令的操作描述各种需要的控制信号表:

	A	B	C	D	E	F	G	H	I
1		指令							
2	信号	add	addu	sub	subu	and	or	xor	nor
3	op	000000	000000	000000	000000	000000	000000	000000	000000
4	rs	rs	rs	rs	rs	rs	rs	rs	rs
5	rt	rt	rt	rt	rt	rt	rt	rt	rt
6	rd	rd	rd	rd	rd	rd	rd	rd	rd
7	shamt	00000	00000	00000	00000	00000	00000	00000	00000
8	funct	100000	100001	100010	100011	100100	100101	100110	100111
9	regdst	1	1	1	1	1	1	1	1
10	alusrc	0	0	0	0	0	0	0	0
11	aluZeroinput	0	0	0	0	0	0	0	0
12	memtoreg	0	0	0	0	0	0	0	0
13	regwrite	1	1	1	1	1	1	1	1
14	memread	0	0	0	0	0	0	0	0
15	memwrite	0	0	0	0	0	0	0	0
16	extop	1	1	1	1	1	1	1	1
17	aluop	+	+u	-	-u	and	or	xor	nor

	A	J	K	L	M	N	O	P	Q
1									
2	信号	slt	sltu	sll	srl	sra	sllv	srlv	srav
3	op	000000	000000	000000	000000	000000	000000	000000	000000
4	rs	rs	rs	00000	00000	00000	rs	rs	rs
5	rt	rt	rt	rt	rt	rt	rt	rt	rt
6	rd	rd	rd	rd	rd	rd	rd	rd	rd
7	shamt	00000	00000	shamt	shamt	shamt	00000	00000	00000
8	funct	101010	101011	000000	000010	000011	000100	000110	000111
9	regdst	1	1	1	1	1	1	1	1
10	alusrc	0	0	0	0	0	0	0	0
11	aluZeroinput	0	0	0	0	0	0	0	0
12	memtoreg	0	0	0	0	0	0	0	0
13	regwrite	1	1	1	1	1	1	1	1
14	memread	0	0	0	0	0	0	0	0
15	memwrite	0	0	0	0	0	0	0	0
16	extop	1	1	1	1	1	1	1	1
17	aluop	slt	sltu	sll	srl	sra	sllv	slrv	srav

	A	R	S	T	U	V	W	X
1								
2	信号	jr	mult	div	mfhi	mflo	syscall	noop
3	op	000000	000000	000000	000000	000000	000000	000000
4	rs	rs	rs	rs	00000	00000	00000	00000
5	rt	00000	rt	rt	00000	00000	00000	00000
6	rd	00000	00000	00000	rd	rd	00000	00000
7	shamt	00000	00000	00000	00000	00000	00000	00000
8	funct	001000	011000	011010	010000	010010	001100	000000
9	regdst	1	1	1	1	1	1	1
10	alusrc	0	0	0	0	0	0	0
11	aluZeroinput	0	0	0	0	0	0	0
12	memtoreg	0	0	0	0	0	0	0
13	regwrite	0	1	1	0	0	0	0
14	memread	0	0	0	0	0	0	0
15	memwrite	0	0	0	0	0	0	0
16	extop	1	1	1	1	1	1	1
17	aluop	x	mult	div	x	x	x	x

	A	B	C	D	E	F	G	H	I
19		指令							
20	信号	bltz	bgez	bltzal	bgezal	bgtz	beq	bne	blez
21	op	000001	000001	000001	000001	000111	000100	000101	000110
22	rs	rs	rs	rs	rs	rs	rs	rs	rs
23	rt	00000	00001	10000	10001	00000	rt	rt	00000
24	immediate	offset	offset	offset	offset	offset	offset	offset	offset
25	regdst	x	x	x	x	x	x	x	x
26	alusrc	0	0	0	0	0	0	0	0
27	aluZeroinput	1	1	1	1	1	0	0	1
28	memtoreg	0	0	0	0	0	0	0	0
29	regwrite	0	0	0	0	0	0	0	0
30	memread	0	0	0	0	0	0	0	0
31	memwrite	0	0	0	0	0	0	0	0
32	extop	x	x	x	x	x	x	x	x
33	aluop	-	-	-	-	-	-	-	-

	A	J	K	L	M	N	O	P	Q
19									
20	信号	addi	addiu	slti	sltiu	andi	ori	xori	lui
21	op	001000	001001	001010	001011	001100	001101	001110	001111
22	rs	rs	rs	rs	rs	rs	rs	rs	00000
23	rt	rt	rt	rt	rt	rt	rt	rt	rt
24	immediate	imme	imme	imme	imme	imme	imme	imme	imme
25	regdst	0	0	0	0	0	0	0	0
26	alusrc	1	1	1	1	1	1	1	1
27	aluZeroinput	0	0	0	0	0	0	0	0
28	memtoreg	0	0	0	0	0	0	0	0
29	regwrite	1	1	1	1	1	1	1	1
30	memread	0	0	0	0	0	0	0	0
31	memwrite	0	0	0	0	0	0	0	0
32	extop	1	1	1	1	0	0	0	1
33	aluop	+	+u	slt	sltu	and	ori	xor	lui

	A	R	S	T	U	V	W
19							
20	信号	lb	lh	lw	sb	sh	sw
21	op	100000	100001	100011	101000	101001	101011
22	rs	rs	rs	rs	rs	rs	rs
23	rt	rt	rt	rt	rt	rt	rt
24	immediate	offset	offset	imme	offset	offset	imme
25	regdst	0	0	0	0	0	0
26	alusrc	1	1	1	1	1	1
27	aluZeroinput	0	0	0	0	0	0
28	memtoreg	1	1	1	0	0	0
29	regwrite	1	1	1	0	0	0
30	memread	1	1	1	0	0	0
31	memwrite	0	0	0	1	1	1
32	extop	1	1	1	1	1	1
33	aluop	+	+	+	+	+	+

	A	B	C
37			
38		指令	
39	信号	j	jal
40	op	000010	000011
41	immediate	imme	imme
42	regdst	x	x
43	alusrc	x	x
44	aluZeroinput	x	x
45	memtoreg	x	x
46	regwrite	x	1
47	memread	x	x
48	memwrite	x	x
49	extop	x	x
50	aluop	x	x
51	jmp	1	1

根据上表，控制器模块负责由输入的指令转换为各种控制信号，为了扩展方便，并没有对一些信号进行压缩，直接采用了指令的命名方式，控制信号的说明和完整实现代码如下：

(代码较长，下一个模块内容在30页面开始)

```
`timescale 1ns / 1ps
module ctr(
           input [5:0] opCode,
           input [5:0] funct,
           input [4:0] rt,
           output reg regDst,  //0:rt,1:rd
           output reg aluSrc, //1:imme 立即数
           output reg aluZeroinput, //1:alu 输入 2 为
0 bltz bgez bltzal bgezal 使用
           output reg memToReg, //从内存写入寄存器
           output reg regWrite, //写寄存器使能
           output reg memRead, //读取内存使能
           output reg memWrite, //写入内存使能
           output reg ExtOp, //符号扩展方式，1 为 sign-extend，
0 为 zero-extend
           output reg[3:0] aluop, // 经过 ALU 控制译码决定 ALU 功能
           output reg jmp,//jump 指令:无条件跳转
           output reg jal,//jal 指令:跳转和链接
           output reg jr,//jr 指令:从寄存器跳转
           output reg bne,//bne 指令:不相等时跳转
           output reg beq,//beq 指令:相等时跳转
           output reg bgez,//bgez 指令:大于等于 0 时跳转
           output reg bgezal,//bgezal 指令:大于等于 0 时跳转并且链接
           output reg bgtz,//bgtz 指令:大于 0 时跳转
           output reg blez,//blez 指令:小于等于 0 时跳转
```

```
            output reg bltz,//bltz 指令:小于 0 时跳转
            output reg bltzal,//bltzal 指令:小于 0 时跳转并且链接
            output reg mult,//mult 指令:乘指令
            output reg div,//div 指令:除法指令
            output reg mflo,//mflo 指令:从 lo 寄存器设置寄存器值
            output reg mfhi,//mfhi 指令:从 hi 寄存器设置寄存器值
            output reg PCWrite,//1:CPU 继续运行，0:停机
            output reg syscall,//系统调用指令
            output reg noop,//不做操作指令
            output reg halt,//停机指令
            output reg [1:0]storemux//00:字节,01 半字,11 全字,load 指令和
store 指令的方式
        );

always@(opCode,funct,rt) begin
    // 操作码改变时改变控制信号
    case(opCode)
        // 'R 型' 指令操作码
        // 算术指令，syscall,mult,div,mufi,mflo,jr
        6'b000000: begin
            regDst = 1;
            aluSrc = 0;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0000;
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
```

```
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
            if(funct==6'b011000)
                mult=1;
            else if(funct==6'b011010)
                div=1;
            else if(funct==6'b010000)//muhi
            begin
                mfhi=1;
            end
            else if(funct==6'b010010)//mflo
            begin
                mflo=1;
            end
            else if(funct==6'b001000)
            begin
                jr=1;
                regWrite=0;
            end
            else if(funct==6'b001100)
            begin
                syscall=1;
                regWrite=0;
            end
            else if(funct==6'b000000)
            begin
                noop=1;//sll
            end
        end

        // 'J 型' 指令操作码
        // 'j' 指令操作码: 000010
        6'b000010: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
```

```
            aluop = 4'b0111;
            jmp =1;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        // 'jal' 指令操作码: 000011
        6'b000011: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;//56t4r
            aluop = 4'b0111;
            jmp =0;
            jal=1;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
```

```
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //'I'型指令操作码
        //addi 操作码 001000
        6'b001000: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0001;
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
```

```
        end
        //addiu 操作码 001001
        6'b001001: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0010;
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end
        //slti 操作码 001010
        6'b001010: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0111;
```

```
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //sltiu 操作码 001011
        6'b001011: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b1000;
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
```

```
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //andi 操作码 001100
        6'b001100: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0100;
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end
```

```
        //ori 操作码 001101
        6'b001101: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0101;
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //xori 操作码 001110
        6'b001110: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0110;
```

```
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //lui 操作码 001111
        6'b001111: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 1;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b1001;
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
```

```
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //lb 操作码 100000
        6'b100000: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 1;
            regWrite = 1;
            memRead = 1;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0001;//加法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end
```

```
        //lh 操作码 100001
        6'b100001: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 1;
            regWrite = 1;
            memRead = 1;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0001;//加法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //lw 操作码 100011
        6'b100011: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 1;
            regWrite = 1;
            memRead = 1;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0001;//加法
```

```
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //sb 操作码 101000
        6'b101000: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 1;
            ExtOp = 1;
            aluop = 4'b0001;//加法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
```

```
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //sh 操作码 101001
        6'b101001: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 1;
            ExtOp = 1;
            aluop = 4'b0001;//加法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end
```

```
        //sw 操作码 101011
        6'b101011: begin
            regDst = 0;
            aluSrc = 1;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 1;
            ExtOp = 1;
            aluop = 4'b0001;//加法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

    //B 指令
        //bltz bgez bltzal bgezal 操作码 000001
        6'b000001: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 1;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
```

```
            aluop = 4'b0011;//减法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
            if(rt == 5'b00000)
                begin //bltz
                    bltz=1;
                end
            else if(rt== 5'b00001)
                begin //bgez
                    bgez=1;
                end
            else if(rt== 5'b10000)
                begin //bltzal
                    bltzal=1;
                end
            else if(rt== 5'b10001)
                begin //bgezal
                    bgezal=1;
                end
        end

        //beq 操作码 000100
        6'b000100: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 0;
```

```
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0011;//减法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=1;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //bne 操作码 000101
        6'b000101: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0011;//减法
            jmp =0;
            jal=0;
            jr=0;
            bne=1;
            beq=0;
```

```
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //blez 操作码 000110
        6'b000110: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 1;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0011;//减法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=1;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
```

```
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //bgtz 操作码 000111
        6'b000111: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 1;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0011;//减法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=1;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=1;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        //halt 操作码 111111
        6'b111111: begin
            regDst = 0;
            aluSrc = 0;
            aluZeroinput = 0;
```

```
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            ExtOp = 1;
            aluop = 4'b0011;//减法
            jmp =0;
            jal=0;
            jr=0;
            bne=0;
            beq=0;
            bgez=0;
            bgezal=0;
            bgtz=0;
            blez=0;
            bltz=0;
            bltzal=0;
            mult=0;
            div=0;
            mflo=0;
            mfhi=0;
            PCWrite=0;
            syscall=0;
            noop=0;
            halt=0;
            storemux=opCode[1:0];
        end

        default: begin
            regDst = 0;
            aluSrc = 0;
            memToReg = 0;
            regWrite = 0;
            memRead = 0;
            memWrite = 0;
            aluop = 3'b0xxx;
            jmp = 0;
            ExtOp = 0;
        end // 默认设置
    endcase
    end
endmodule
```

四、符号扩展模块signext

负责将16位的输入通过extop信号控制转换为对应的32位信号。

实现代码如下：

```
`timescale 1ns / 1ps
module signext(
           input [15:0] inst, // 输入 16 位
           input  ExtOp,
           output [31:0] data // 输出 32 位
       );
// 根据符号补充符号位
//  如果符号位为 1，则补充 16 个 1，即 16'h ffff
// 如果符号位为 0，则补充 16 个 0
assign data= inst[15:15]&ExtOp?{16'hffff,inst}:{16'h0000,inst};

endmodule
```

五、aluinput2模块

该模块负责控制alu的aluinput2的输入数据选择，分析得知，alu的第二输入有以下几种状态，

1.是判断是否是I型指令，需要由符号扩展模块输入input2

2.判断是否是blez,bgtz等一些需要和0做比较的指令，如果是，则由ctr模块输出了aluZeroinput信号，则aluinput2需要为0

3.如果不是以上的几种状态，则aluinput2需要输出正常情况下从regfile读取的Rtdata数据

实现代码如下：

```
module aluinput2(
    input [31:0] ext_data,
    input [31:0] rt_data,
    input aluSrc,
    input aluZeroinput,
    output [31:0]alu_input2
);
    wire [31:0] muxext_data,muxzero;
    assign muxext_data=aluSrc?ext_data:rt_data;
    assign muxzero=aluZeroinput?32'h00000000:muxext_data;
    assign alu_input2=muxzero;
endmodule
```

六、aluctr模块

首先编写好alu各种功能和aluop的对应表，根据表填写代码

	H	I	J	K	L
39	R形aluop	aluop	funct	aluctr	功能
40	0000	0001	100000	00000	add
41	0000	0010	100001	00001	addu
42	0000	0011	100010	00010	sub
43	0000		100011	00011	subu
44	0000	0100	100100	00100	and
45	0000	0101	100101	00101	or
46	0000	0110	100110	00110	xor
47	0000		100111	00111	nor
48	0000	0111	101010	01000	slt
49	0000	1000	101011	01001	sltu
50		1001		01010	lui
51	0000		000000	01011	sll
52	0000		000010	01100	srl
53	0000		000100	01101	sllv
54	0000		000110	01110	srlv
55	0000		000011	01111	sra
56	0000		000111	10000	srav
57	0000		011000	10001	mult
58	0000		011010	10010	div
59					default

该模块负责读取由pc生成的控制信号和funct，生成对应的alu控制信号。

实现代码如下

```
`timescale 1ns / 1ps
module aluctr(
        input [3:0] ALUOp,
        input [5:0] funct,
        output reg [4:0] ALUCtr
       );

always @(ALUOp or funct) //  如果操作码或者功能码变化执行操作
casex({ALUOp, funct}) // 拼接操作码和功能码便于下一步的判断
    10'b0001xxxxxx: ALUCtr = 5'b00000; // add:addi lb lh lw sb sh sw
    10'b0010xxxxxx: ALUCtr = 5'b00001; // addu:addiu
    10'b0011xxxxxx: ALUCtr = 5'b00010; // sub:beq bne bltz bgez bltzal
bgezal beq bne blez
    10'b0100xxxxxx: ALUCtr = 5'b00100; // and:andi
    10'b0101xxxxxx: ALUCtr = 5'b00101; // or:ori
    10'b0110xxxxxx: ALUCtr = 5'b00110; // xor:xori
    10'b0111xxxxxx: ALUCtr = 5'b01000; // slt:slti
    10'b1000xxxxxx: ALUCtr = 5'b01001; // sltu:sltiu
    10'b1001xxxxxx: ALUCtr = 5'b01010; // //lui:lui
//R形
    10'b0000_100000: ALUCtr = 5'b00000; // add
    10'b0000_100001: ALUCtr = 5'b00001; // addu
    10'b0000_100010: ALUCtr = 5'b00010; // sub
```

```
    10'b0000_100011: ALUCtr = 5'b00011; // subu
    10'b0000_100100: ALUCtr = 5'b00100; // and
    10'b0000_100101: ALUCtr = 5'b00101; // or
    10'b0000_100110: ALUCtr = 5'b00110; // xor
    10'b0000_100111: ALUCtr = 5'b00111; // nor
    10'b0000_101010: ALUCtr = 5'b01000; // slt
    10'b0000_101011: ALUCtr = 5'b01001; // sltu
    10'b0000_000000: ALUCtr = 5'b01011; // sll
    10'b0000_000010: ALUCtr = 5'b01100; // srl
    10'b0000_000100: ALUCtr = 5'b01101; //sllv
    10'b0000_000110: ALUCtr = 5'b01110; //srlv
    10'b0000_000011: ALUCtr = 5'b01111;//sra
    10'b0000_000111: ALUCtr = 5'b10000;//srav
    10'b0000_011000: ALUCtr = 5'b10001;//mult
    10'b0000_011010: ALUCtr = 5'b10010;//div

    default:ALUCtr = 5'b11111;
endcase
endmodule
```

七、alu模块

alu模块负责根据输入的两个数据和指令中的shamt数据段和aluctr生成的控制码运算，并且输出了ZF,PF的信号，方便之后控制bltz，bgez等和零比较的控制信号的判断，为了实现乘法和除法的指令，加入的hi，lo两个运算结果寄存器。

根据下表中的aluctr，填写alu模块代码

	H	I	J	K	L
39	R形aluop	aluop	funct	aluctr	功能
40	0000	0001	100000	00000	add
41	0000	0010	100001	00001	addu
42	0000	0011	100010	00010	sub
43	0000		100011	00011	subu
44	0000	0100	100100	00100	and
45	0000	0101	100101	00101	or
46	0000	0110	100110	00110	xor
47	0000		100111	00111	nor
48	0000	0111	101010	01000	slt
49	0000	1000	101011	01001	sltu
50		1001		01010	lui
51	0000		000000	01011	sll
52	0000		000010	01100	srl
53	0000		000100	01101	sllv
54	0000		000110	01110	srlv
55	0000		000011	01111	sra
56	0000		000111	10000	srav
57	0000		011000	10001	mult
58	0000		011010	10010	div
59					default

具体实现代码如下：

(代码较长，下一个模块内容在42页面开始)

```
`timescale 1ns / 1ps
module alu(
           input [4:0] shamt,
           input [31:0] input1,
           input [31:0] input2,
           input [4:0] aluCtr,
           output reg [31:0] aluRes,
           output reg ZF, //=0 则为 1，否则为 0
           output reg CF,OF,//溢出?
           output reg PF,//正数=1，负数为 0
           output reg [31:0]hi,
           output reg [31:0]lo
       );
always @(input1 or input2 or aluCtr) // 运算数或控制码变化时操作
begin

    case(aluCtr)
        5'b00000: //  add
        begin
            aluRes = $signed(input1) + $signed(input2);
            if(aluRes==0)
                ZF=1;
```

```
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b00001: //  addu
        begin
            aluRes = input1 + input2;
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b00010: //  sub
        begin
            aluRes = $signed(input1) - $signed(input2);
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b00011: //  subu
        begin
            aluRes = input1 - input2;
            if(aluRes==0)
```

```
            ZF=1;
        else
            ZF=0;
        if($signed(aluRes)>0)
            PF=1;
        else
            PF=0;
        hi=0;
        lo=0;
    end

    5'b00100: //  and
    begin
        aluRes = input1 & input2;
        if(aluRes==0)
            ZF=1;
        else
            ZF=0;
        if($signed(aluRes)>0)
            PF=1;
        else
            PF=0;
        hi=0;
        lo=0;
    end

    5'b00101: //  or
    begin
        aluRes = input1 | input2;
        if(aluRes==0)
            ZF=1;
        else
            ZF=0;
        if($signed(aluRes)>0)
            PF=1;
        else
            PF=0;
        hi=0;
        lo=0;
    end

    5'b00101: //  or
    begin
        aluRes = input1 | input2;
```

```
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b00110: //  xor
        begin
            aluRes = ((~input1) & input2) | (input1 & (~input2));
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b00111: //  nor
        begin
            aluRes = ~(input1 | input2);
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01000: //  slt
        begin
```

```
            if($signed(input1) < $signed(input2))
                aluRes = 1;
            else
                aluRes = 0;
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01001: //  sltu
        begin
            if(input1 < input2)
                aluRes = 1;
            else
                aluRes = 0;
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01010: //  lui
        begin
            aluRes={input2[15:0],16'b0000_0000_0000_0000};
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
```

```
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01011: //  sll
        begin
            aluRes=input2 << shamt;
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01100: //  srl
        begin
            aluRes=input2 >> shamt;
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01101: //  sllv
        begin
            aluRes=input2 << input1[4:0];
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
```

```
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01110: //  srlv
        begin
            aluRes=input2 >> input1[4:0];
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b01111: //  sra
        begin
            aluRes= $signed(input2) >>> shamt;
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b10000: //  srav
        begin
            aluRes= $signed(input2) >>> input1[4:0];
            if(aluRes==0)
                ZF=1;
            else
                ZF=0;
            if($signed(aluRes)>0)
```

```
                PF=1;
            else
                PF=0;
            hi=0;
            lo=0;
        end

        5'b10001: //  mult
        begin
            {hi,lo}=input1 * input2;
            aluRes=0;
            ZF=1;
            OF=0;
            PF=0;
        end

        5'b10010: //  div
        begin
            hi=input1 / input2;
            lo=input1 % input2;
            aluRes=0;
            ZF=1;
            OF=0;
            PF=0;
        end

        default:
        begin
            aluRes = 0; ZF=1;OF=0;PF=1;
        end
    endcase
end
endmodule
```

八、dataToReg模块

dataToReg模块通过输入的jal，bgezal和PF,ZF等等信号，根据控制输出将要写入寄存器堆的数据和是否需要写入寄存器。

经过分析，要写入寄存器堆的数据和是否写入的控制可以分为以下几种情况：

1.是jal指令或者是bltzal，bgezal指令，同时满足branch跳转条件的情况下，需要将PC+4写入ra寄存器中，则此时写入寄存器堆的数据为PC+4，写入控制信号为高电平。

2.如果是lw，lh，lb指令需要将ram中的读取数据写入寄存器堆中，则此时写入寄存器堆的

数据为ram的读取数据值，写入控制信号已经由ctr模块设置为高电平，此时直接输出原始的regWrite信号即可。

3.其他情况下，写入寄存器堆的数据为ALU的运算结果，则此时写入寄存器堆的数据为ALUres。

具体实现代码如下：

```
module dataToReg(
    input memToReg,
    input jal,
    input bgezal,
    input bltzal,
    input ZF,
    input PF,
    input [31:0]PCAdd4,
    input [31:0]alures,
    input [31:0]mem_data,
    input regWrite,
    output [31:0]w_data,
    output link_reg_write
);
wire [31:0] muxmemread,muxlink;
assign muxmemread=memToReg?mem_data:alures;
assign muxlink = (jal | (bgezal & (PF | ZF) | (bltzal & (~PF & ~ZF))))?
 PCAdd4:muxmemread;
assign w_data = muxlink;
assign link_reg_write =  (jal | (bgezal & (PF | ZF) | (bltzal & (~PF &
~ZF))))?1'b1:regWrite;
endmodule
```

九、link模块

该模块负责控制寄存器堆的写入地址，寄存器堆的写入地址有以下两种情况。

1. 是jal指令或者是bltzal，bgezal指令，同时满足branch跳转条件的情况下，需要写入ra寄存器，则此时写入寄存器堆的地址为31.

2.其他情况下，写入寄存器的地址已经在top模块中根据regdst设置完成，无需改变，直接输出地址。

具体实现代码如下

```
module link(
    input jal,
    input bgezal,
    input bltzal,
```

```
    input ZF,
    input PF,
    input [4:0]reg_W_Addr,
    output [4:0]W_Addr
);
    assign W_Addr=(jal | (bgezal & (PF | ZF) | (bltzal & (~PF & ~ZF))))
?5'b11111:reg_W_Addr;
endmodule
```

十、RegFile模块

该模块实现了一个寄存器堆，在以前实现过的寄存器堆的基础上，需要添加hi，lo寄存器满足对mult，div指令的结果储存的需求，同时对于mfhi和mflo指令，需要特别判断，并根据信号直接写入到寄存区堆正常区域的地址中。

具体实现代码如下：

```
`timescale 1ns / 1ps//寄存器堆模块
module RegFile(
input Clk,//写入时钟信号
input Clr,//清零信号
input Write_Reg,//写控制信号
input mfhi,
input mflo,
input mult,
input div,
input [31:0]W_data_hi,
input [31:0]W_data_lo,
input [4:0]R_Addr_A,//A 端口读寄存器地址
input [4:0]R_Addr_B,//B 端口读寄存器地址
input [4:0]W_Addr,//写寄存器地址
input [31:0]W_Data,//写入数据
output [31:0]R_Data_A,//A 端口读出数据
output [31:0]R_Data_B//B 端口读出数据
);
reg [31:0]REG_Files[0:31];//寄存器堆本体
reg [31:0]hi;//hi 寄存器
reg [31:0]lo;//lo 寄存器
integer i;//用于遍历 NUMB 个寄存器
initial//初始化 NUMB 个寄存器，全为 0
    for(i=0;i<32;i=i+1)
        REG_Files[i]<=0;
always@(posedge Clk or posedge Clr)//时钟信号或清零信号上升沿
begin
    if(Clr)begin//清零
        for(i=0;i<32;i=i+1)
```

```
            REG_Files[i]<=0;
        lo<=0;
        hi<=0;
    end

    else//不清零,检测写控制, 高电平则写入寄存器
        if(Write_Reg)begin
            if(mfhi)//移动 hi
                REG_Files[W_Addr]=hi;
            else if(mflo)//移动 lo
                REG_Files[W_Addr]=lo;
            else if(mult || div)begin//乘除法结果写入 lo,hi 寄存器
                hi=W_data_hi;
                lo=W_data_lo;
            end
            else
                REG_Files[W_Addr]<=W_Data;//其他正常写入寄存器

        end
end         //读操作没有使能或时钟信号控制, 使用数据流建模(组合逻辑电路,读不需要时钟控制)
assign R_Data_A=REG_Files[R_Addr_A];
assign R_Data_B=REG_Files[R_Addr_B];
endmodule
```

十一、dram模块

dram模块实现了一个数据内存，根据时钟信号和写入与读取地址控制写入数据和读取数据，为了支持半字和字节操作，需要一个flag控制内存写入，在top模块中为storemux

具体实现代码如下

```
`timescale 1ns / 1ps
module dram (
    input clk,
    input memwrite,
    input reset,
    input [1:0] flag,//storemux 信号
    input [7:0] addr,
    input [31:0] write_data,
    output [31:0] read_data
);

reg [7:0] RAM[255:0];
```

```
//read
assign read_data=flag[1]? { {RAM[addr+3]},{RAM[addr+2]},{RAM[addr+1]},{
RAM[addr+0]}} : ( flag[0]?{ {16{RAM[addr+1][7]}} ,{RAM[addr+1]},{RAM[ad
dr]} }:{ {24{RAM[addr][7]}} ,RAM[addr]} );

integer i;
always @ (posedge clk,posedge reset)
begin
    if(reset)begin
        for(i = 0; i < 256; i = i + 1)
            RAM[i]=0;
    end
    else if (memwrite) begin
        if(flag==2'b00)
        begin
            RAM[addr]=write_data[7:0];
        end
        else if(flag==2'b01 )
        begin
            { {RAM[addr+1]},{RAM[addr]} }=write_data[15:0];
        end
        else if(flag==2'b11 )
        begin
            { {RAM[addr+3]},{RAM[addr+2]},{RAM[addr+1]},{RAM[addr+0]}}=
write_data;
        end
    end
end
endmodule
```

十二、top模块：

完成了以上所有的模块之后，在top模块中进行组装，只是这个top模块中还是有三条assign指令，没有完全将这三条指令放在某个模块中，还是有一点违背top模块的基本原理，我也不想再去改了。

top模块完整实现代码如下：

```
`timescale 1ns / 1ps
module top(
           input clkin,
           input reset,
           output [6:0] sm_duan,//段码
           output [3:0] sm_wei,//哪个数码管
           output [31:0] PC,
```

```
            output [31:0] aluRes,
            output [31:0] instruction
        );
// 复用器信号线

//数据存储器
wire[31:0] memreaddata;
// 指令存储器
//wire [31:0] instruction;
reg[7:0] Addr;
// CPU 控制信号线
wire reg_dst,memread, memwrite, memtoreg,alu_src,alu_zeroinput,ExtOp;
wire[3:0] aluop;
wire regwrite;
wire jmp,jal,jr,bne,beq,bgez,bgezal,bgtz,blez,bltz,bltzal;
wire mult,div,mflo,mfhi,PCWrite,syscall,noop,halt;
wire [1:0]storemux;
// ALU 控制信号线
wire ZF,OF,CF,PF; //alu 运算标志
wire[31:0] hi,lo; //alu 运算结果
// ALU 控制信号线
wire[4:0] aluCtr;//根据 aluop 和指令后 6 位 选择 alu 运算类型
//
wire[31:0] input2;
wire [15:0]data;
//link
wire [4:0] linkaddr;
//datatoreg
wire link_reg_write;

//PCctr
wire [31:0] CPCadd4;
// 寄存器信号线
wire[31:0] RsData, RtData;
wire[31:0] expand; wire[4:0] shamt;
wire [4:0]regWriteAddr;
wire[31:0]regWriteData;
//link
wire [4:0]muxlinkaddr;
assign shamt=instruction[10:6];
assign regWriteAddr = reg_dst ? instruction[15:11] : instruction[20:16]
;//写寄存器的目标寄存器来自 rt 或 rd
assign data=aluRes[15:0];
```

```
//assign regWriteData = memtoreg ? memreaddata : aluRes; //写入寄存器的数据来自 ALU 或数据寄存器
//assign input2 = alu_src ? expand : RtData; //ALU 的第二个操作数来自寄存器堆输出或指令低 16 位的符号扩展
// 例化指令存储器

rom_top trom(
            .Clk(clkin),
            .Rst(1'b0),
            .PC(PC),
            .data(instruction)
        );
// 实例化控制器模块
ctr mainctr(
        .opCode(instruction[31:26]),
        .funct(instruction[5:0]),
        .rt(instruction[20:16]),
        .regDst(reg_dst),
        .aluSrc(alu_src),
        .aluZeroinput(alu_zeroinput),
        .memToReg(memtoreg),
        .regWrite(regwrite),
        .memRead(memread),
        .memWrite(memwrite),
        .ExtOp(ExtOp),
        .aluop(aluop),
        .jmp(jmp),
        .jal(jal),
        .jr(jr),
        .bne(bne),
        .beq(beq),
        .bgez(bgez),
        .bgezal(bgezal),
        .bgtz(bgtz),
        .blez(blez),
        .bltz(bltz),
        .bltzal(bltzal),
        .mult(mult),
        .div(div),
        .mflo(mflo),
        .mfhi(mfhi),
        .PCWrite(PCWrite),
        .syscall(syscall),
        .noop(noop),
```

```
        .halt(halt),
        .storemux(storemux)
    );
aluinput2 aluinput2_i(
              .ext_data(expand),
              .rt_data(RtData),
              .aluSrc(alu_src),
              .aluZeroinput(alu_zeroinput),
              .alu_input2(input2)
          );
//  实例化 ALU 控制模块
aluctr aluctr1(
           .ALUOp(aluop),
           .funct(instruction[5:0]),
           .ALUCtr(aluCtr)
       );
//link
link link1(
         .jal(jal),
         .bgezal(bgezal),
         .bltzal(bltzal),
         .ZF(ZF),
         .PF(PF),
         .reg_W_Addr(regWriteAddr),
         .W_Addr(muxlinkaddr)
     );
PCctr PCctr1(
          .clkin(clkin),
          .reset(reset),
          .PCWrite(PCWrite),
          .target(instruction[25:0]),
          .imm_16(instruction[15:0]),
          .R_Data_A(RsData),
          .jmp(jmp),
          .jal(jal),
          .jr(jr),
          .bne(bne),
          .beq(beq),
          .bgez(bgez),
          .bgezal(bgezal),
          .bgtz(bgtz),
          .blez(blez),
          .bltz(bltz),
          .bltzal(bltzal),
```

```
            .ZF(ZF),
            .PF(PF),
            .PC(PC),
            .CPCadd4(CPCadd4)
        );
dataToReg dataToReg1(
                .memToReg(memtoreg),
                .jal(jal),
                .regWrite(regwrite),
                .bgezal(bgezal),
                .bltzal(bltzal),
                .ZF(ZF),
                .PF(PF),
                .PCAdd4(CPCadd4),
                .alures(aluRes),
                .mem_data(memreaddata),
                .w_data(regWriteData),
                .link_reg_write(link_reg_write)
            );
// 。。。。。。。。。。。。。。。。。。。。。。。。。实例化寄存器模块
RegFile regfile(
              .Clk(!clkin),
              .Clr(reset),
              .Write_Reg(link_reg_write),
              .R_Addr_A(instruction[25:21]),
              .R_Addr_B(instruction[20:16]),
              .W_Addr(muxlinkaddr),
              .W_Data(regWriteData),
              .mfhi(mfhi),
              .mflo(mflo),
              .mult(mult),
              .div(div),
              .W_data_hi(hi),
              .W_data_lo(lo),
              .R_Data_A(RsData),
              .R_Data_B(RtData)
          );
// 。。。。。。。。。。。。。。。。。。。。。。。。实例化 ALU 模块
alu alu(
          .shamt(shamt),
          .input1(RsData), //写入 alu 的第一个操作数必是 Rs
          .input2(input2),
          .aluCtr(aluCtr),
          .ZF(ZF),
```

```
        .OF(OF),
        .CF(CF),
        .PF(PF),
        .hi(hi),
        .lo(lo),
        .aluRes(aluRes)
    );
//实例化符号扩展模块
signext signext(
            .inst(instruction[15:0]),
            .ExtOp(ExtOp),
            .data(expand)
        );
//实例化数据存储器
dram dm(
         .clk(clkin),
         .memwrite(memwrite),
         .reset(reset),
         .flag(storemux),
         .addr(aluRes[7:0]),
         .write_data(RtData),
         .read_data(memreaddata)
     );
//.............................实例化数码管显示模块
display Smg(.clk(clkin),.sm_wei(sm_wei),.data(data),.sm_duan(sm_duan));

endmodule
```

以上CPU中所有的模块完成，下面进行仿真测试

仿真测试代码如下，通过时钟转换的控制cpu的运行。

```
module topsim;

// Inputs
reg clkin;
reg reset;
// Instantiate the Unit Under Test (UUT)
top uut (
        .clkin(clkin),
        .reset(reset)
    );

initial begin
    // Initialize Inputs
```

```
    reset=0;
    #1;
    reset = 1;
    #1;
    reset=0;
    // Wait 100 ns for global reset to finish
    #20;
end

parameter PERIOD = 40;
always begin
    clkin = 1'b0;
    #(PERIOD / 2) clkin = 1'b1;
    #(PERIOD / 2) ;
end

endmodule
```

其中用于测试CPU运行功能的指令如下：

关于测试单周期 CPU 的简单方法（修正）

（特别说明：本表每个同学都必须建立，检查实验时，必须提供！）

1、测试程序段

地址	汇编程序	指令代码						运行结果
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码		
0x00000000	addiu $1,$0,8	001001	00000	00001	0000 0000 0000 1000	=	24010008	$1=8
0x00000004	ori $2,$0,2	001101	00000	00010	0000 0000 0000 0010		34020002	$2=2
0x00000008	add $3,$2,$1	000000	00010	00001	00011 00000 100000		00411820	$3=10
0x0000000C	sub $5,$3,$2	000000	00011	00010	00101 00000 100010		00622822	$5=8
0x00000010	and $4,$5,$2	000000	00101	00010	00100 00000 100100		00a22024	$4=0
0x00000014	or $8,$4,$2	000000	00100	00010	01000 00000 100101		00824025	$8=2
0x00000018	sll $8,$8,1	000000	00000	01000	01000 00001 000000		00084040	$8=4,=8
0x0000001C	**bne $8,$1,-2 (≠,转 18)**	000101	01000	00001	1111111111111110		1501fffe	跳转 18,不跳转
0x00000020	slti $6,$2,4	001010	00010	00110	0000000000000100		28460004	$6=1
0x00000024	slti $7,$6,0	001010	00110	00111	0000000000000000		28c70000	$7=0
0x00000028	addiu $7,$7,8	001001	00111	00111	0000000000001000		24e70008	$7=8, $7=16
0x0000002C	**beq $7,$1,-2 (=, 转 28)**	000100	00111	00001	1111111111111110		10e1fffe	跳转 28，不跳转
0x00000030	sw $2,4($1)	101011	00001	00010	0000000000000100		ac220004	M[12:15]=2

0x00000034	lw $9,4($1)	100011	00001	01001	0000000000000100		8c290004	$9=2
0x00000038	addiu $10,$0,-2	001001	00000	01010	1111111111111110		240afffe	$10 = -2
0x0000003C	addiu $10,$10,1	001001	01010	01010	0000000000000001		254a0001	$10 = -1, $10 = 0
0x00000040	**bltz $10,-2(<0,转 3C)**	000001	01010	00000	1111111111111110		0540fffe	转 3c,不转
0x00000044	**andi $11,$2,2**	001100	00010	01011	0000000000000010		304b0002	$11 = 2
0x00000048	**j 0x00000050**	000010	00000	00000	0000000000010100		08000014	跳转 50
0x0000004C	or $8,$4,$2	000000	00100	00010	01000 00000 100101		00824025	跳过
0x00000050	**NOP**	000000	00000	00000	0000000000000000	=	00000000	NO OP

上一部分是实验内容总给的必须要测试的部分，下面的补充测试为为了展示实现的其他指令进行的测试。

补充测试

地址	**汇编程序**	**指令代码**						**运行结果**
		op（6）	**rs(5)**	**rt(5)**	**rd(5)/immediate (16)**	**16 进制数代码**		
0x00000054	lui $10,10	001111	00000	01010	0000000000001010		3c0a000a	$10= 10
0x00000058	mult $11,$6	000000	01011	00110	0000000000011000		01660018	hi=0 lo=2
0x0000005C	mfhi $12	000000	00000	00000	01100 00000 010000		00006010	$12=0
0x00000060	mflo $12	000000	00000	00000	01100 00000 010010		00006012	$12=2
0x00000064	div $11,$6	000000	01011	00110	00000 00000 011010		0166001a	hi=2 lo=0
0x00000068	jal 0x1b	000011	00000	00000	0000000000011011		0c00001b	Pc=6c
0x0000006C	bltzal $12 -2	000001	01100	10000	1111111111111110		0590fffe	Pc=70
0x00000070	blez $12 -3	000110	01100	00000	1111111111111101		1980fffd	Pc=74
0x00000074	sllv $13,$12,$6	000000	00110	01100	01101 00000 000100		00cc6804	$13=4
0x00000078	addi $14,$0,132	001000	00000	01110	00000 00001 010100		200e0084	$14=0x84
0x0000007C	jr $14	000000	01110	00000	00000 00000 001000		01c00008	Pc=0x84
0x00000080	nop	000000	00000	00000	0000000000000000		00000000	被跳过
0x00000084	sra $12,$12,1	000000	00000	01100	01100 00001 000011		000c6043	$12=1 $12=0
0x00000088	bgtz $12,-2	000111	01100	00000	1111111111111110		1d80fffe	跳转 84 不跳转
0x0000008C	addi $15,$0,-1	001000	00000	01111	1111111111111111		200fffff	$15=-1
0x00000090	bgez $15,-4	000001	01111	00001	1111111111111100		05e1fffc	不跳转
0x00000094	sb $15,0($s1)	101000	10001	01111	0000000000000000		a22f0000	mem[0]=-1
0x00000098	sh $15,2($s1)	101001	10001	01111	0000000000000010		a62f0002	mem[2:3]=-1
0x0000009C	lb $16,0($s1)	100000	10001	10000	0000000000000000		82300000	$16=-1，
0x000000A0	lh $16,2($s1)	100001	10001	10000	0000000000000010		86300002	$16=-1
0x000000A4	slt $16,$15,$0	000000	01111	00000	10000 00000 101010		01e0802a	$16=1
0x000000A8	sltiu $16,$15,-5	001011	01111	10000	1111111111111011		2df0fffb	$16=0
0x000000AC	sltu $16,$15,$0	000000	01111	00000	10000 00000 101011		01e0802b	$16=0
0x000000B0	addi $8,$0,8	001000	00000	01000	00000 00000 001000		20080008	$8=8
0x000000B4	srl $8,$8,1	000000	00000	01000	01000 00001 000010		00084042	$8=4

0x000000B8	srlv $8,$8,$16	000001	01000	10001	00000 00000 000000		02084006	$8=4
0x000000BC	bgezal $8,0	000000	10000	01000	01000 00000 000110		05110000	Pc+4 $31=0xC0
0x000000C0	addu $9,$8,$13	000000	01000	01101	01001 00000 100001		010d4821	$9=8
0x000000C4	subu $9,$8,$13	000000	01000	01101	01001 00000 100011		010d4823	$9=0
0x000000C8	nor $9,$8,$13	000000	01000	01101	01001 00000 100111		010d4827	$9= -5
0x000000CC	xor $9,$8,$13	000000	01000	01101	01001 00000 100110		010d4826	$9=-0
0x000000D0	srav $9,$8,$13	000000	01000	01101	01001 00000 000111		01a84807	$9= 0
0x000000D4	xori $13,$8,-1	001110	01000	01101	1111111111111111		390dffff	$13 = -5
0x000000D8	syscall	000000	00000	00000	00000 00000 001100		0000000c	Pc+4
0x000000DC	halt	111111	00000	00000	0000000000000000	=	FC000000	停机

将以上的指令的16进制代码写入rom的IP核的coe文件中，coe文件如下：

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
24010008,
34020002,
00411820,
00622822,
00a22024,
00824025,
00084040,
1501fffe,
28460004,
28c70000,
24e70008,
10e1fffe,
ac220004,
8c290004,
240afffe,
254a0001,
0540fffe,
304b0002,
08000014,
00824025,
00000000,
3c0a000a,
01660018,
00006010,
00006012,
0166001a,
0c00001b,
0590fffe,
1980fffd,
```

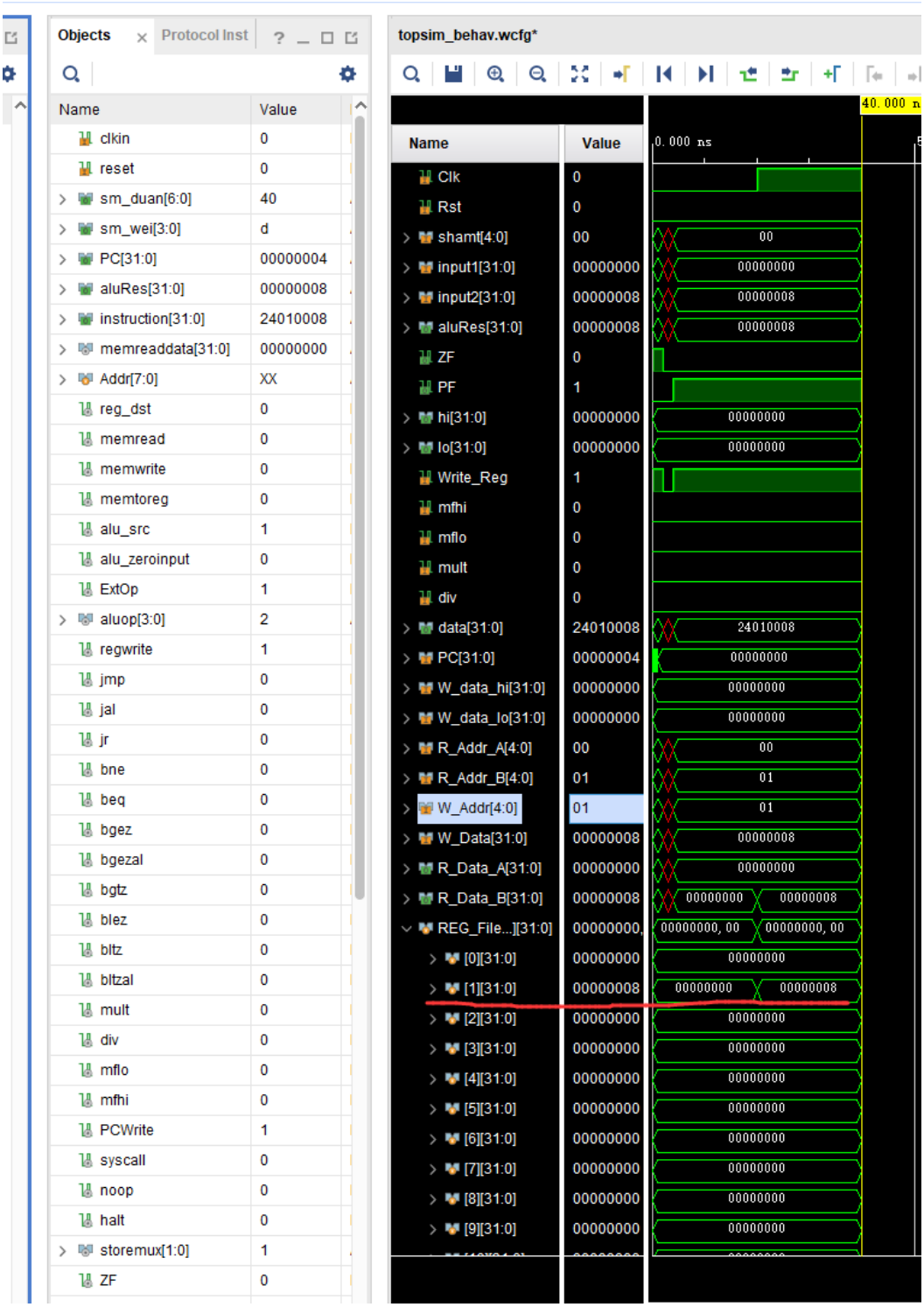
```
00cc6804,
200e0084,
01c00008,
00000000,
000c6043,
1d80fffe,
200fffff,
05e1fffc,
a22f0000,
a62f0002,
82300000,
86300002,
01e0802a,
2df0fffb,
01e0802b,
20080008,
00084042,
02084006,
05110000,
0000000c,
fc000000;
```

接下来在vivado中进行仿真测试

指令1：

地址	汇编程序	16进制代码	预期结果
0x00000000	addiu $1,$0,8	24010008	$1=8

$1=0+8=8 alures=8

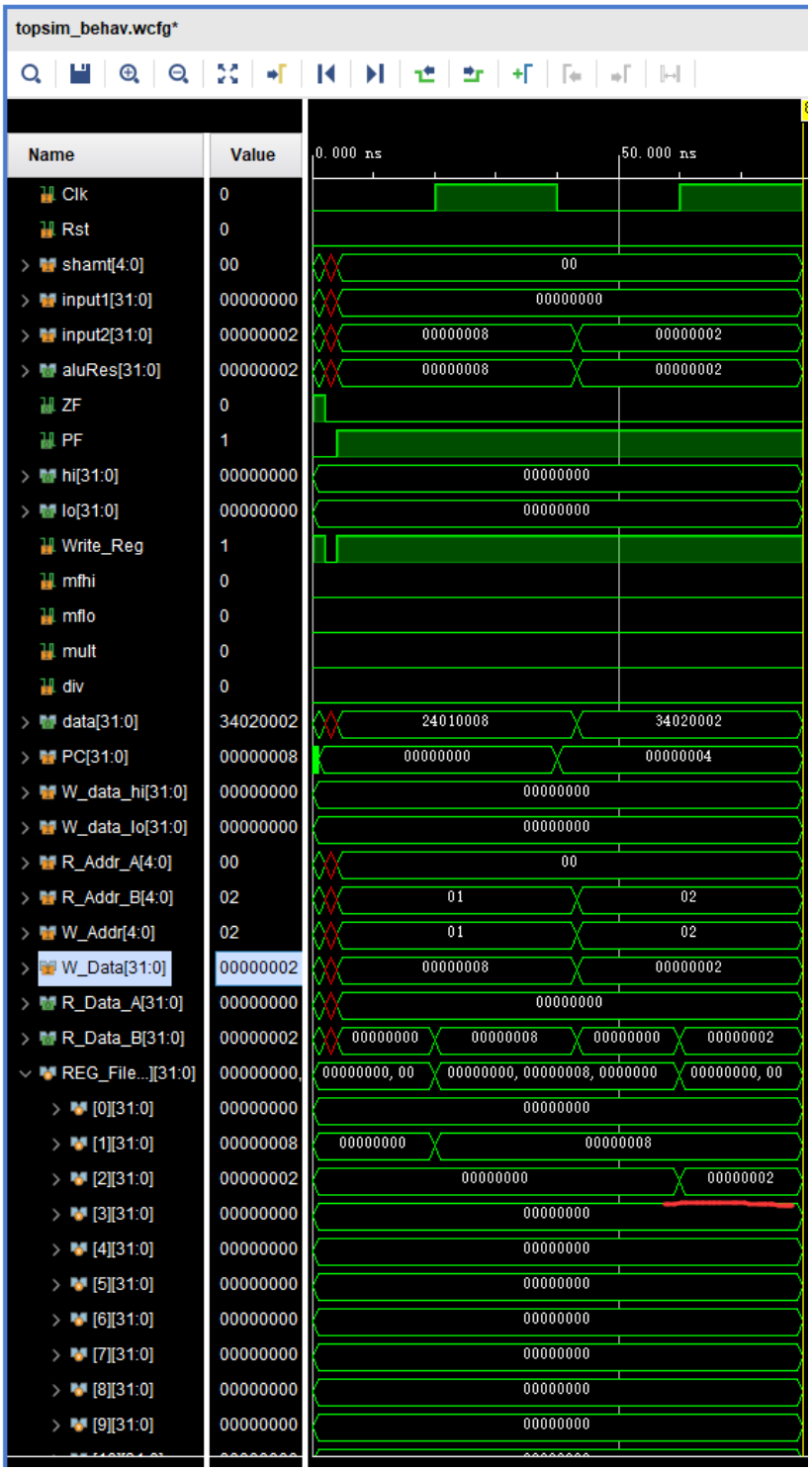


$1=8，运行正确

指令2：

地址	汇编程序	16进制代码	预期结果
0x00000004	ori $2,$0,2	34020002	$2=2

$2=0 | 2 = 2 alures=2



$2=2 运行正确

指令3：

地址	汇编程序	16进制代码	预期结果
0x00000008	add $3,$2,$1	00411820	$3=10

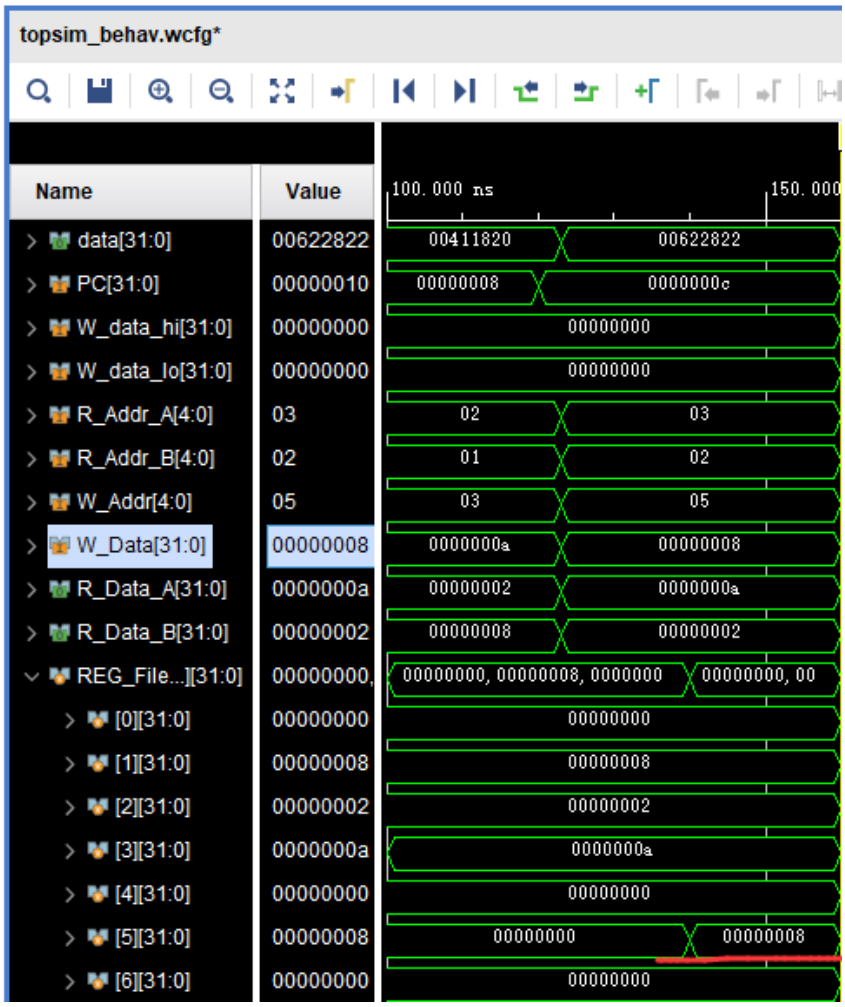
$3=2+8=0x0a alures=0x0a

$3=0x0a=10 运行正确

指令4：

地址	汇编程序	16进制代码	预期结果
0x0000000C	sub $5,$3,$2	00622822	$5=8

$5=10-2=8 alures=8

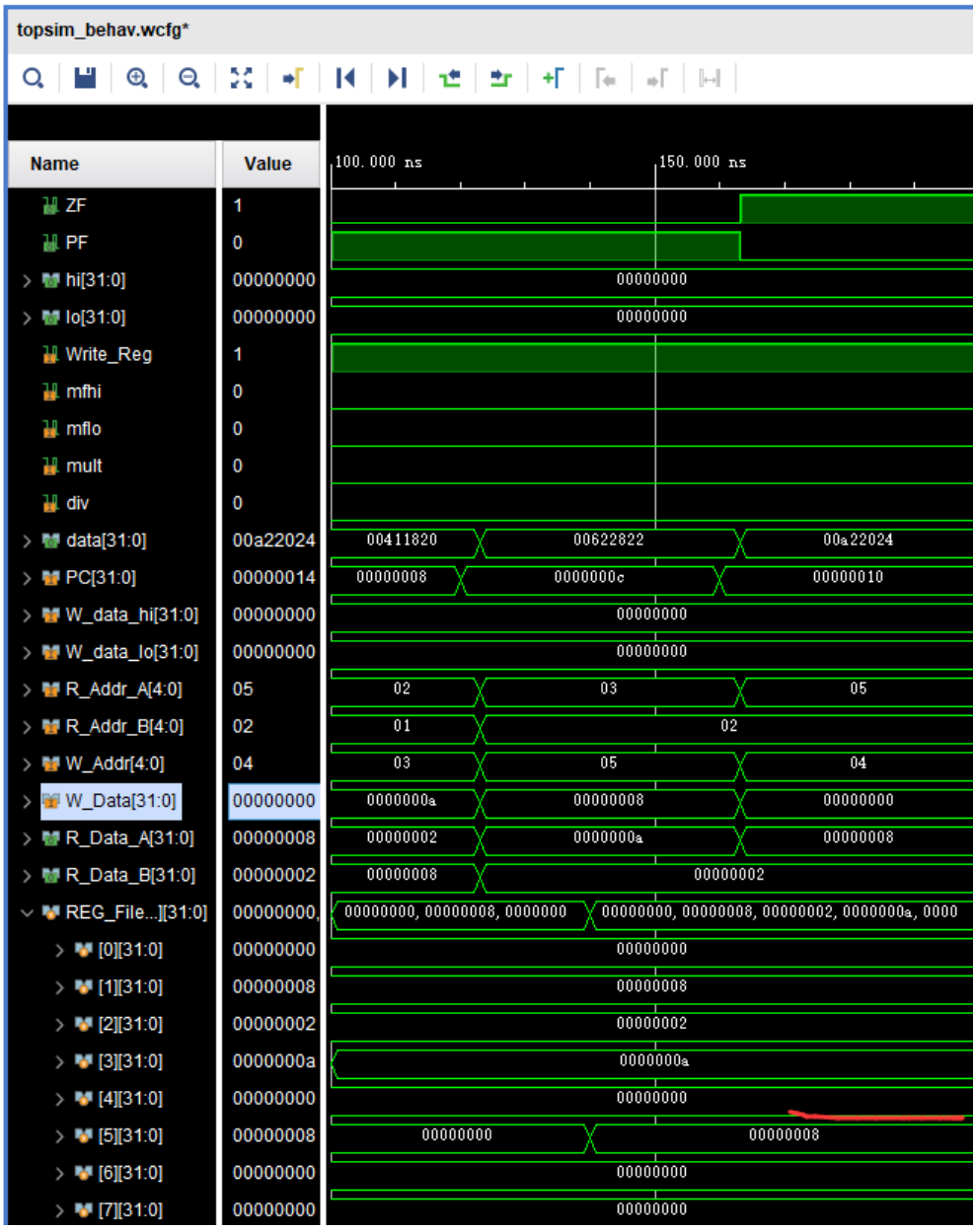


$5=8,运行正确。

指令5：

地址	汇编程序	16进制代码	预期结果
0x00000010	and $4,$5,$2	00a22024	$4=0

$4=8 & 2=0 alures=0

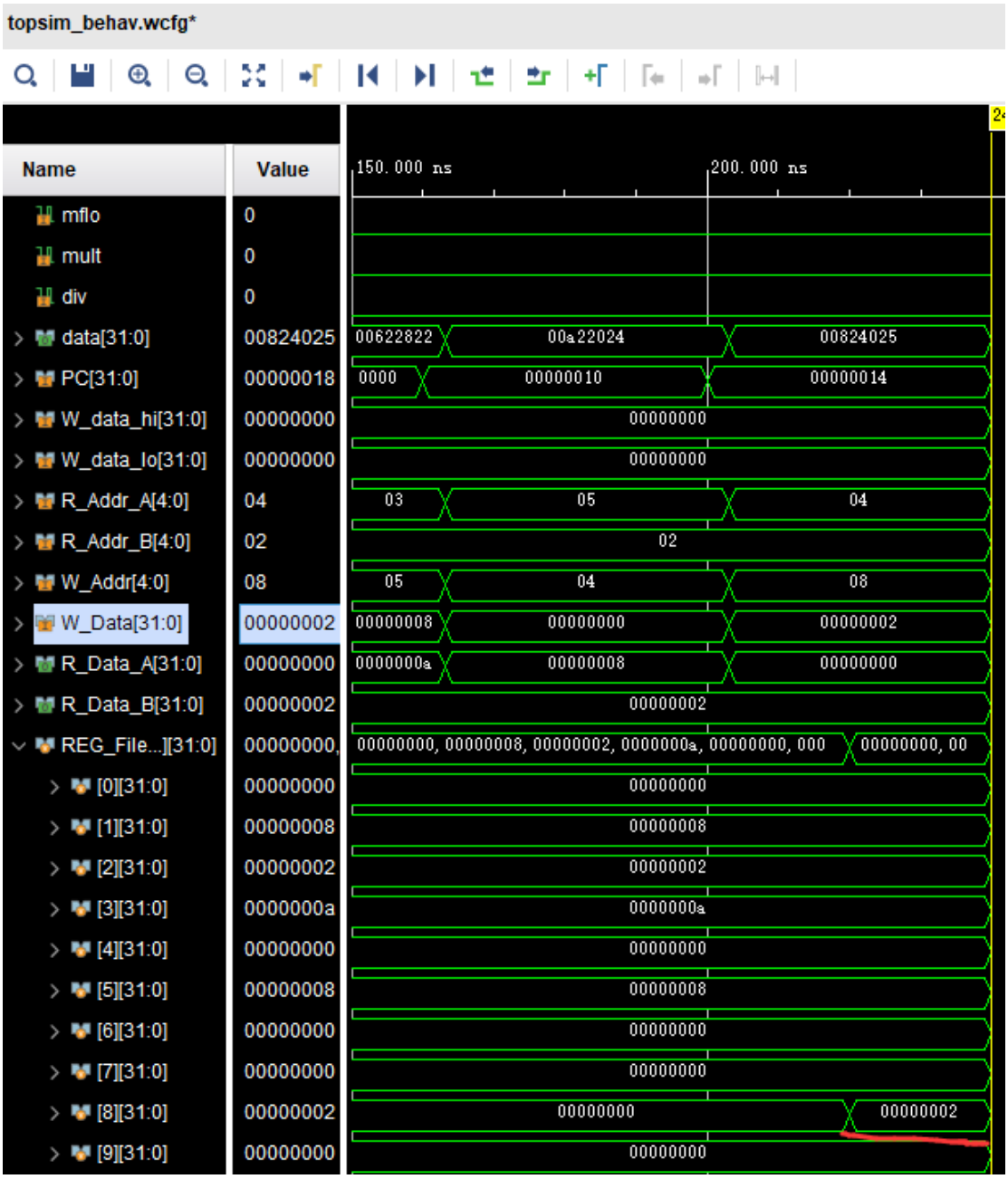


$4=0,运行正确。

指令6：

地址	汇编程序	16进制代码	预期结果
0x00000014	or $8,$4,$2	00824025	$8=2

$8=0 | 2=2 alures=2

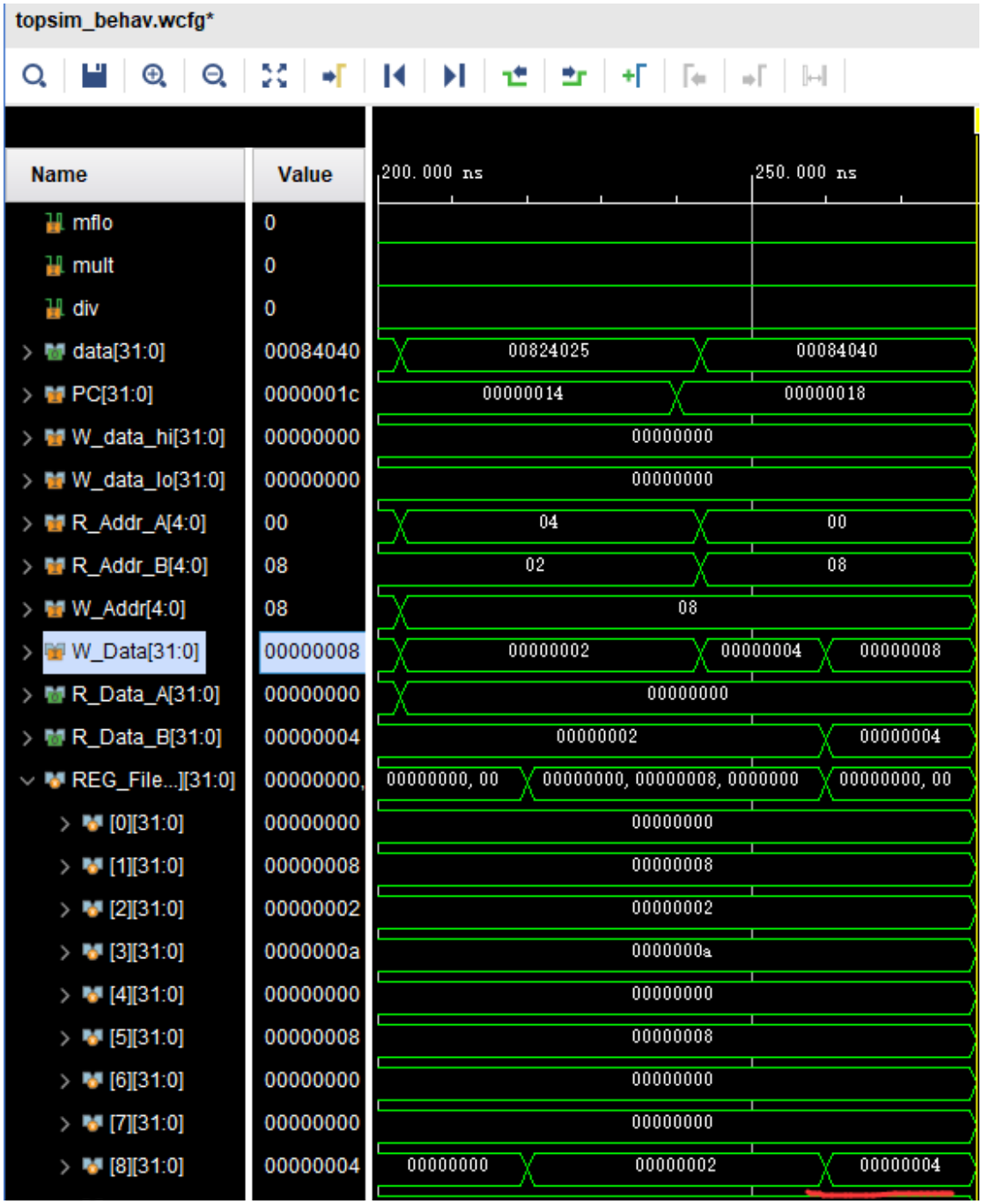


$8=2，运行正确

指令7：

地址	汇编程序	16进制代码	预期结果
0x00000018	sll $8,$8,1	00084040	$8=4

$8=2<<1=4 alures=4

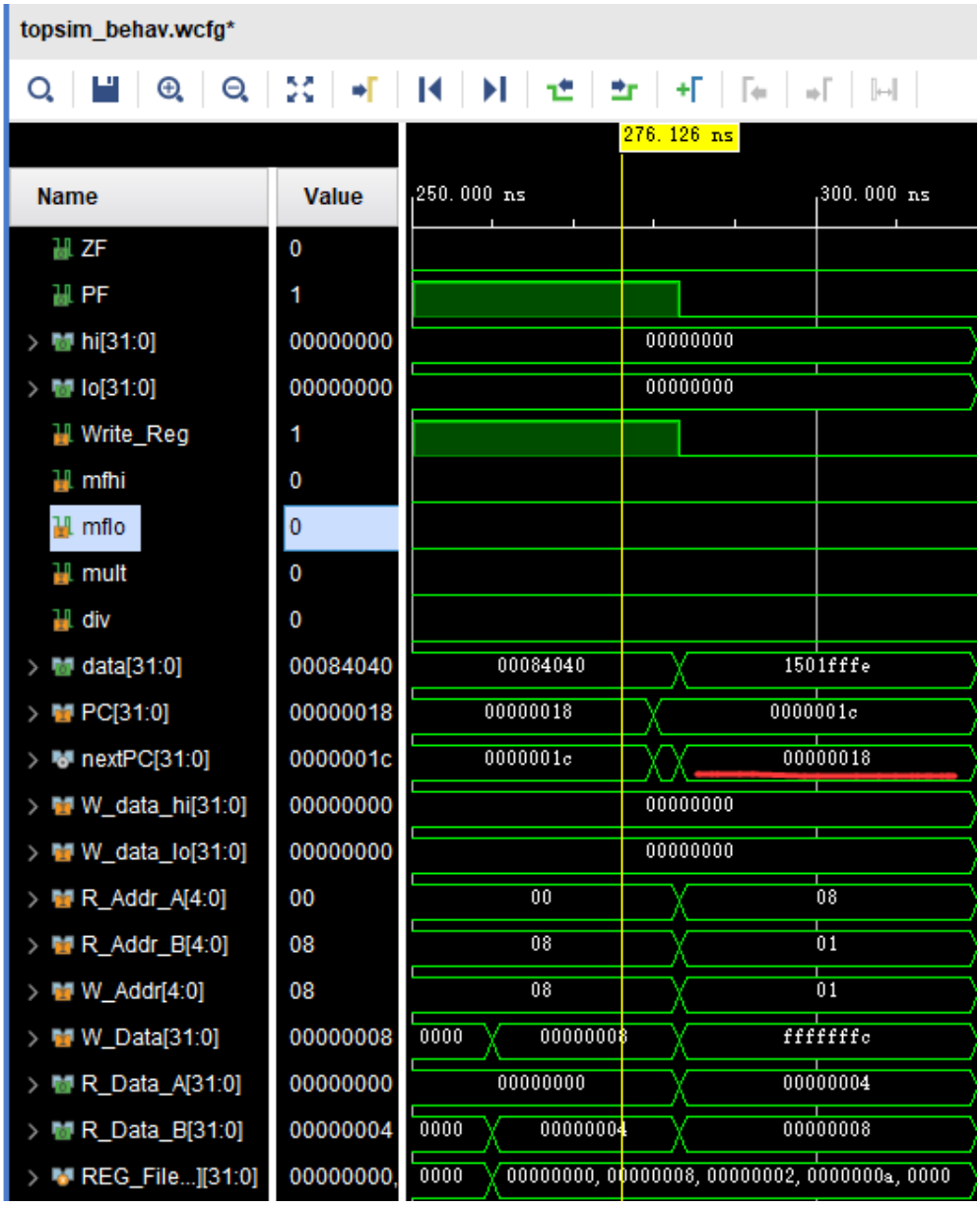


$8=4，运行正确

指令8：

地址	汇编程序	16进制代码	预期结果
0x0000001C	**bne $8,$1,-2 (≠, 转18)**	1501fffe	nextpc=0x18

$8=4 不等于 8 nextpc=0x18

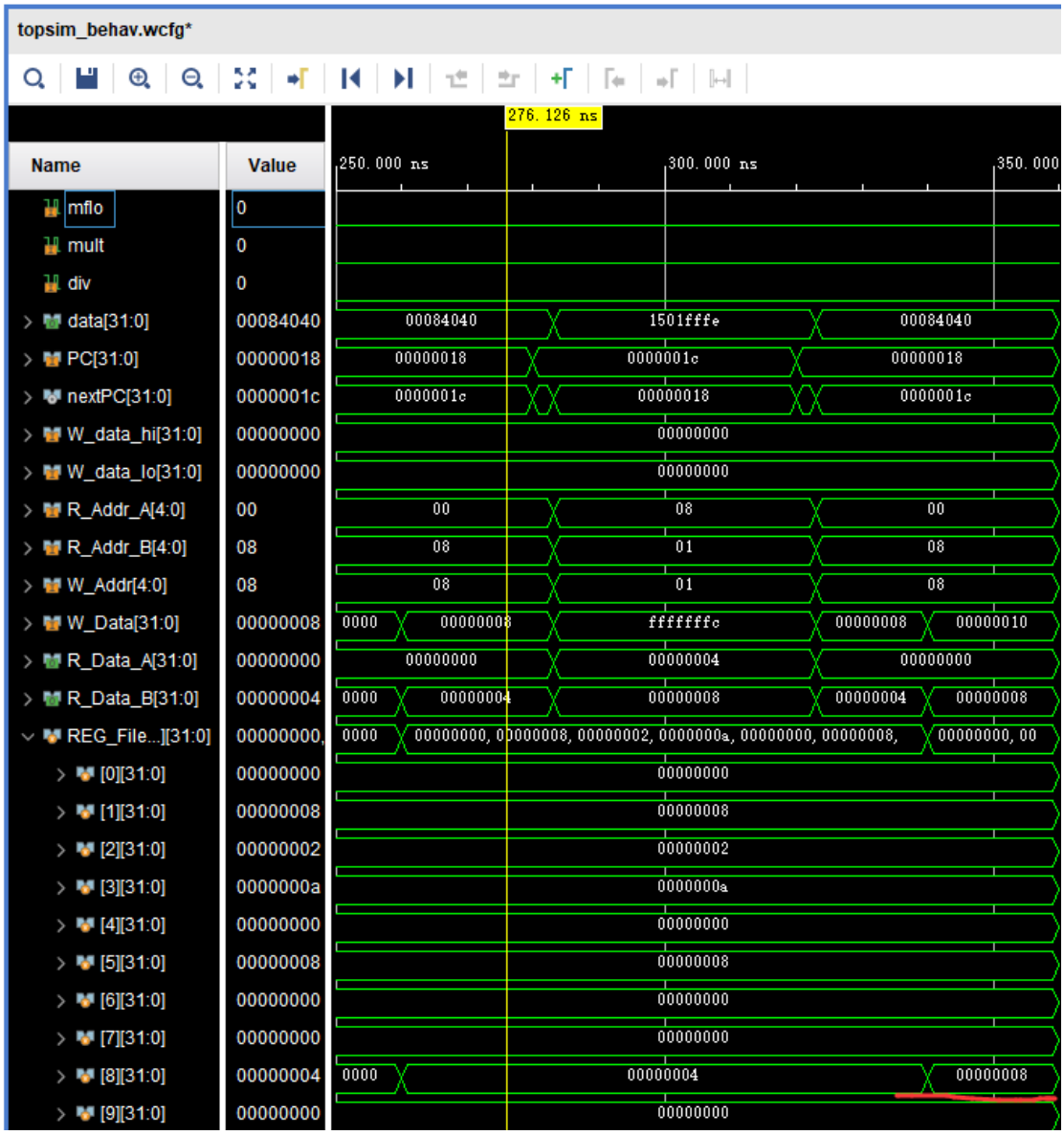


nextPC=18，运行正确

指令9：

地址	汇编程序	16进制代码	预期结果
0x00000018	sll $8,$8,1	00084040	$8=8

$8=4<<1=8 alures=8

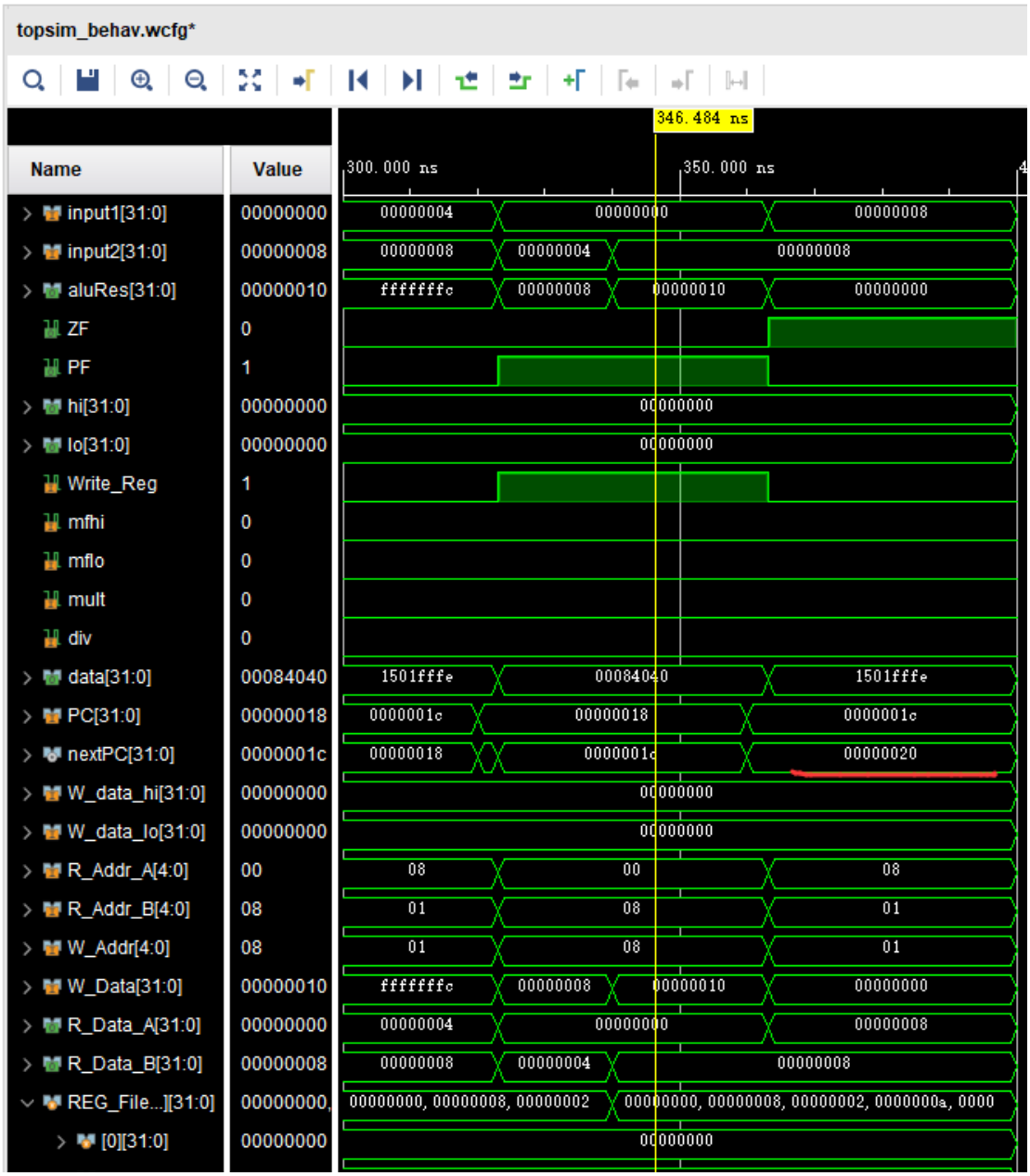


$8=8,运行正确

指令10：

地址	汇编程序	16进制代码	预期结果
0x0000001C	**bne $8,$1,-2 (≠, 转18)**	1501fffe	不跳转，nextpc=20

$8=8 等于 8 不跳转 nextpc=0x20

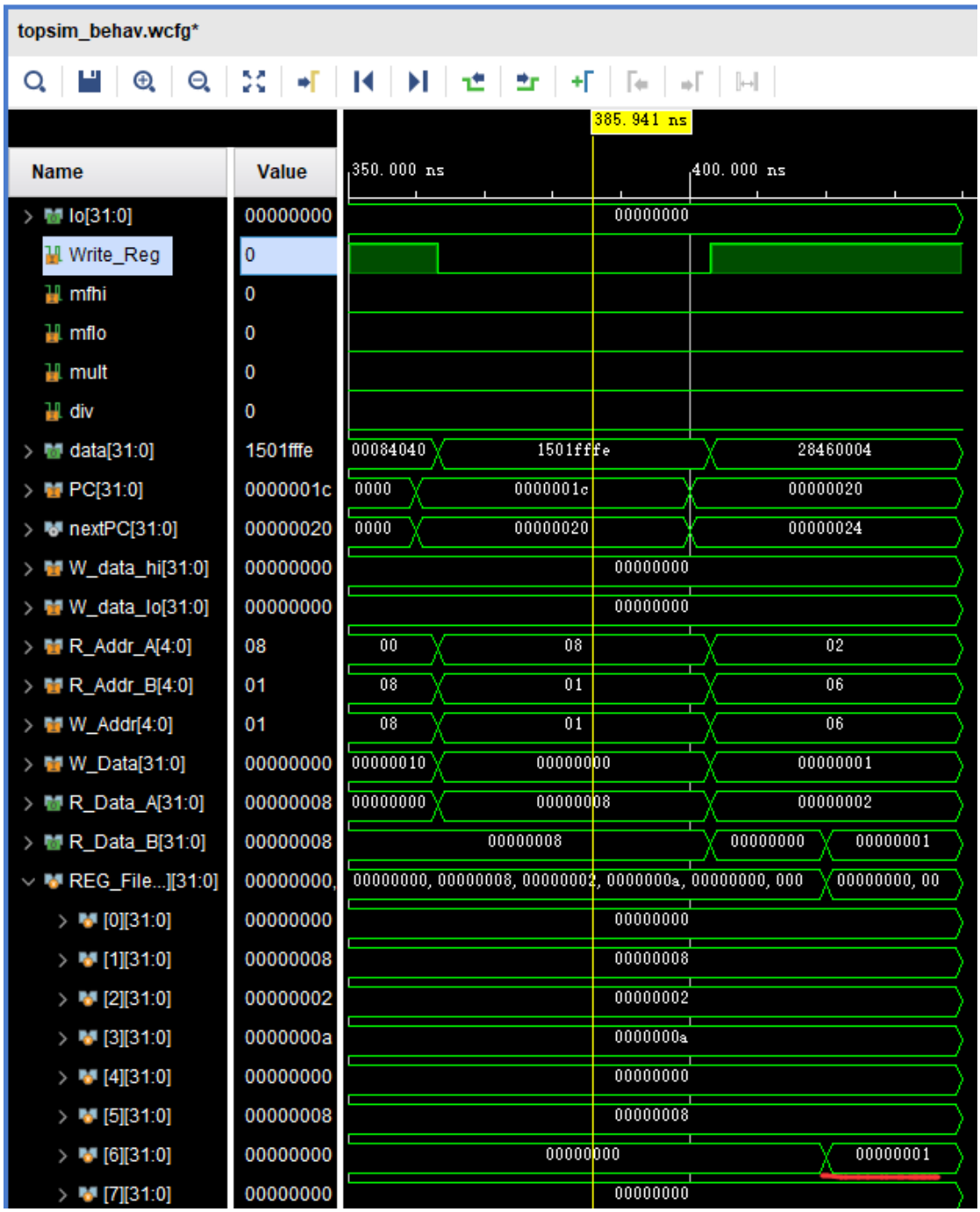


nextPC=20，运行正确

指令11：

地址	汇编程序	16进制代码	预期结果
0x00000020	slti $6,$2,4	28460004	$6=1

$2=2 < 4 alures=1 $6=1

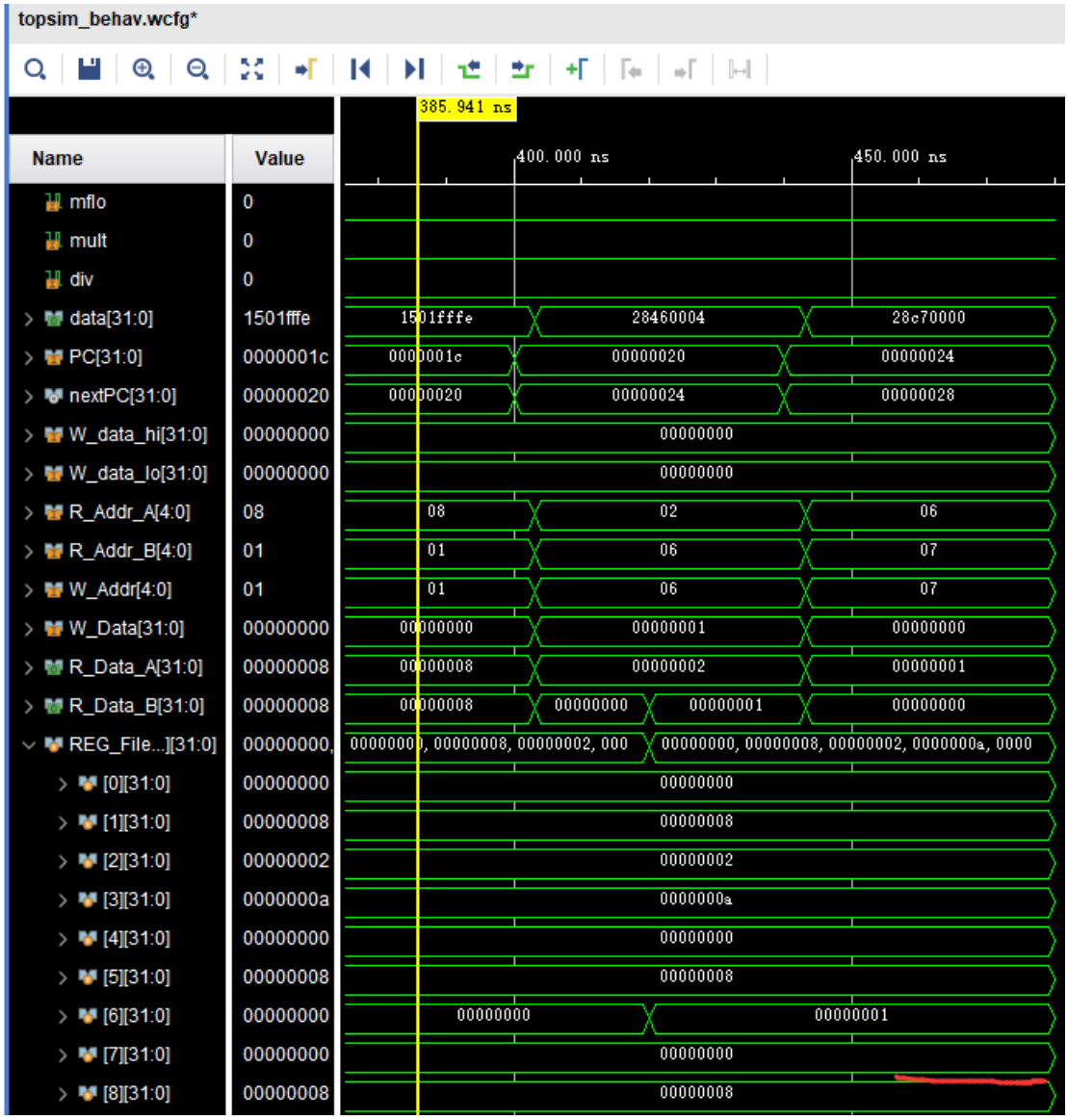


$6=1，运行正确

指令12：

地址	汇编程序	16进制代码	预期结果
0x00000024	slti $7,$6,0	28c70000	$7=0

$6=1 > 0 alures =0 $7=0

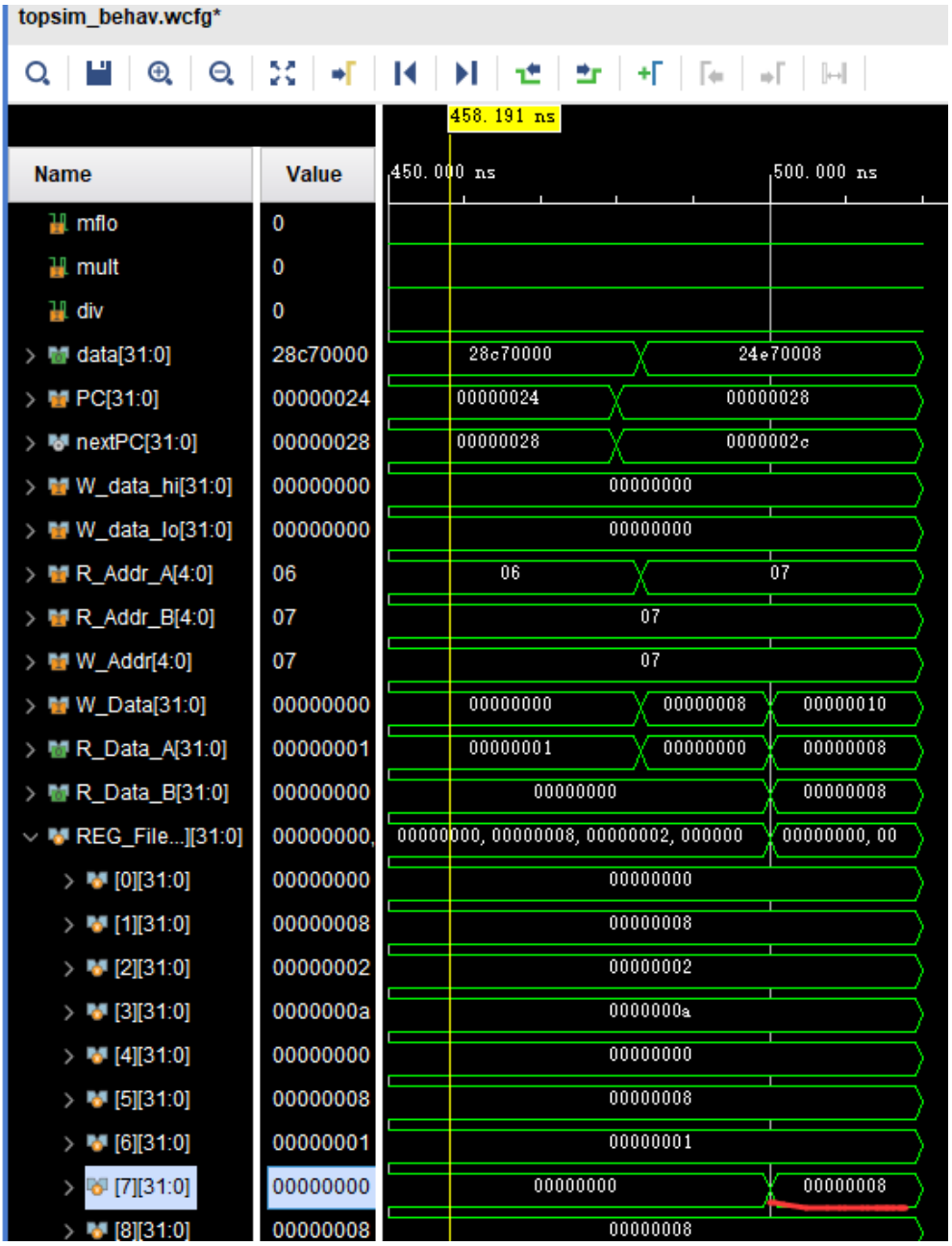


$7=0，运行正确

指令13：

地址	汇编程序	16进制代码	预期结果
0x00000028	addiu $7,$7,8	24e70008	$7=8

$7=0+8 alures =8 $7=8

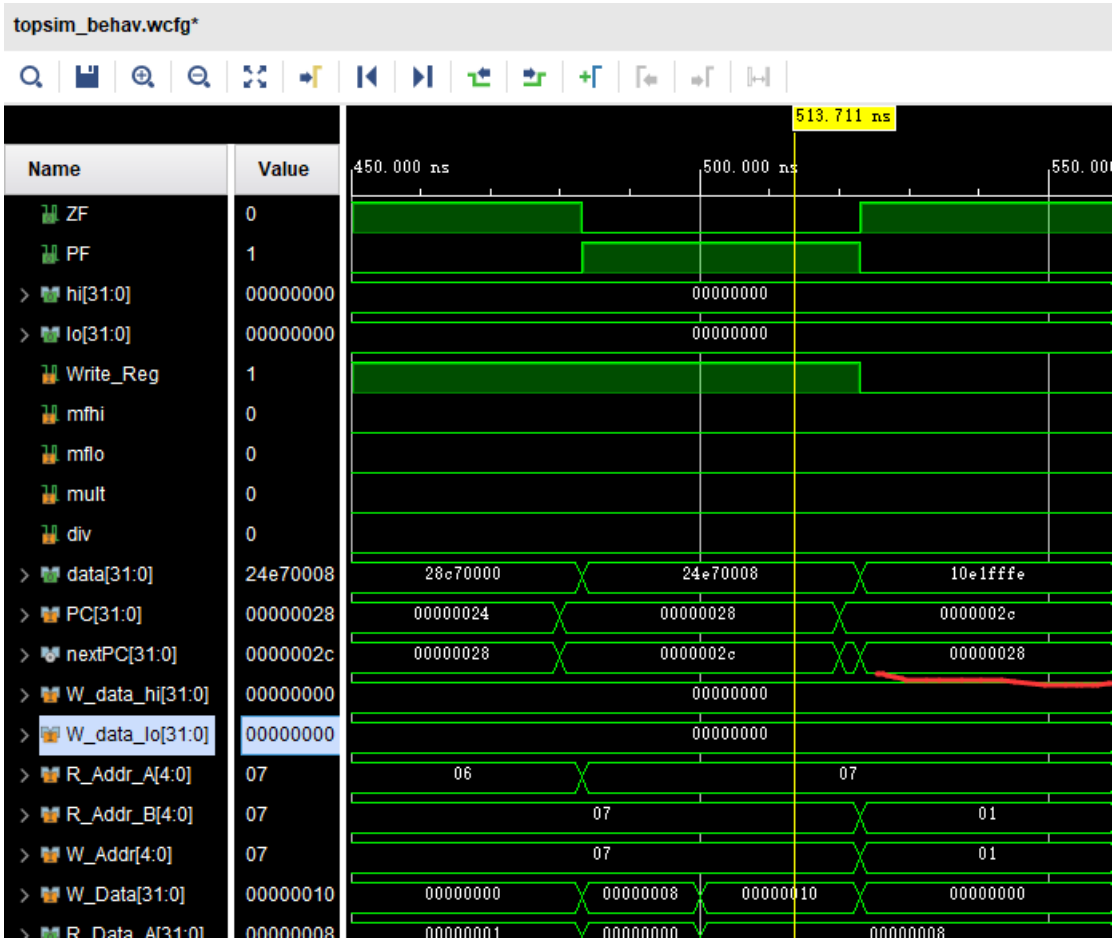


$7=8，运行正确

指令14：

地址	汇编程序	16进制代码	预期结果
0x0000002C	**beq $7,$1,-2 (=, 转28)**	10e1fffe	nextpc=28

$7=8 == $1=8　跳转，nextpc=0x28

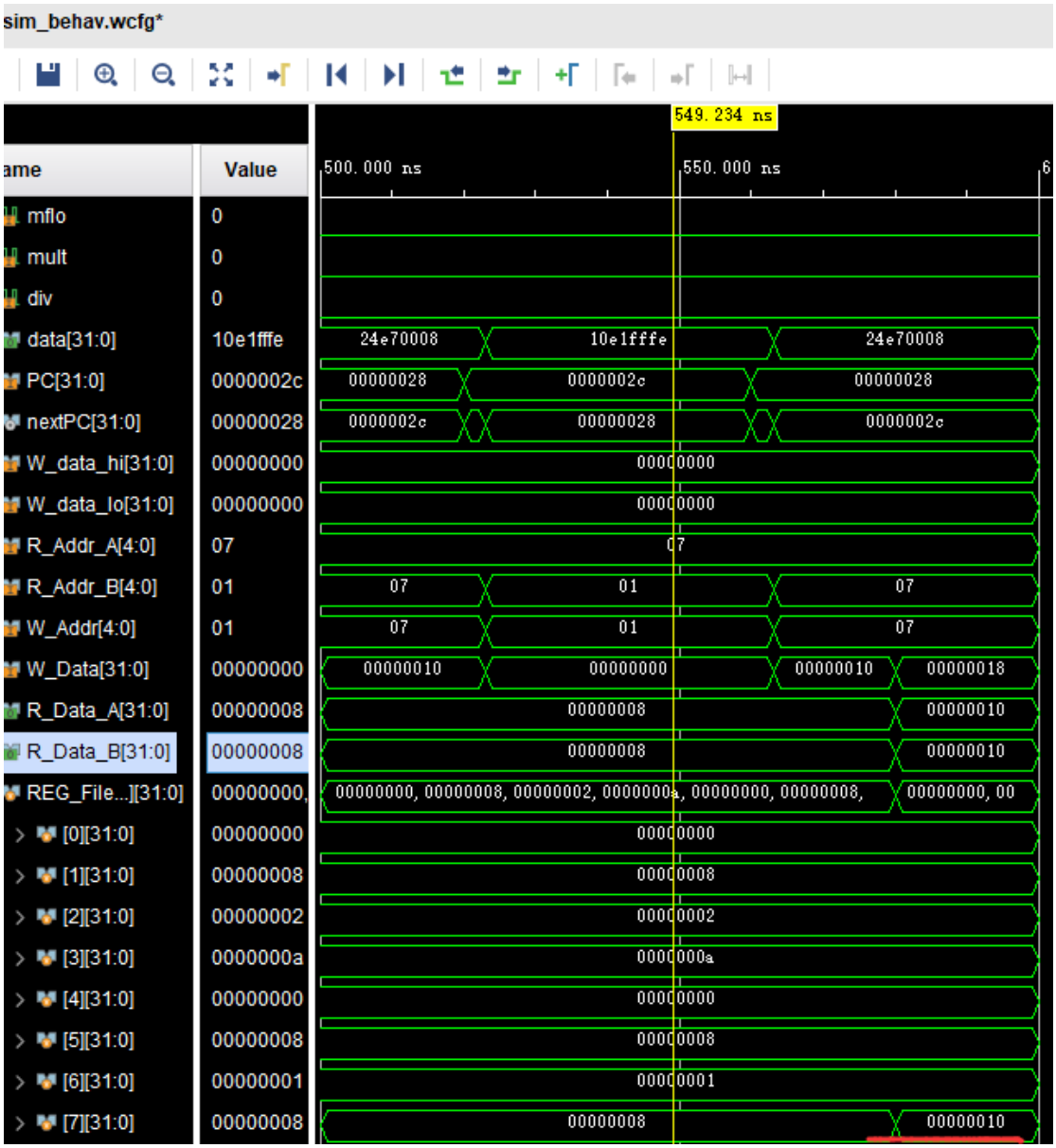


nextpc=28,运行正确

指令15：

地址	汇编程序	16进制代码	预期结果
0x00000028	addiu $7,$7,8	24e70008	$7=16

$7=8 + 8 =16　alures=16　$7=16

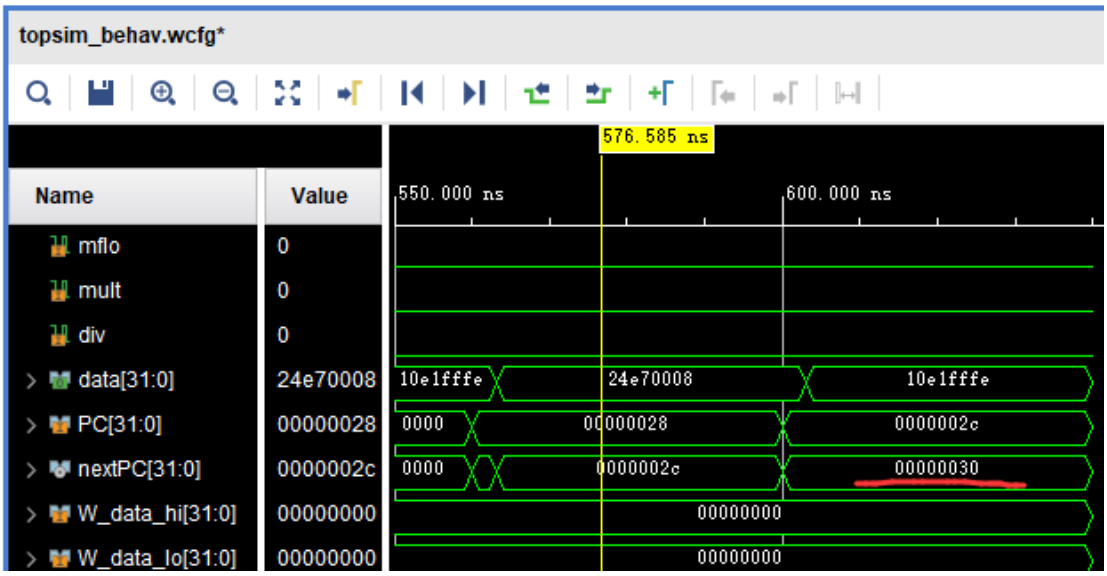


$7=0x10=16，运行正确

指令16：

地址	汇编程序	16进制代码	预期结果
0x0000002C	**beq $7,$1,-2**	10e1fffe	不跳转，nextpc=30

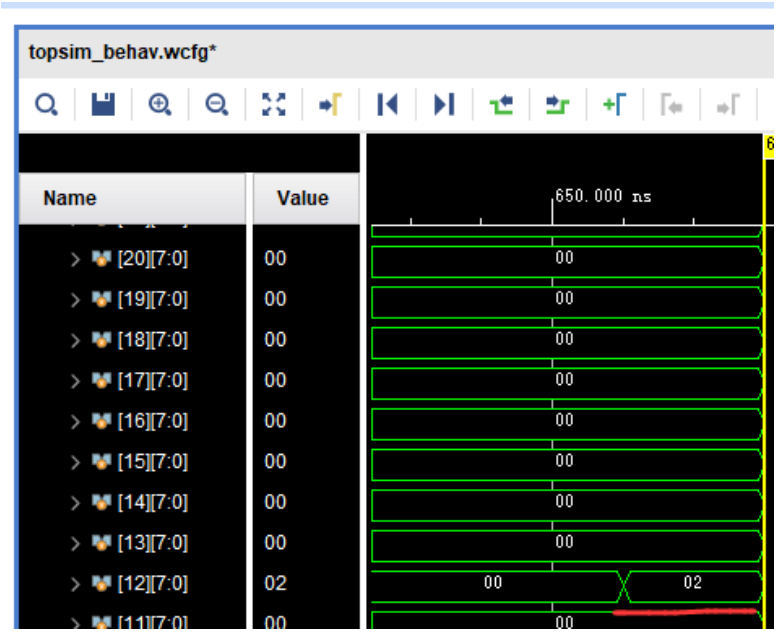
$7=16 不等于 $1=8 不跳转，nextpc=0x30



nextpc=30,运行正确

指令16：

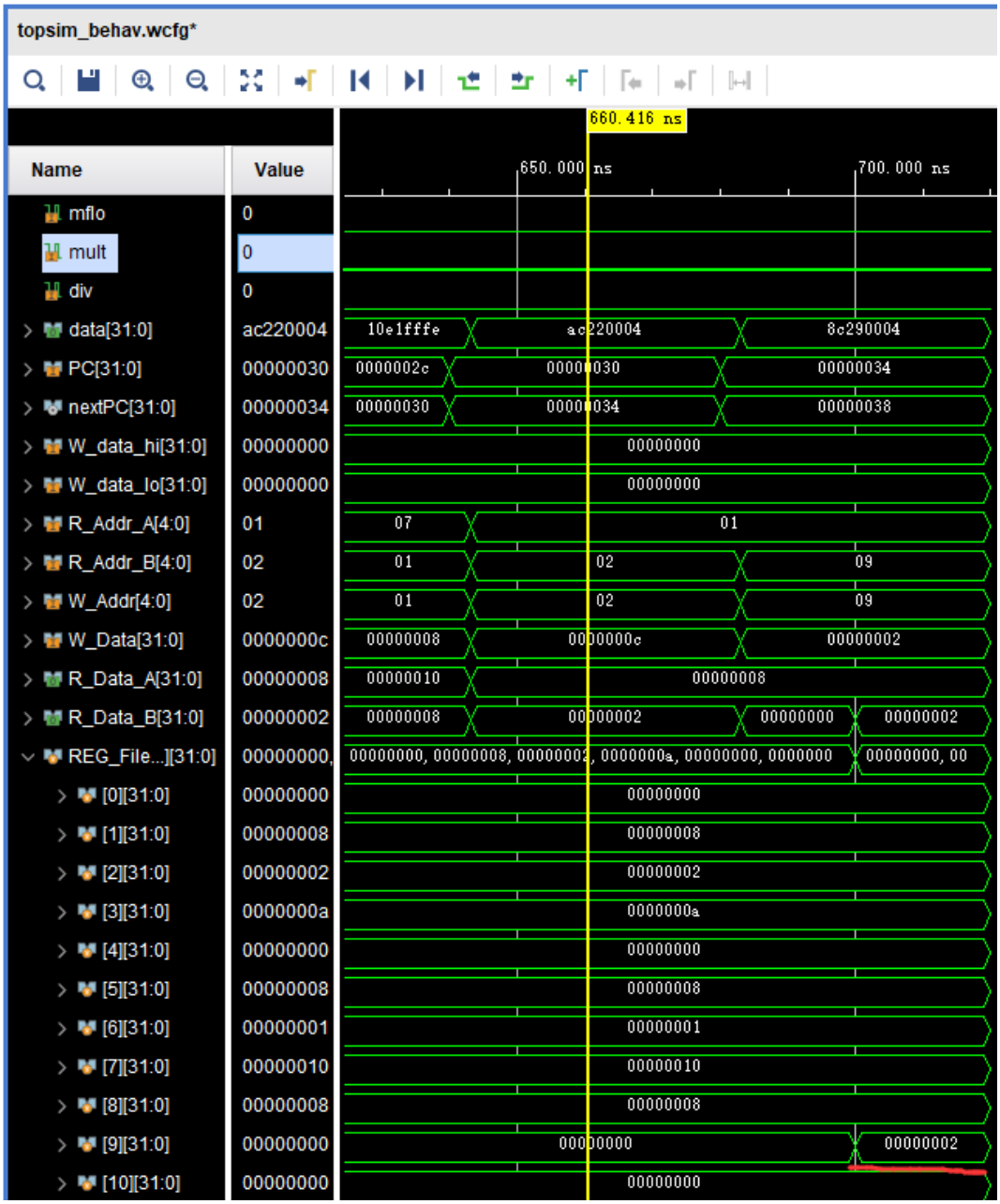
地址	汇编程序	16进制代码	预期结果
0x00000030	sw $2,4($1)	ac220004	M[12:15]=2



M[12:15]=2，运行正确

指令17：

地址	汇编程序	16进制代码	预期结果
0x00000034	lw　$9,4($1)	8c290004	$9=2

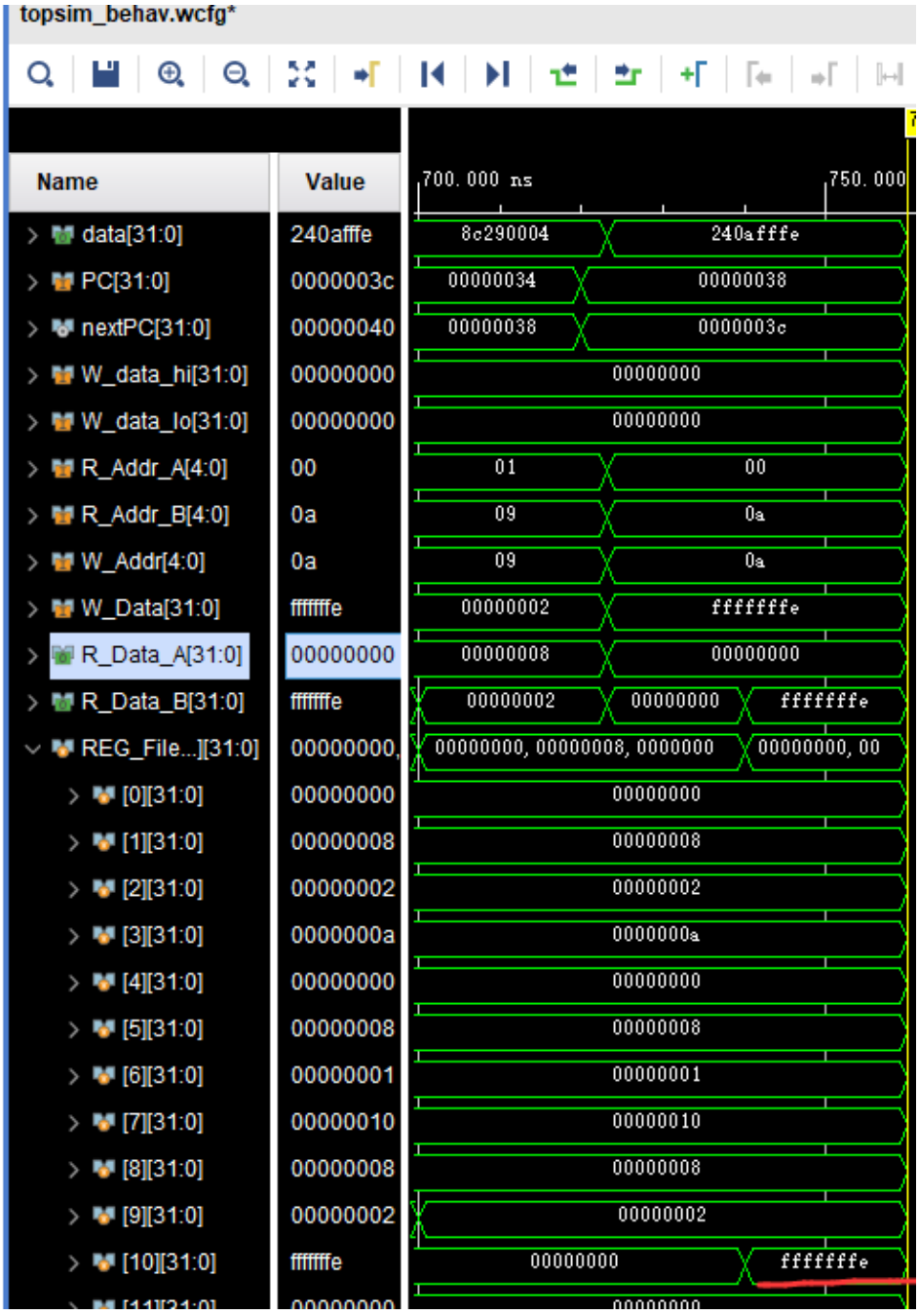


$9=2，运行正确

指令18：

地址	汇编程序	16进制代码	预期结果
0x00000038	addiu $10,$0,-2	240afffe	$10 = -2

$10 = 0 + -2 = -2 alures=-2 $10 = -2

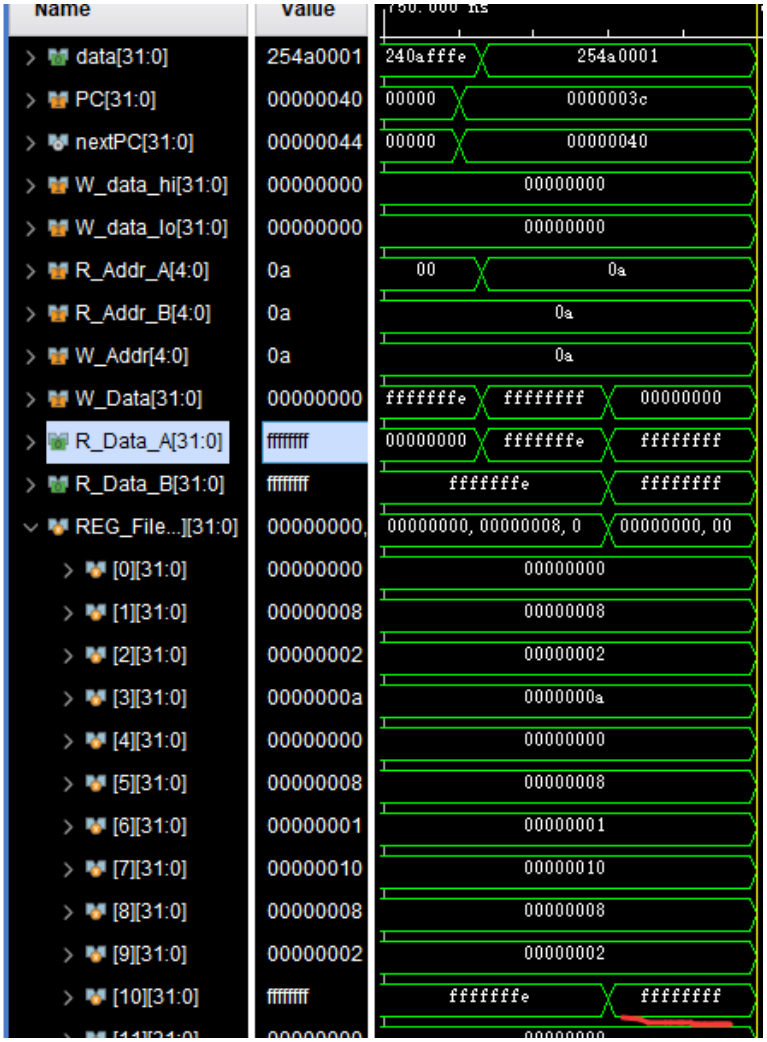


$10 = -2，运行正确

指令19：

地址	汇编程序	16进制代码	预期结果
0x0000003C	addiu $10,$10,1	254a0001	$10 = -1

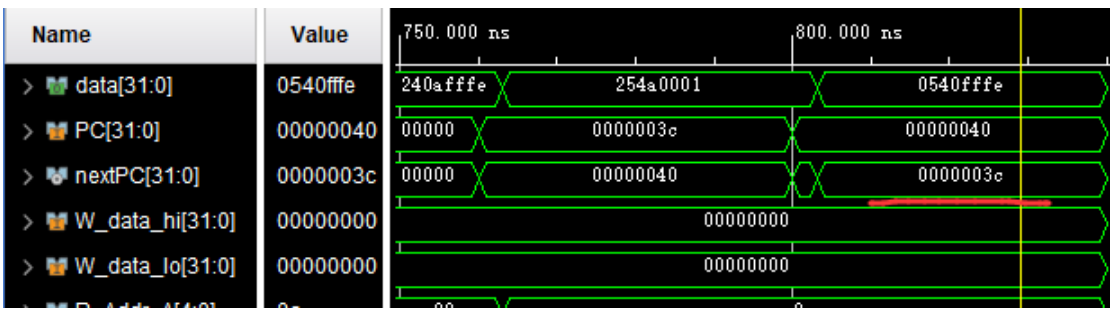
$10 = -2+1= -1 alures=--1 $10 = -1



$10 = -1，结果正确

指令20：

地址	汇编程序	16进制代码	预期结果
0x00000040	**bltz $10,-2(<0,转 3C)**	0540fffe	nextpc=0x3c



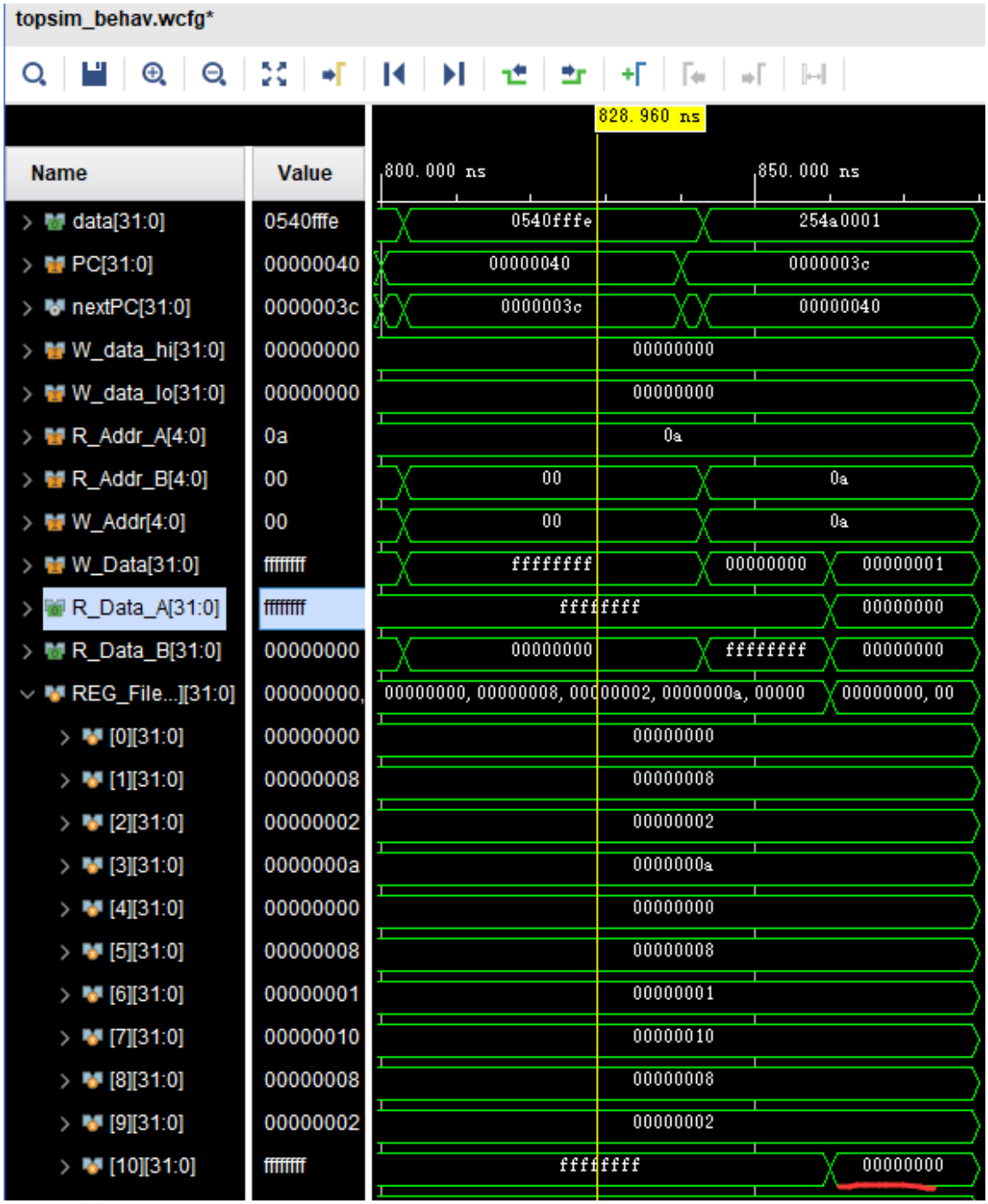
$10 < 0 跳转地址 0x3c

nextpc=3c,运行正确

指令21：

地址	汇编程序	16进制代码	预期结果
0x0000003C	addiu $10,$10,1	254a0001	$10 = 0

$10 = -1+1= 0 alures=-0 $10 = 0

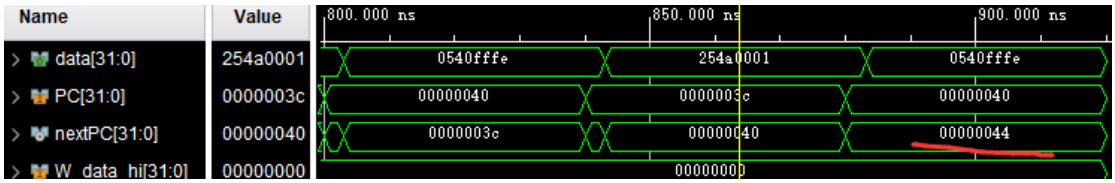


$10 = 0，运行正确。

指令22：

地址	汇编程序	16进制代码	预期结果
0x00000040	**bltz $10,-2**	0540fffe	nextpc=44

$10 不小于 0 跳转地址 0x3c

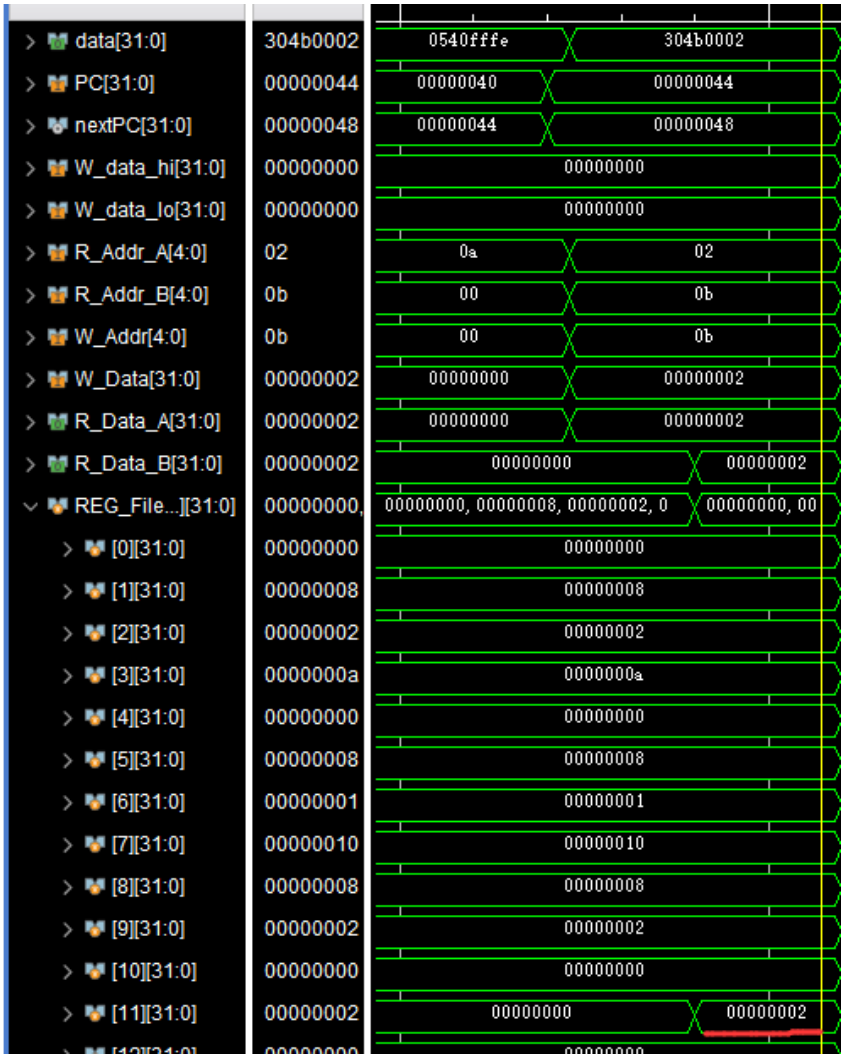


nextpc=44,运行正确

指令23：

地址	汇编程序	16进制代码	预期结果
0x00000044	**andi $11,$2,2**	304b0002	$11 = 2

$11 = 0+2= 2 alures=2 $11 = 2

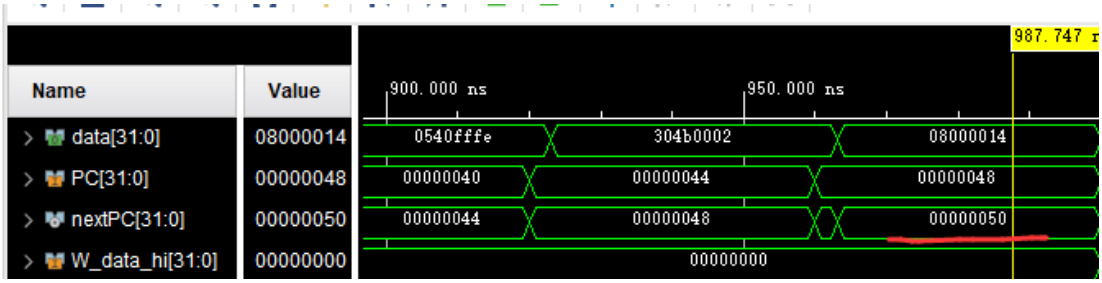


$11 = 2，运行正确

指令24：

地址	汇编程序	16进制代码	预期结果
0x00000048	**j　0x00000050**	08000014	nextpc=50

无条件跳转 nextpc=50



nextpc=50,运行正确

指令25：

地址	汇编程序	16进制代码	预期结果
0x0000004C	or　$8,$4,$2	00824025	该条指令被跳过

指令被跳过

指令26：

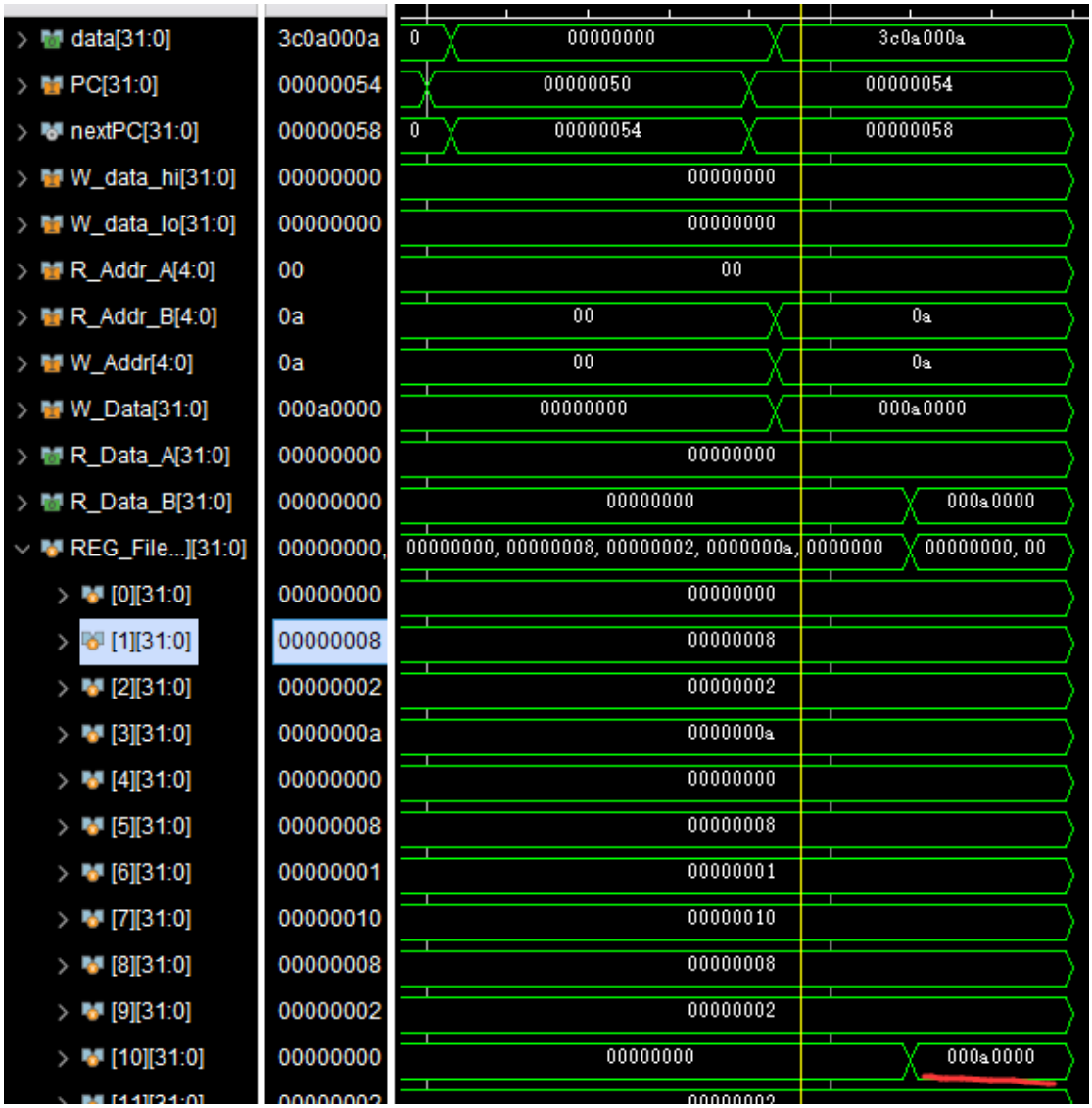
地址	汇编程序	16进制代码	预期结果
0x00000050	**NOP**	00000000	无

以上是老师给的表里的所有程序，除了将pc=50的halt指令改为nop指令以外，均测试完毕，下面为为了测试自己实现的扩展指令所设计的指令。

指令27：

地址	汇编程序	16进制代码	预期结果
0x00000054	lui $10,10	3c0a000a	$10= 0x000a0000

$10=0x0a<<16 $10= 0x000a0000

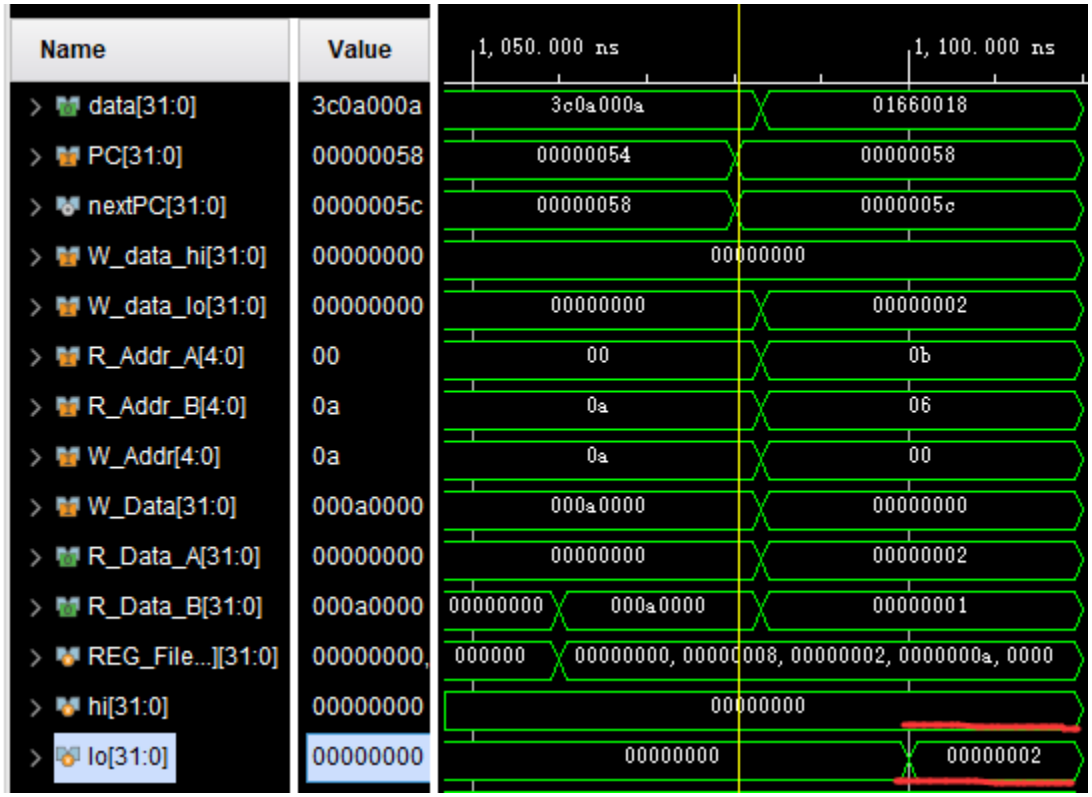


$10= 0x000a0000，运行正确

指令28：

地址	汇编程序	16进制代码	预期结果
0x00000058	mult $11,$6	01660018	hi=0 lo=2

1 x 2 =2

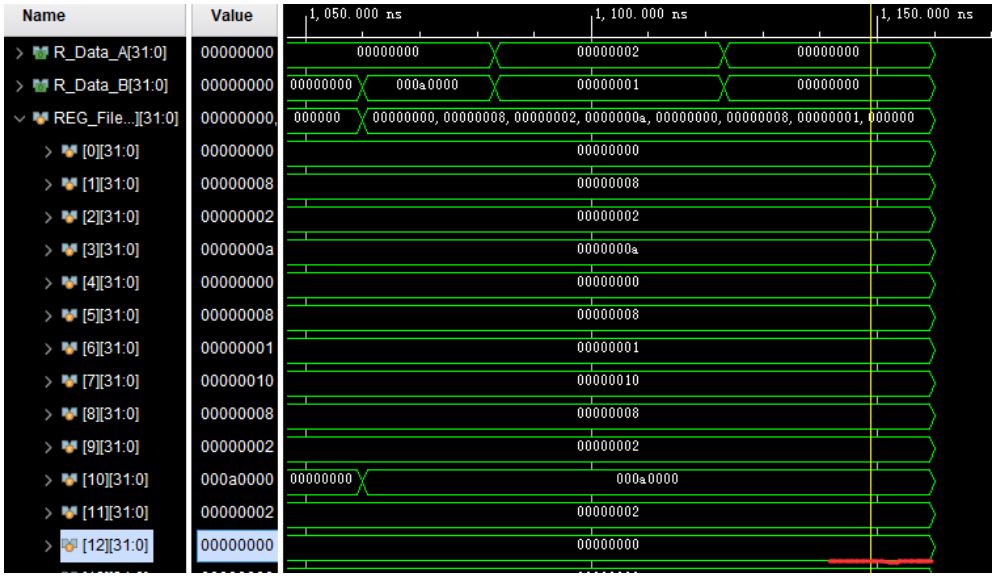


hi=0,lo=2,运行正确

指令29：

地址	汇编程序	16进制代码	预期结果
0x0000005C	mfhi $12	00006010	$12=0

hi=0, $12=0

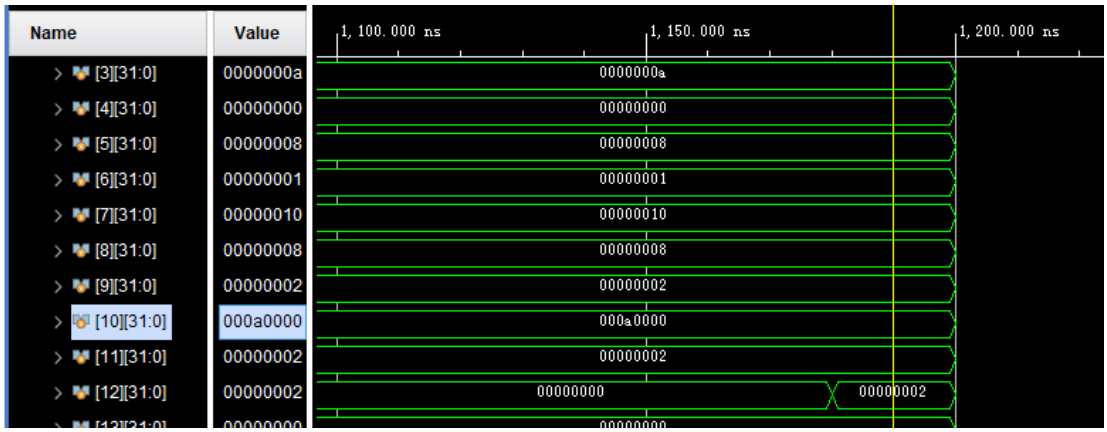


$12=0，运行正确。

指令30：

地址	汇编程序	16进制代码	预期结果
0x00000060	mflo $12	00006012	$12=2

hi=2, $12=2

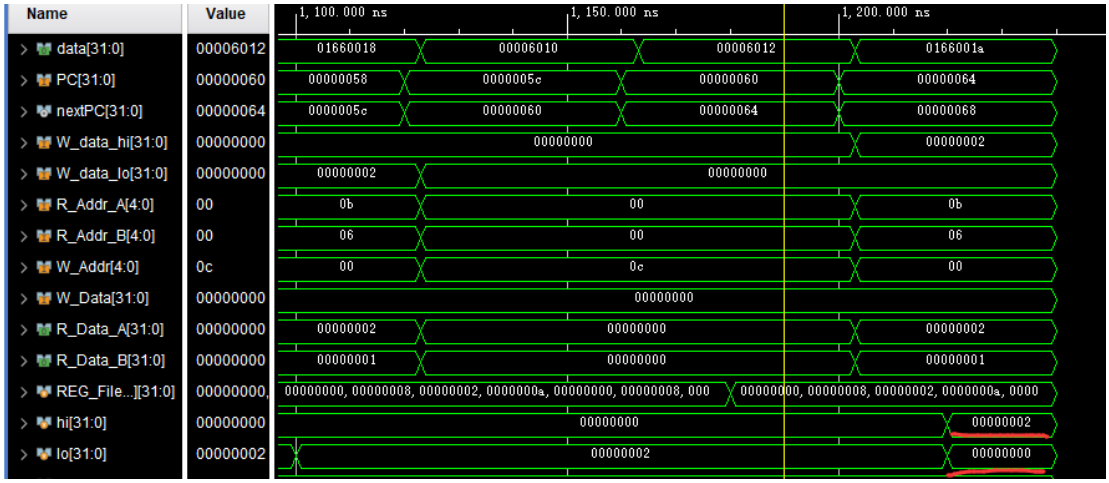


$12=2，运行正确

指令31：

地址	汇编程序	16进制代码	预期结果
0x00000064	div $11,$6	0166001a	hi=2 lo=0

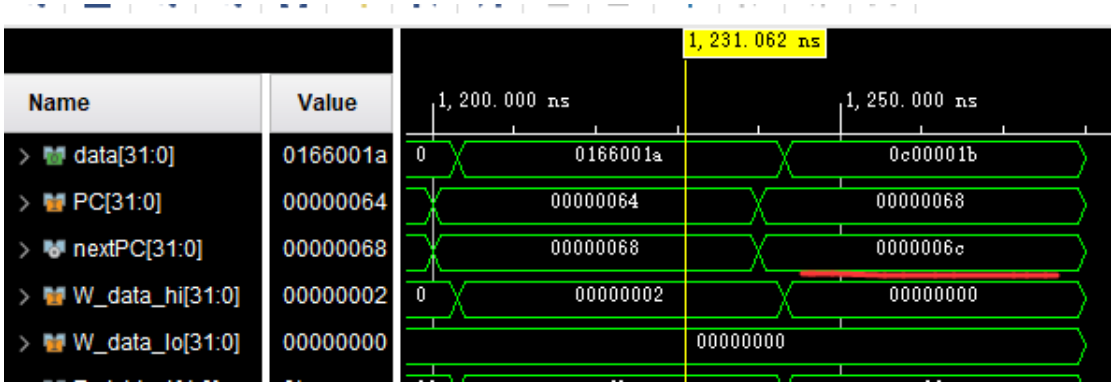
2 / 1 = 2　　2 % 1 = 0

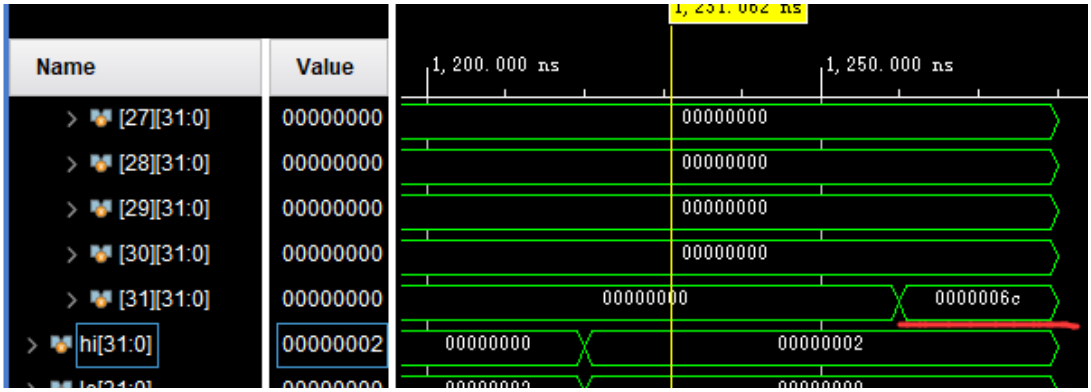


hi=2,lo=0,运行正确

指令32：

地址	汇编程序	16进制代码	预期结果
0x00000068	jal 0x1b	0c00001b	nextPc=0x6c,$31=6c



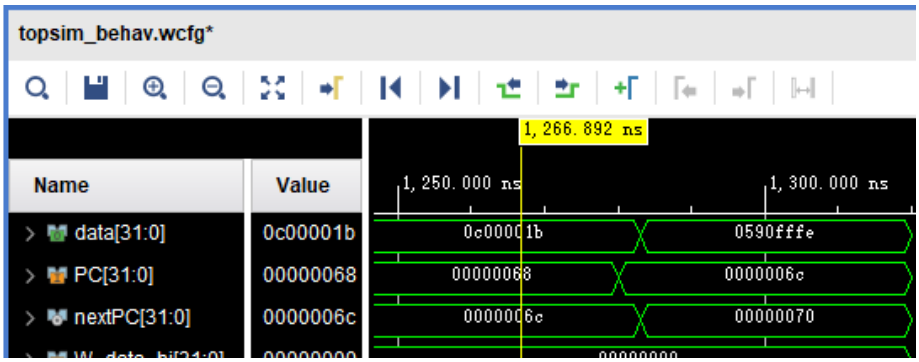


nextpc=0x6c,$ra=0x6c,运行正确。

指令33：

地址	汇编程序	16进制代码	预期结果
0x0000006C	bltzal $12 -2	0590fffe	nextPc=0x70

$12=2 不<0 不跳转

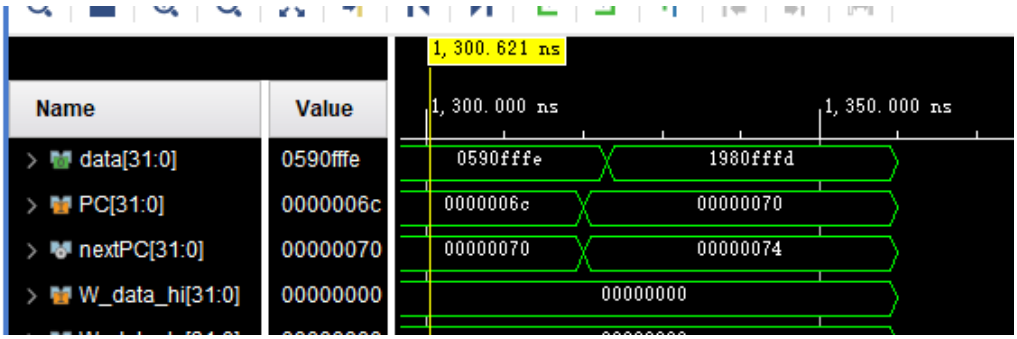


nextpc=0x70，运行正确。

指令34：

地址	汇编程序	16进制代码	预期结果
0x00000070	blez $12 -3	1980fffd	nextpc=0x74

$12=2 不<=0 不跳转



nextpc=74,运行正确。

指令35：

地址	汇编程序	16进制代码	预期结果
0x00000074	sllv $13,$12,$6	00cc6804	$13=4

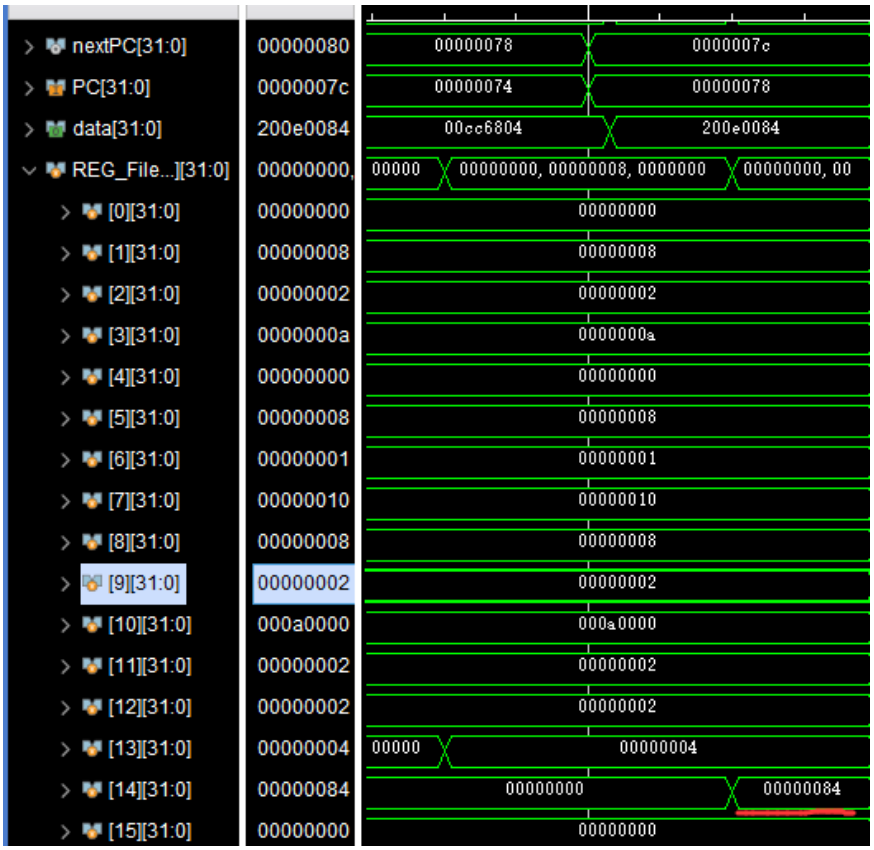
$13=2<<1 = 4 alures =4

$13=4,运行正确。

指令36：

地址	汇编程序	16进制代码	预期结果
0x00000078	addi $14,$0,132	200e0084	$14=0x84

$14=0+132 = 0x84 alures =132

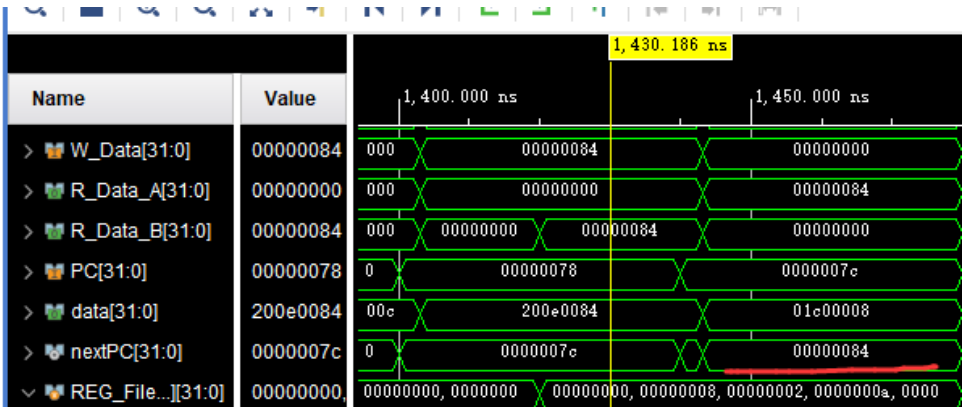


$14=0x84，运行正确

指令37：

地址	汇编程序	16进制代码	预期结果
0x0000007C	jr $14	01c00008	nextpc=0x84

跳转到寄存器地址 nextpc=0x84



nextpc=0x84，运行正确

指令38：

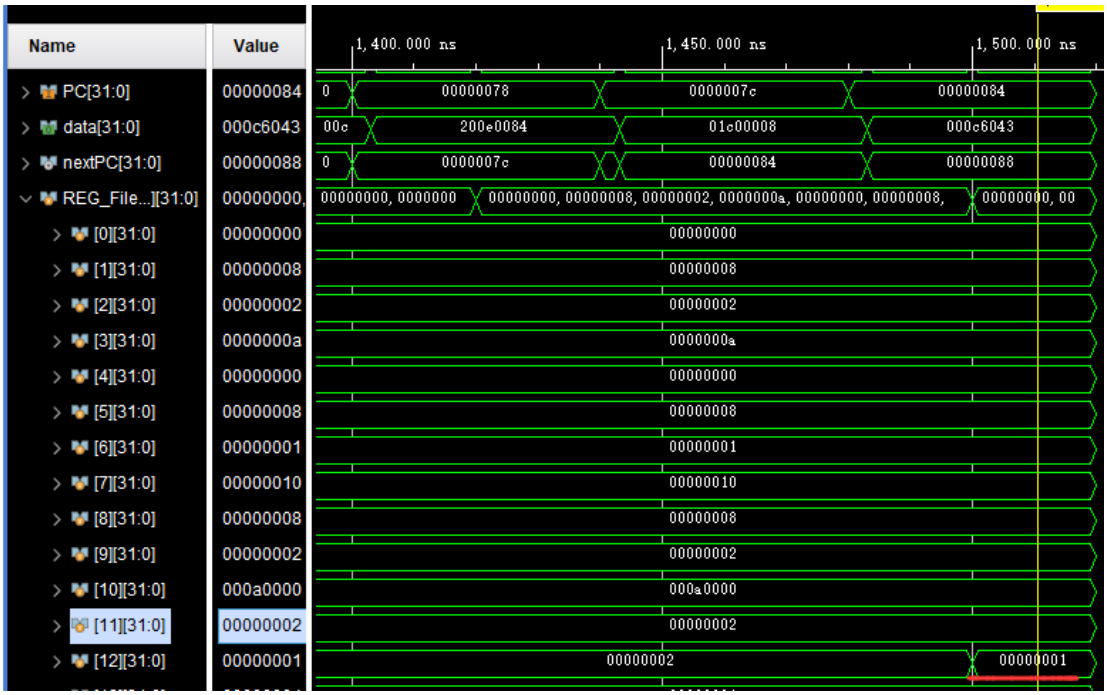
地址	汇编程序	16进制代码	预期结果
0x00000080	nop	00000000	被跳过

该指令被跳过

指令39：

地址	汇编程序	16进制代码	预期结果
0x00000084	sra $12,$12,1	000c6043	$12=1

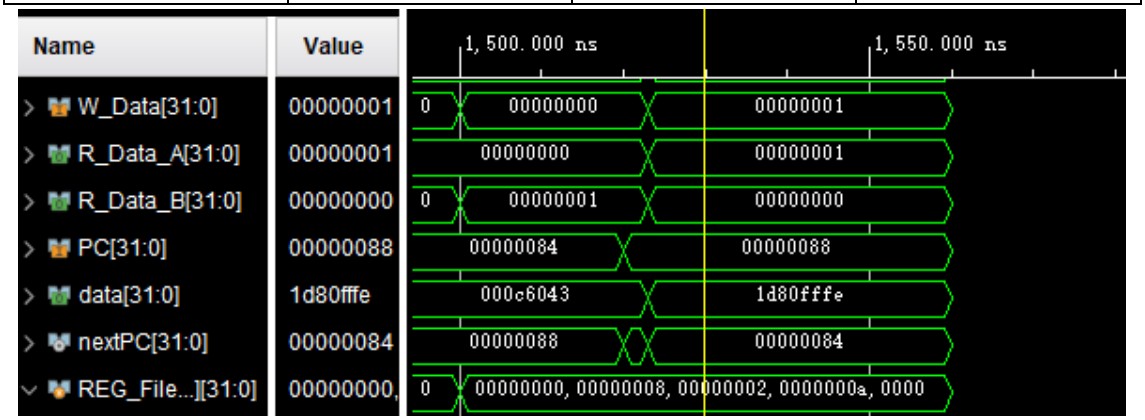
$12=2>>1=1 alures=1



$12=1，运行正确

指令40：

地址	汇编程序	16进制代码	预期结果
0x00000088	bgtz $12,-2	1d80fffe	跳转，nextpc=84



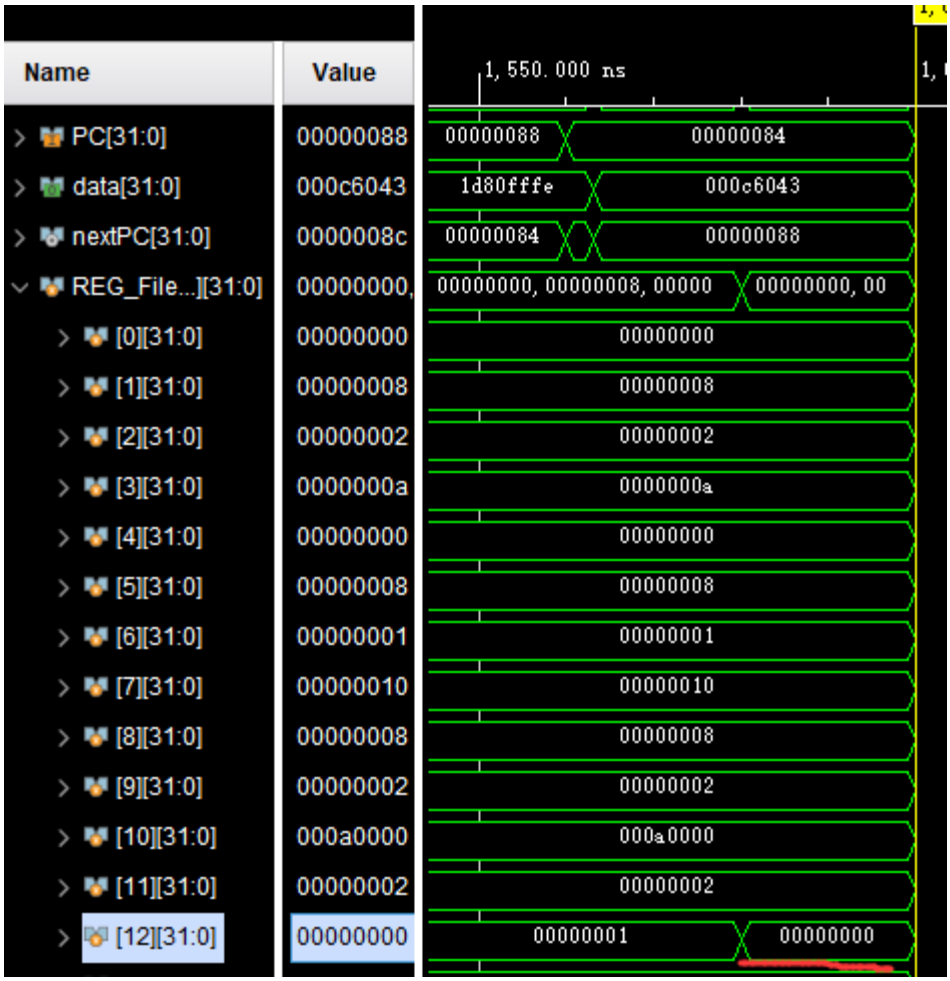
nextpc=0x84，运行正确

指令41：

地址	汇编程序	16进制代码	预期结果

0x00000084	sra $12,$12,1	000c6043	$12=0

$12=1>>1=0 alures=0

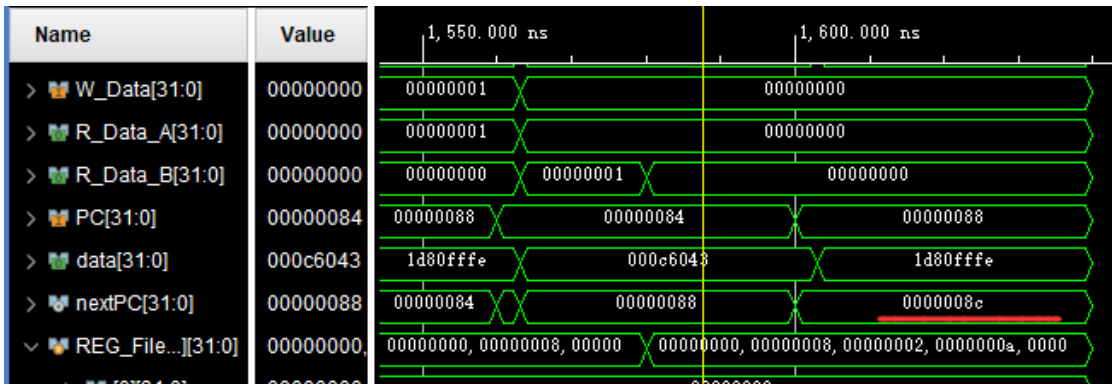


$12=0，运行正确。

指令42：

地址	汇编程序	16进制代码	预期结果
0x00000088	bgtz $12,-2	1d80fffe	不跳转，nextpc=0x8c

$12=0 不大于零 ，不跳转 nextpc=0x8c

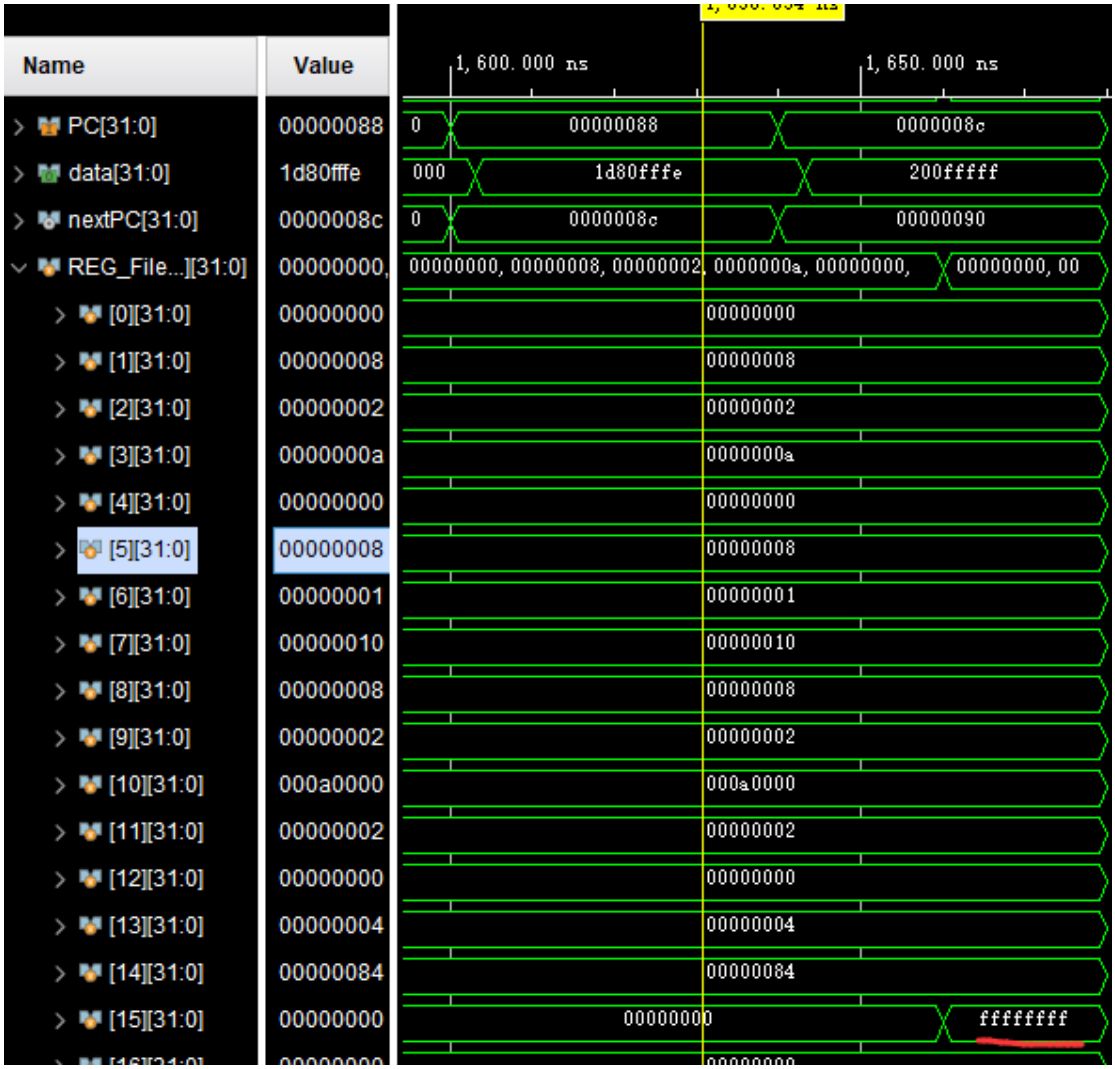


nextpc=0x8c，运行正确。

指令43：

地址	汇编程序	16进制代码	预期结果
0x0000008C	addi $15,$0,-1	200fffff	$15=-1

$15=0+ -1 = -1

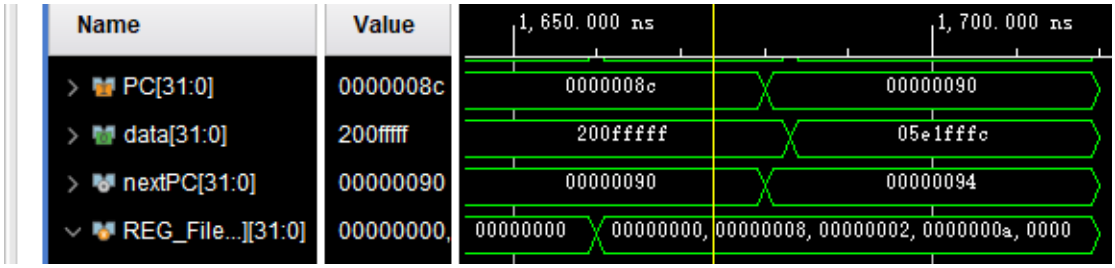


$15=-1,运行正确

指令43：

地址	汇编程序	16进制代码	预期结果
0x00000090	bgez $15,-4	05e1fffc	不跳转，nextpc=0x94

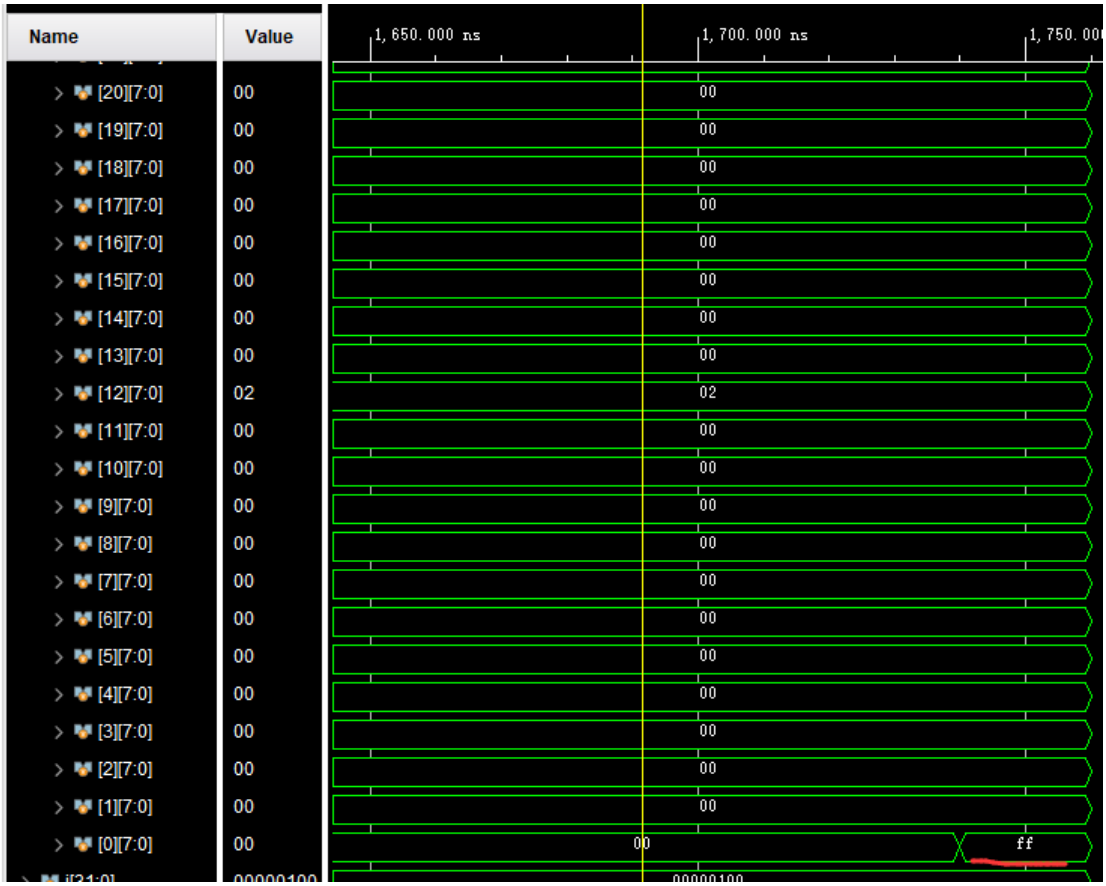
$15=-1 不大于等于零 ， 不跳转 nextpc=0x94



nextpc=0x94,运行正确

指令44：

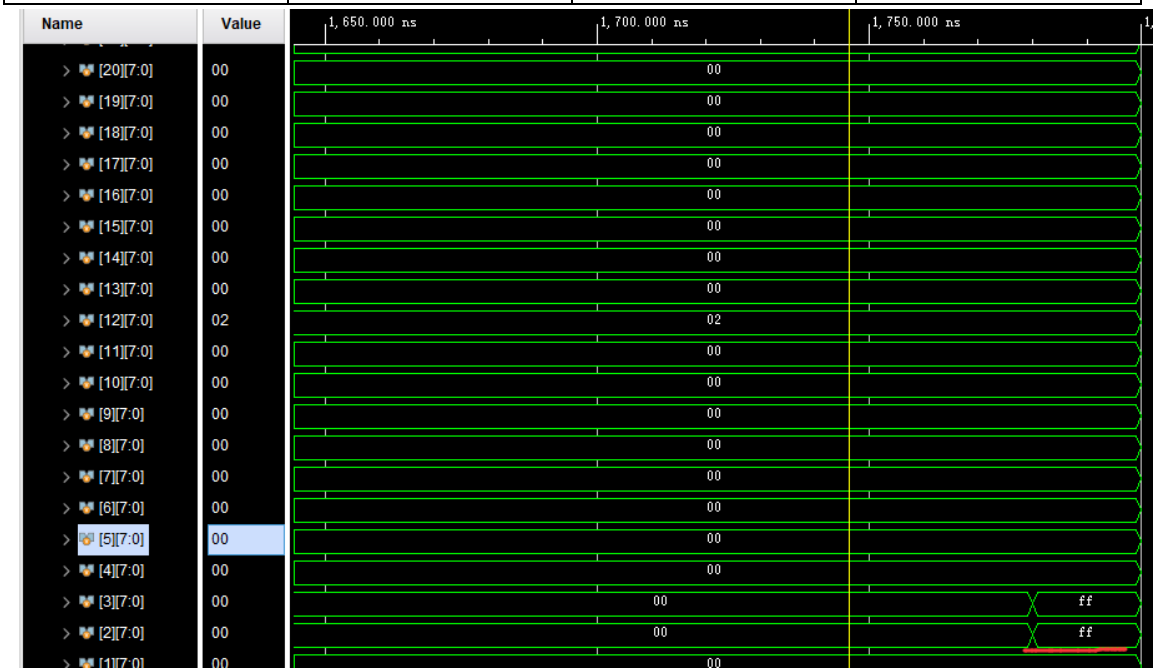
地址	汇编程序	16进制代码	预期结果
0x00000094	sb $15,0($s1)	a22f0000	mem[0]=-1



mem[0]=-1,运行正确

指令45：

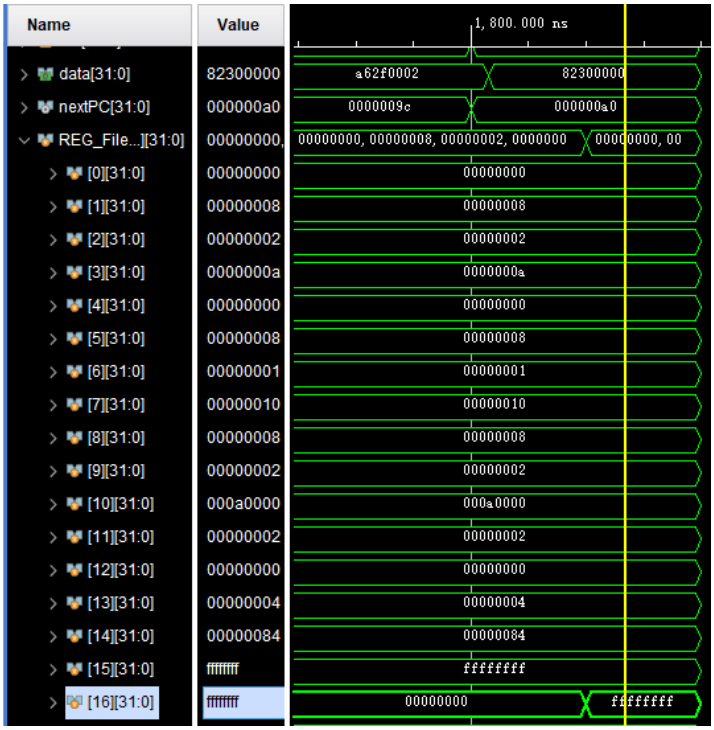
地址	汇编程序	16进制代码	预期结果
0x00000098	sh $15,2($s1)	a62f0002	mem[2:3]=0xffff



mem[2:3]=0xffff,运行正确。

指令46：

地址	汇编程序	16进制代码	预期结果
0x0000009C	lb $16,0($s1)	82300000	$16=0xffff



$16=0xffff,运行正确。

指令47：

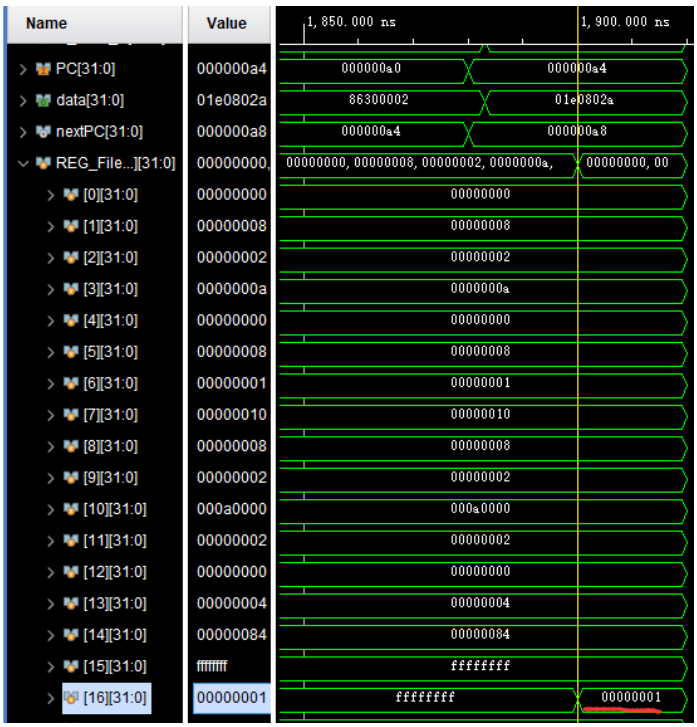
地址	汇编程序	16进制代码	预期结果
0x000000A0	lh $16,2($s1)	86300002	$16=0xffff

$16=0xffff,运行正确。

指令48：

地址	汇编程序	16进制代码	预期结果
0x000000A4	slt $16,$15,$0	01e0802a	$16=1

$15=-1 < 0 $16=1

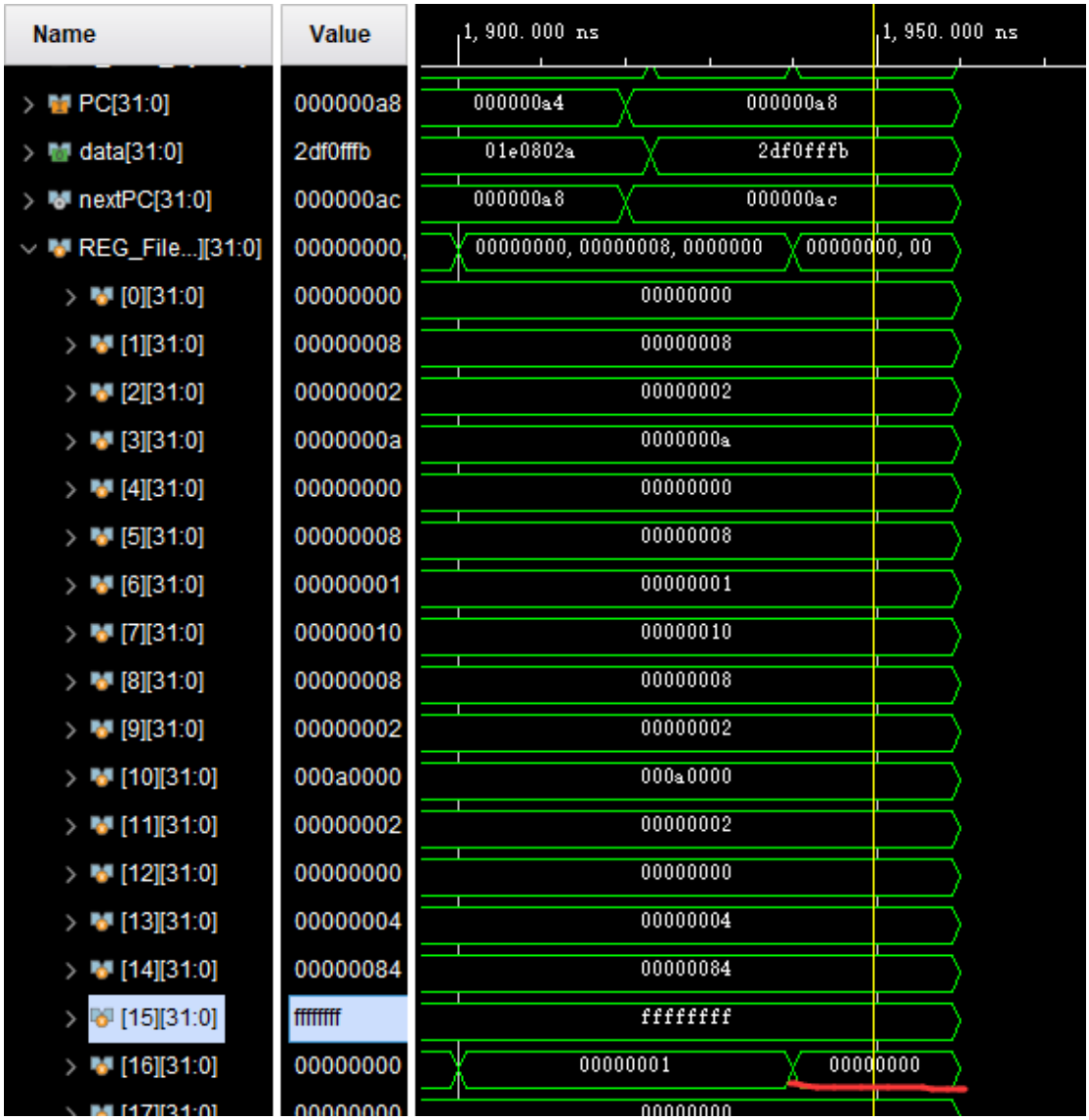


$16=1,运行正确。

指令49：

地址	汇编程序	16进制代码	预期结果
0x000000A8	sltiu $16,$15,-5	01e0802a	$16=0

$15=-5 < 0 $16=0

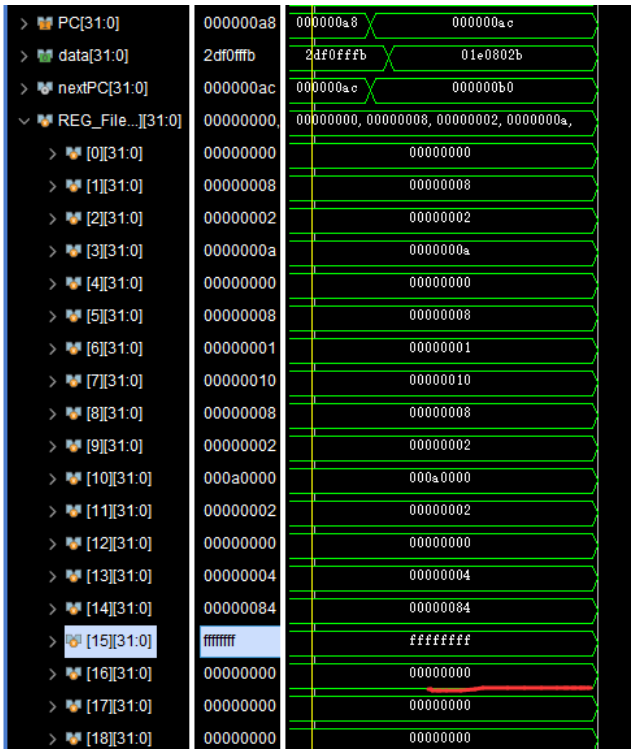


$16=0,运行正确。

指令50：

地址	汇编程序	16进制代码	预期结果
0x000000AC	sltu $16,$15,$0	01e0802b	$16=0

$15=-5 < 0 $16=0

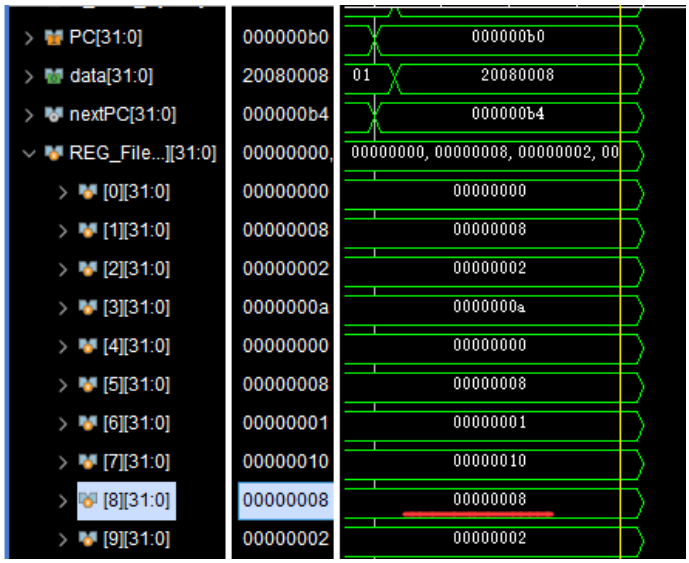


$16=0，运行正确。

指令50：

地址	汇编程序	16进制代码	预期结果
0x000000B0	addi $8,$0,8	20080008	$8=8

$8=0+8 = 8 $8=8

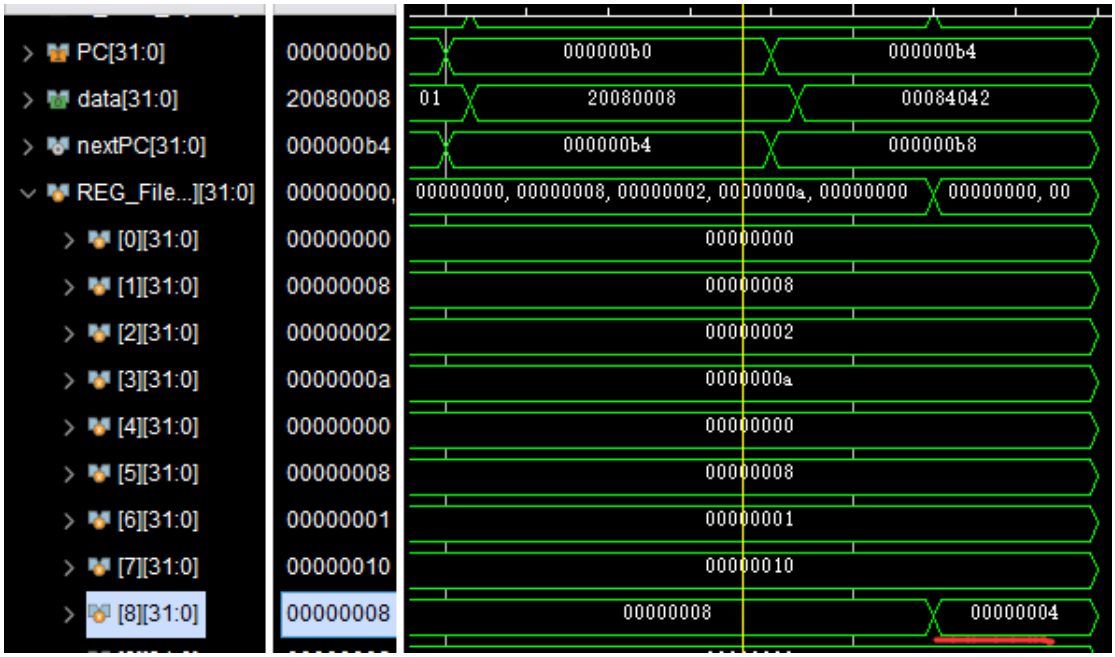


$8=8，运行正确。

指令51：

地址	汇编程序	16进制代码	预期结果
0x000000B4	srl $8,$8,1	00084042	$8=4

$8 = 8>>1 = 4 $8=4

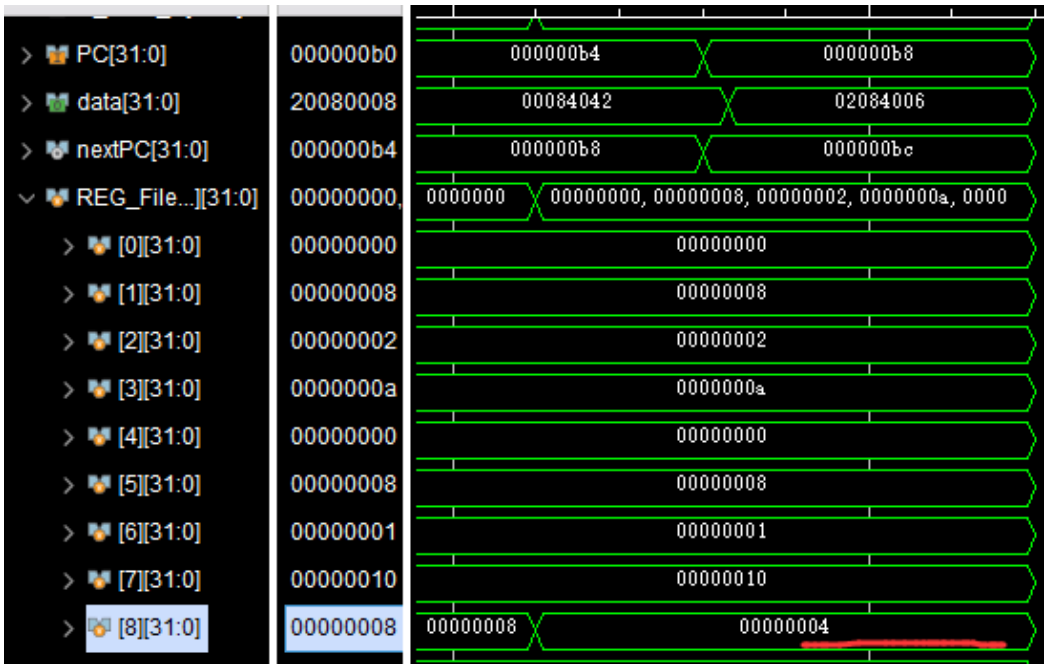


$8=4，运行正确。

指令52：

地址	汇编程序	16进制代码	预期结果
0x000000B8	srlv $8,$8,$16	02084006	$8=4

$8 = 4>>0 = 4 $8=4

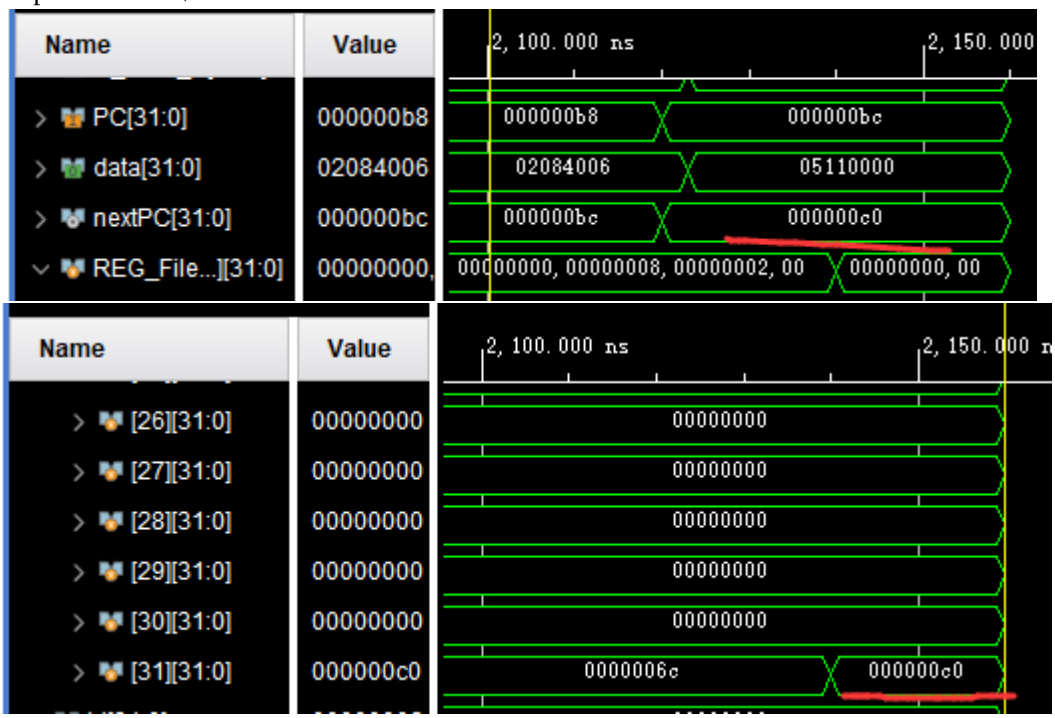


$8=4，运行正确。

指令53：

地址	汇编程序	16进制代码	预期结果
0x000000BC	bgezal $8,0	05110000	nextpc=0xC0 $31=0xC0

nextpc=0xC0　　$31=0xC0



nextpc=0xC0，$31=0xC0,运行正确

指令 54-59 放在下面一张图上

指令 54：

地址	汇编程序	16进制代码	预期结果
0x000000C0	addu $9,$8,$13	010d4821	$9=8

$9=4+4 = 8 运行正确

指令 55：

地址	汇编程序	16进制代码	预期结果
0x000000C4	subu $9,$8,$13	010d4823	$9=0

$9=4-4 = 0 运行正确

指令 56

地址	汇编程序	16进制代码	预期结果
0x000000C8	nor $9,$8,$13	010d4827	$9=-5

$9=4 nor 4 = -5 = 0xfffb 运行正确

指令 57

地址	汇编程序	16进制代码	预期结果
0x000000CC	xor $9,$8,$13	010d4826	$9=0

$9=4 xor 4 = 0　运行正确

指令 58

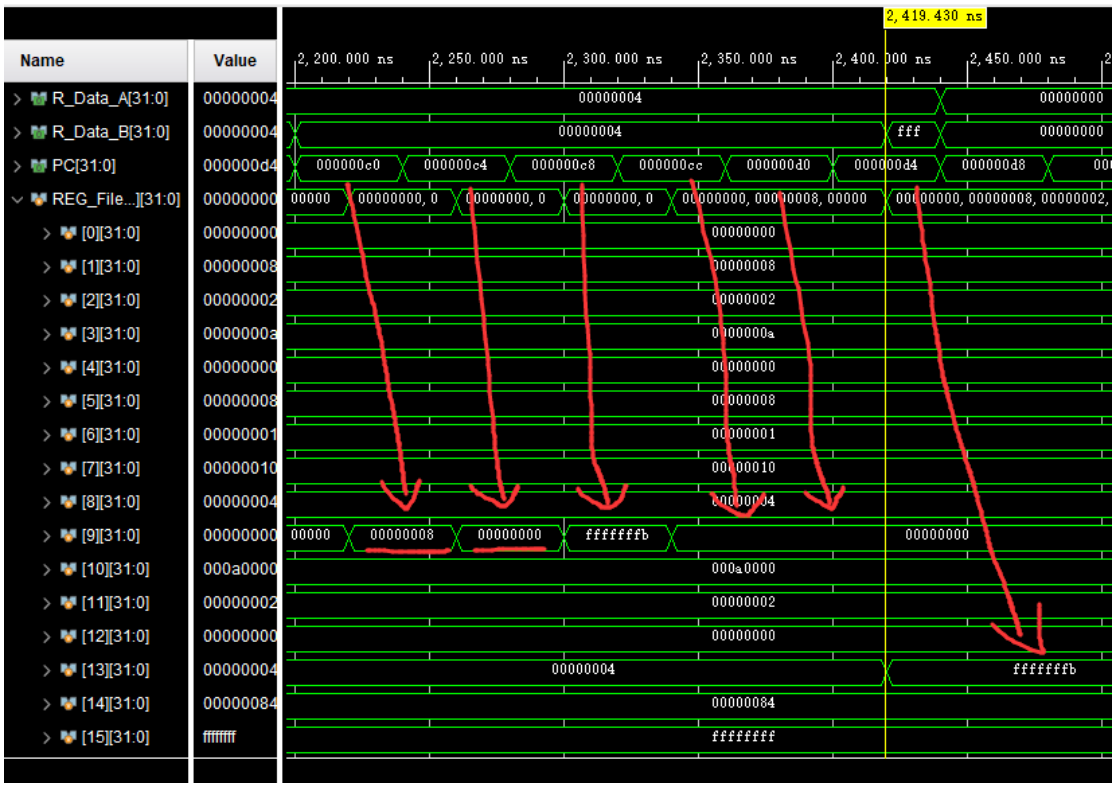
地址	汇编程序	16进制代码	预期结果
0x000000D0	srav $9,$8,$13	01a84807	$9=0

$9=4 >> 4 = 0　运行正确

指令 59

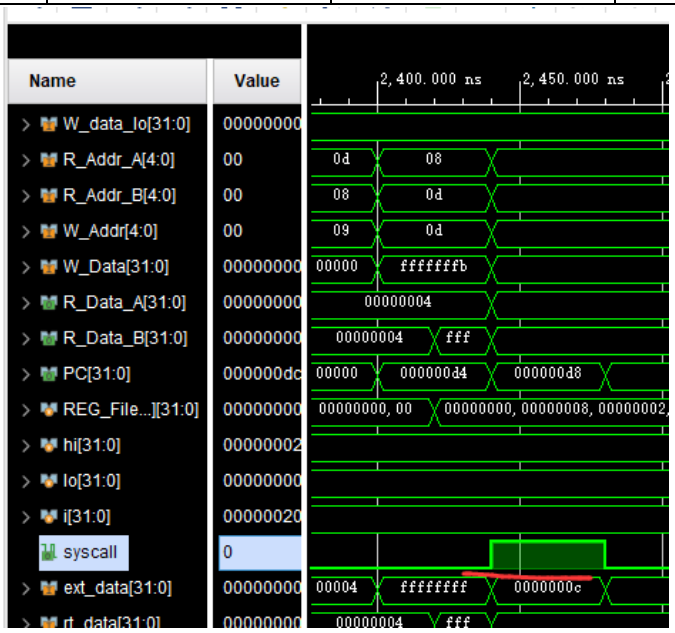
地址	汇编程序	16进制代码	预期结果
0x000000D4	xori $13,$8,-1	390dffff	$9=-5

$13=4 xor -1 = -5　运行正确



指令60：

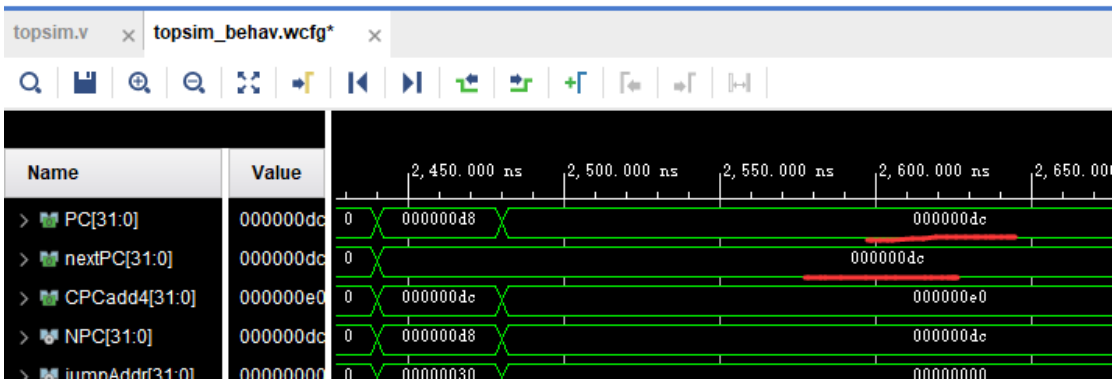
地址	汇编程序	16进制代码	预期结果
0x000000C0	syscall	0000000c	syscall=1



syscall=1，运行正确

指令55：

地址	汇编程序	16进制代码	预期结果
0x000000C4	**halt**	FC000000	停机，nextpc=pc



nextpc=pc,停机，运行正确。

以上是表格中所有指令运行完毕，仿真测速运行结果均正确。
综上，该实验中，实现了可以运行 mips 指令集中 47 条指令的 CPU。

烧板测试

首先改写 top 文件 2 能够符合烧板的需求,由于使用 rom ip 核心一直烧板不成功，这里换用自己实现的指令内存

指令内存内容如下：

```
`timescale 1ns / 1ps
module im(
    input [7:0] Addr,       //
    output reg [31:0]  instruction );//
    reg [31:0] regs [0:63]; //
initial
begin
regs[0]=32'h24010008;
regs[1]=32'h34020002;
regs[2]=32'h00411820;
regs[3]=32'h00622822;
regs[4]=32'h00a22024;
regs[5]=32'h00824025;
regs[6]=32'h00084040;
regs[7]=32'h1501fffe;
regs[8]=32'h28460004;
regs[9]=32'h28c70000;
regs[10]=32'h24e70008;
regs[11]=32'h10e1fffe;
regs[12]=32'hac220004;
regs[13]=32'h8c290004;
regs[14]=32'h240afffe;
regs[15]=32'h254a0001;
regs[16]=32'h0540fffe;
regs[17]=32'h304b0002;
regs[18]=32'h08000014;
regs[19]=32'h00824025;
regs[20]=32'h00000000;
regs[21]=32'h3c0a000a;
regs[22]=32'h01660018;
regs[23]=32'h00006010;
regs[24]=32'h00006012;
regs[25]=32'h0166001a;
regs[26]=32'h0c00001b;
regs[27]=32'h0590fffe;
regs[28]=32'h1980fffd;
regs[29]=32'h00cc6804;
regs[30]=32'h200e0084;
regs[31]=32'h01c00008;
regs[32]=32'h00000000;
regs[33]=32'h000c6043;
regs[34]=32'h1d80fffe;
regs[35]=32'h200fffff;
```

```
regs[36]=32'h05e1fffc;
regs[37]=32'ha22f0000;
regs[38]=32'ha62f0002;
regs[39]=32'h82300000;
regs[40]=32'h86300002;
regs[41]=32'h01e0802a;
regs[42]=32'h2df0fffb;
regs[43]=32'h01e0802b;
regs[44]=32'h20080008;
regs[45]=32'h00084042;
regs[46]=32'h02084006;
regs[47]=32'h05110000;
regs[48]=32'h010d4821;
regs[49]=32'h010d4823;
regs[50]=32'h010d4827;
regs[51]=32'h010d4826;
regs[52]=32'h01a84807;
regs[53]=32'h390dffff;
regs[54]=32'h0000000c;
regs[55]=32'hfc000000;

end
always @(Addr)
    instruction=regs[Addr] ;
endmodule
```

重新改写后的 top 模块如下：

```
`timescale 1ns / 1ps
module top(
        input clkin,
        input clkdisplay,
        input reset,
        input high_low_choose,
        input pc_npc_choose,
        output [6:0] sm_duan,//段码
        output [3:0] sm_wei,//哪个数码管
        output  syscall,
        output [7:0] lPC

        //output [31:0] aluRes,
        //output [31:0] instruction
       );
wire [31:0] aluRes;
wire [31:0] PC;
```

```
wire [31:0] instruction;
assign lPC=PC[9:2];

//数据存储器
wire[31:0] memreaddata;
// 指令存储器
//wire [31:0] instruction;
reg[7:0] Addr;
// CPU 控制信号线
wire reg_dst,memread, memwrite, memtoreg,alu_src,alu_zeroinput,ExtOp;
wire[3:0] aluop;
wire regwrite;
wire jmp,jal,jr,bne,beq,bgez,bgezal,bgtz,blez,bltz,bltzal;
wire mult,div,mflo,mfhi,PCWrite,noop,halt;
wire [1:0]storemux;
// ALU 控制信号线
wire ZF,OF,CF,PF; //alu 运算标志
wire[31:0] hi,lo; //alu 运算结果
// ALU 控制信号线
wire[4:0] aluCtr;//根据 aluop 和指令后 6 位 选择 alu 运算类型
//
wire[31:0] input2;
wire[31:0] nextPC;
wire [15:0]data;
//link
wire [4:0] linkaddr;
//datatoreg
wire link_reg_write;

//PCctr
wire [31:0] CPCadd4;
// 寄存器信号线
wire[31:0] RsData, RtData;
wire[31:0] expand; wire[4:0] shamt;
wire [4:0]regWriteAddr;
wire[31:0]regWriteData;
//link
wire [4:0]muxlinkaddr;
assign shamt=instruction[10:6];
assign regWriteAddr = reg_dst ? instruction[15:11] : instruction[20:16]
;//写寄存器的目标寄存器来自 rt 或 rd
assign data=aluRes[15:0];
//assign regWriteData = memtoreg ? memreaddata : aluRes; //写入寄存器的数
据来自 ALU 或数据寄存器
```

```
//assign input2 = alu_src ? expand : RtData; //ALU 的第二个操作数来自寄存器堆输出或指令低 16 位的符号扩展
// 例化指令存储器

// rom_top trom(
//              .Clk(clkin),
//              .Rst(1'b0),
//              .PC(PC),
//              .data(instruction)
//          );
im trom(
    .Addr(PC[9:2]),
    .instruction(instruction)
);
// 实例化控制器模块
ctr mainctr(
        .opCode(instruction[31:26]),
        .funct(instruction[5:0]),
        .rt(instruction[20:16]),
        .regDst(reg_dst),
        .aluSrc(alu_src),
        .aluZeroinput(alu_zeroinput),
        .memToReg(memtoreg),
        .regWrite(regwrite),
        .memRead(memread),
        .memWrite(memwrite),
        .ExtOp(ExtOp),
        .aluop(aluop),
        .jmp(jmp),
        .jal(jal),
        .jr(jr),
        .bne(bne),
        .beq(beq),
        .bgez(bgez),
        .bgezal(bgezal),
        .bgtz(bgtz),
        .blez(blez),
        .bltz(bltz),
        .bltzal(bltzal),
        .mult(mult),
        .div(div),
        .mflo(mflo),
        .mfhi(mfhi),
        .PCWrite(PCWrite),
```

```
        .syscall(syscall),
        .noop(noop),
        .halt(halt),
        .storemux(storemux)
    );
aluinput2 aluinput2_i(
              .ext_data(expand),
              .rt_data(RtData),
              .aluSrc(alu_src),
              .aluZeroinput(alu_zeroinput),
              .alu_input2(input2)
          );
//  实例化 ALU 控制模块
aluctr aluctr1(
           .ALUOp(aluop),
           .funct(instruction[5:0]),
           .ALUCtr(aluCtr)
       );
//link
link link1(
         .jal(jal),
         .bgezal(bgezal),
         .bltzal(bltzal),
         .ZF(ZF),
         .PF(PF),
         .reg_W_Addr(regWriteAddr),
         .W_Addr(muxlinkaddr)
     );
PCctr PCctr1(
          .clkin(clkin),
          .reset(reset),
          .PCWrite(PCWrite),
          .target(instruction[25:0]),
          .imm_16(instruction[15:0]),
          .R_Data_A(RsData),
          .jmp(jmp),
          .jal(jal),
          .jr(jr),
          .bne(bne),
          .beq(beq),
          .bgez(bgez),
          .bgezal(bgezal),
          .bgtz(bgtz),
          .blez(blez),
```

```
          .bltz(bltz),
          .bltzal(bltzal),
          .ZF(ZF),
          .PF(PF),
          .PC(PC),
          .CPCadd4(CPCadd4),
          .nextPC(nextPC)
      );
dataToReg dataToReg1(
              .memToReg(memtoreg),
              .jal(jal),
              .regWrite(regwrite),
              .bgezal(bgezal),
              .bltzal(bltzal),
              .ZF(ZF),
              .PF(PF),
              .PCAdd4(CPCadd4),
              .alures(aluRes),
              .mem_data(memreaddata),
              .w_data(regWriteData),
              .link_reg_write(link_reg_write)
          );
// 。。。。。。。。。。。。。。。。。。。。。。。。。。。。。实例化寄存器模块
RegFile regfile(
            .Clk(clkin),
            .Clr(reset),
            .Write_Reg(link_reg_write),
            .R_Addr_A(instruction[25:21]),
            .R_Addr_B(instruction[20:16]),
            .W_Addr(muxlinkaddr),
            .W_Data(regWriteData),
            .mfhi(mfhi),
            .mflo(mflo),
            .mult(mult),
            .div(div),
            .W_data_hi(hi),
            .W_data_lo(lo),
            .R_Data_A(RsData),
            .R_Data_B(RtData)
        );
// 。。。。。。。。。。。。。。。。。。。。。。。。。。。。。实例化 ALU 模块
alu alu(
        .shamt(shamt),
        .input1(RsData), //写入 alu 的第一个操作数必是 Rs
```

```
        .input2(input2),
        .aluCtr(aluCtr),
        .ZF(ZF),
        .OF(OF),
        .CF(CF),
        .PF(PF),
        .hi(hi),
        .lo(lo),
        .aluRes(aluRes)
    );
//实例化符号扩展模块
signext signext(
            .inst(instruction[15:0]),
            .ExtOp(ExtOp),
            .data(expand)
        );
//实例化数据存储器
dram dm(
         .clk(clkin),
         .memwrite(memwrite),
         .reset(reset),
         .flag(storemux),
         .addr(aluRes[7:0]),
         .write_data(RtData),
         .read_data(memreaddata)
     );
//.............................实例化数码管显示模块
display Smg(
    .clk(clkdisplay),
    .high_low_choose(high_low_choose),
    .pc_npc_choose(pc_npc_choose),
    .alures(aluRes),
    .pc(PC),
    .npc(nextPC),
    .sm_wei(sm_wei),
    .sm_duan(sm_duan)
);

endmodule
```

同时也需要修改 display 模块

```
`timescale 1ns / 1ps
module display(
```

```
    input clk,
    input high_low_choose,
    input pc_npc_choose,
    input [31:0] alures,
    input [31:0] pc,
    input [31:0] npc,
    output [3:0] sm_wei,
    output [6:0] sm_duan
);
//分频
integer clk_cnt=0;
reg clk_400Hz;
wire [15:0] low_data;
wire [15:0] high_data;
wire [15:0] data;
wire [15:0] pcdata;
assign low_data=alures[15:0];
assign high_data=alures[31:16];
assign pcdata=pc_npc_choose?{pc[7:0],npc[7:0]}:low_data;
assign data=high_low_choose?high_data:pcdata;

always @(posedge clk)
    if(clk_cnt==32'd100_000)
        begin clk_cnt <= 1'b0;
        clk_400Hz = ~clk_400Hz;
        end
    else clk_cnt = clk_cnt + 1'b1;
//位控制

reg [3:0]wei_ctrl=4'b1110;
always @(posedge clk_400Hz)
wei_ctrl = {wei_ctrl[2:0],wei_ctrl[3]}; //位控制
reg [3:0]duan_ctrl;

always @(wei_ctrl)
case(wei_ctrl)
4'b1110:duan_ctrl=data[3:0];
4'b1101:duan_ctrl=data[7:4];
4'b1011:duan_ctrl=data[11:8];
4'b0111:duan_ctrl=data[15:12];
default:duan_ctrl=4'hf;
 endcase

//解码模块
```

```
reg [6:0]duan;
always @(duan_ctrl)
case(duan_ctrl)
4'h0:duan=7'b100_0000;//0
4'h1:duan=7'b111_1001;//1
4'h2:duan=7'b010_0100;//2
4'h3:duan=7'b011_0000;//3
4'h4:duan=7'b001_1001;//4
4'h5:duan=7'b001_0010;//5
4'h6:duan=7'b000_0010;//6
4'h7:duan=7'b111_1000;//7
4'h8:duan=7'b000_0000;//8
4'h9:duan=7'b001_0000;//9
4'ha:duan=7'b000_1000;//a
4'hb:duan=7'b000_0011;//b
4'hc:duan=7'b100_0110;//c
4'hd:duan=7'b010_0001;//d
4'he:duan=7'b000_0111;//e
4'hf:duan=7'b000_1110;//f
// 4'hf:duan=7'b111_1111;//不显示
default : duan = 7'b100_0000;//0
endcase

//------------------------------------------------------------
assign sm_wei = wei_ctrl;
assign sm_duan = duan;
endmodule
```

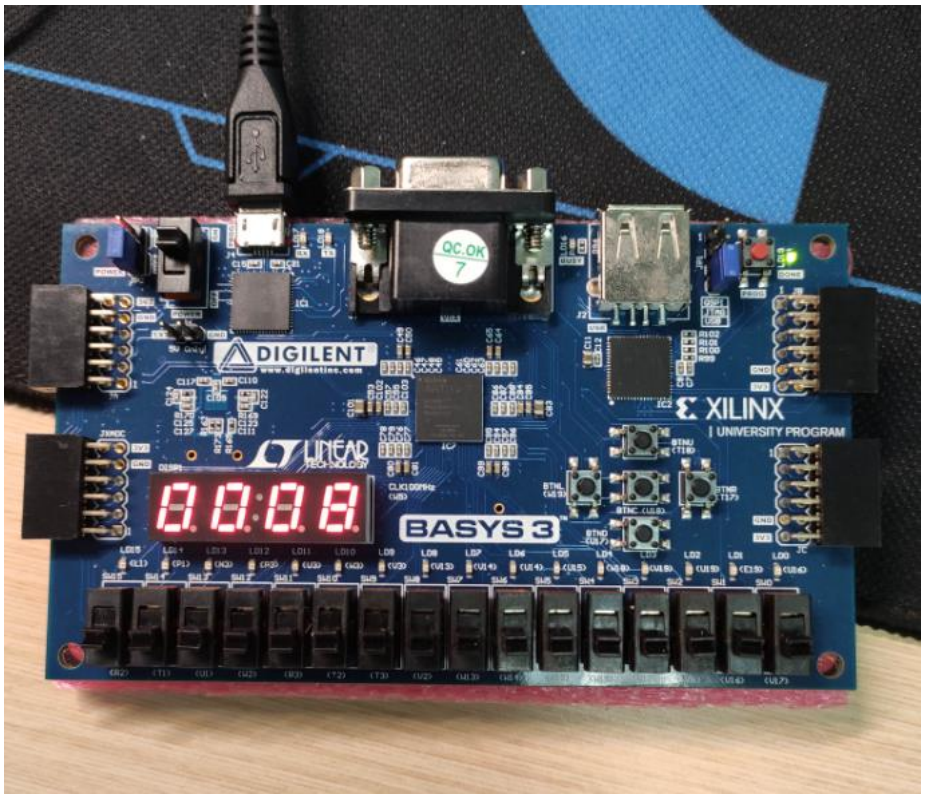
然后根据要求编写约束文件

这里用中间按钮控制时钟信号，用上按钮控制清零信号，用从右一个和第二个开关控制显示 pc 和 nextpc 或者显示 alures 的低 16 位或者显示 alures 的高 16 位。

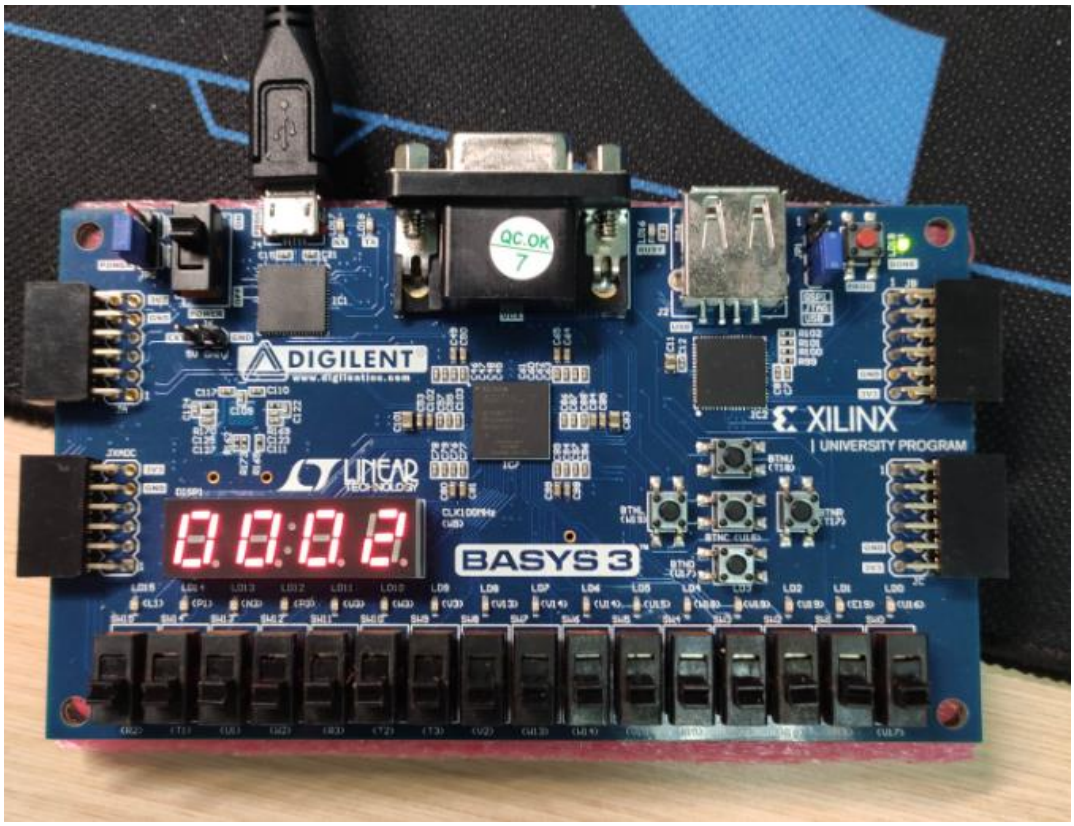
```
set_property PACKAGE_PIN U2 [get_ports {sm_wei[0]}]
set_property PACKAGE_PIN U4 [get_ports {sm_wei[1]}]
set_property PACKAGE_PIN V4 [get_ports {sm_wei[2]}]
set_property PACKAGE_PIN W4 [get_ports {sm_wei[3]}]
set_property PACKAGE_PIN U7 [get_ports {sm_duan[6]}]
set_property PACKAGE_PIN V5 [get_ports {sm_duan[5]}]
set_property PACKAGE_PIN U5 [get_ports {sm_duan[4]}]
set_property PACKAGE_PIN V8 [get_ports {sm_duan[3]}]
set_property PACKAGE_PIN U8 [get_ports {sm_duan[2]}]
set_property PACKAGE_PIN W6 [get_ports {sm_duan[1]}]
set_property PACKAGE_PIN W7 [get_ports {sm_duan[0]}]
```

```
set_property PACKAGE_PIN W5  [get_ports {clkdisplay}]
set_property PACKAGE_PIN U18 [get_ports {clkin}]
set_property PACKAGE_PIN T18 [get_ports {reset}]
set_property PACKAGE_PIN V17 [get_ports {high_low_choose}]
set_property PACKAGE_PIN V16 [get_ports {pc_npc_choose}]
set_property PACKAGE_PIN U16 [get_ports {lPC[0]}]
set_property PACKAGE_PIN E19 [get_ports {lPC[1]}]
set_property PACKAGE_PIN U19 [get_ports {lPC[2]}]
set_property PACKAGE_PIN V19 [get_ports {lPC[3]}]
set_property PACKAGE_PIN W18 [get_ports {lPC[4]}]
set_property PACKAGE_PIN U15 [get_ports {lPC[5]}]
set_property PACKAGE_PIN U14 [get_ports {lPC[6]}]
set_property PACKAGE_PIN V14 [get_ports {lPC[7]}]

set_property PACKAGE_PIN L1 [get_ports {syscall}]

set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_wei[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sm_duan[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {clkdisplay}]
set_property IOSTANDARD LVCMOS33 [get_ports {clkin}]
set_property IOSTANDARD LVCMOS33 [get_ports {reset}]
set_property IOSTANDARD LVCMOS33 [get_ports {high_low_choose}]
set_property IOSTANDARD LVCMOS33 [get_ports {pc_npc_choose}]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clkin]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lPC[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {syscall}]
```

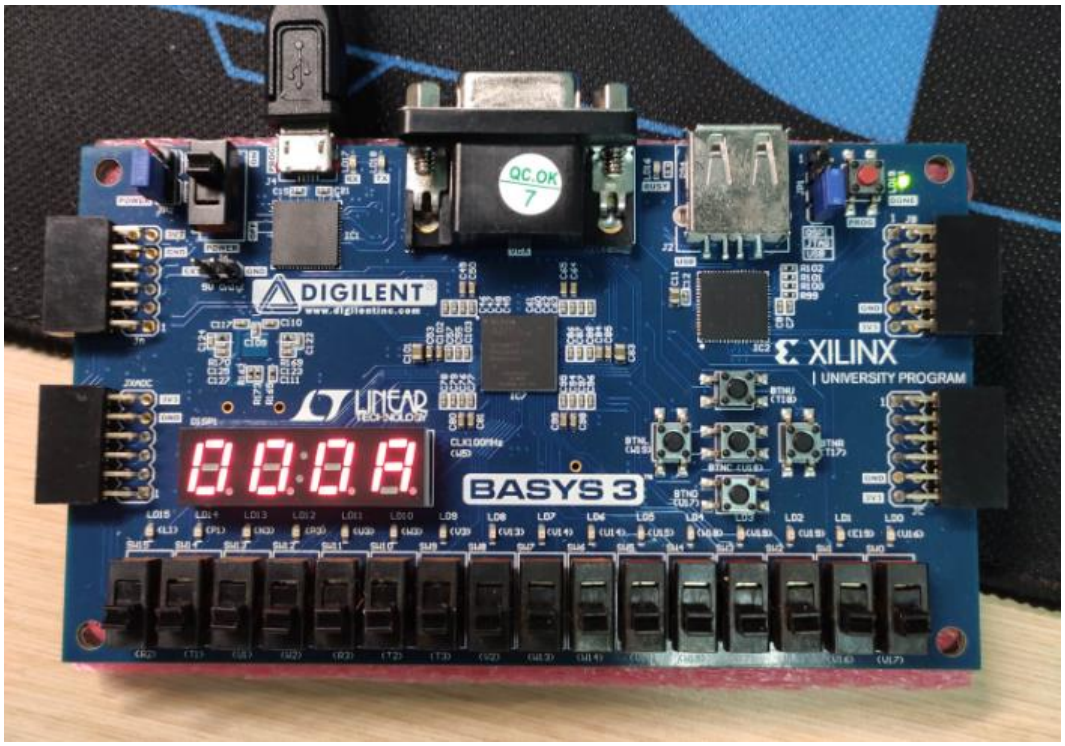
接下来进行烧板测试，对测试数据的连续前 5 条进行测试。



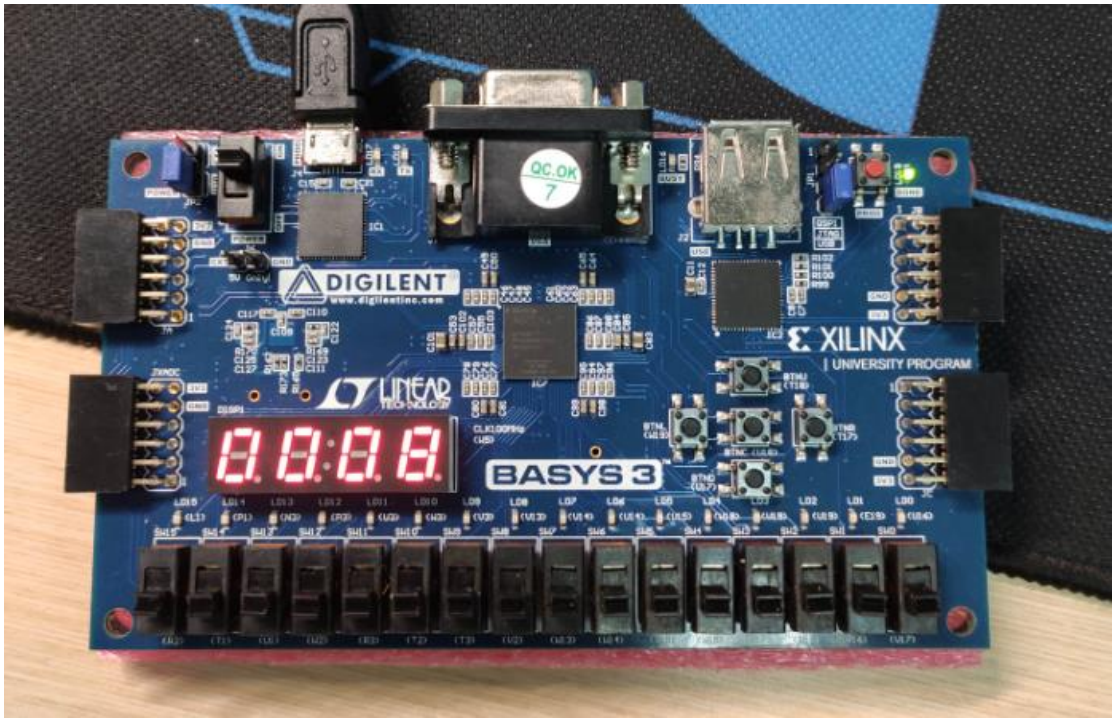
addiu　$1,$0,8，alures=8,运行正确



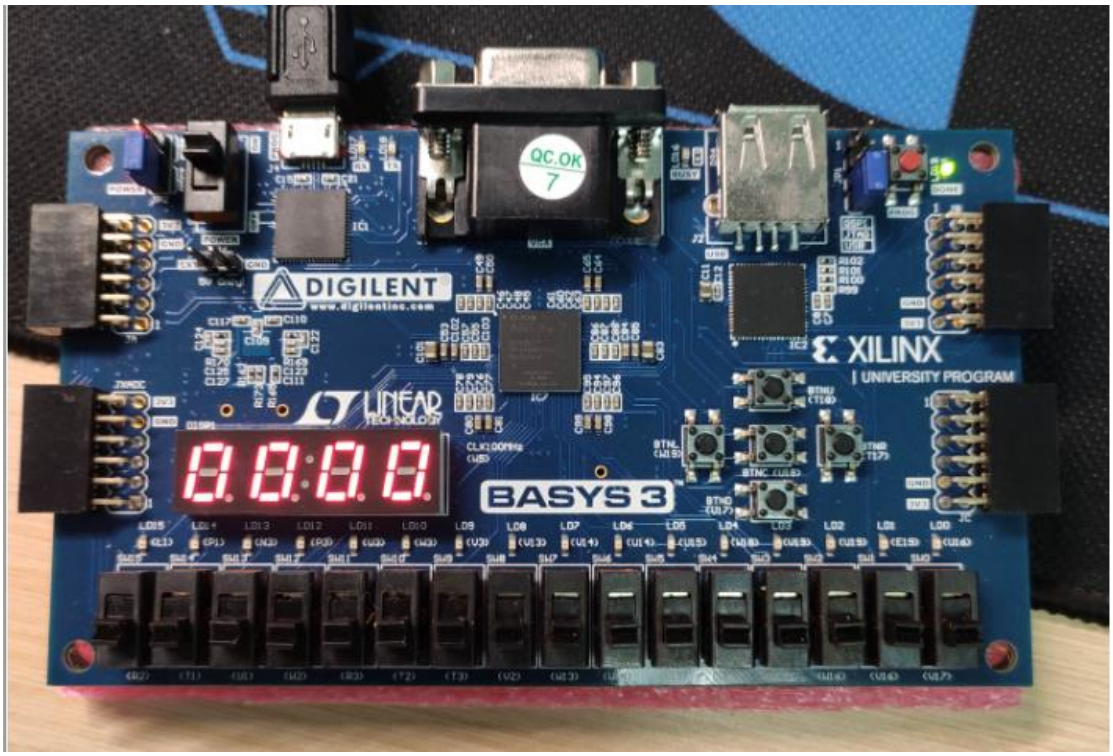
ori　$2,$0,2 alures=2,运行正确



add　$3,$2,$1，alures= 0x0a 运行正确



sub　$5,$3,$2，alures=10-2= 8 运行正确



and　$4,$5,$2，alures=8 & 2= 0 运行正确

经过更多的测试，cpu 的烧板运行正常，此处不再赘述。

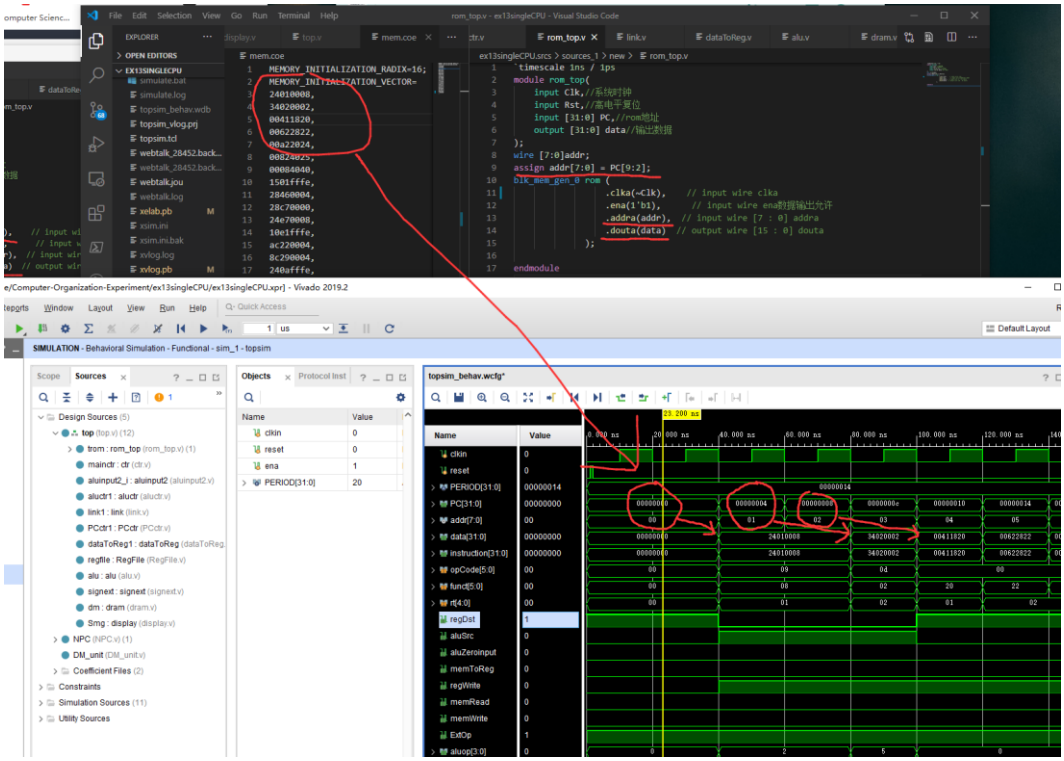
六．实验心得

本次实验是一次对综合实现CPU的实验，虽然在之前的实验中已经对一些指令进行了测速，但是为了在本次实验中扩展指令，还是对整体实现的架构进行了较大的修改，例如，对与各种对$ra进行操作的jal，bgtzal，blezal指令中，需要根据信号选择对应的写入寄存区和写入数据，对于blez，bgez，等等和零有关的branch指令，alu的运算数的来源除了来自立即数和寄存器读取值之外，还需要额外选择来自0来和零做比较，同时也需要在alu模块中加入表明运算结果正负号的PF信号来判断是否要执行这些分支指令，在这些扩展指令的实现过程中，对整个原始的设计架构进行了较大的改变，同时，将尽可能能够模块化的部分模块化，才有了不同于只能执行基本指令的cpu的aluinput2，link，dataToReg模块来执行这些额外的指令，最终一共实现了47条mips指令。

在整个实现的过程中，第一周也感觉很困难，不知道如何下手，但是通过做前期的准备工作，例如整理指令对应的各种信号表，以及需要理解各个指令的数据通路，才能有一个结构清晰的后续完成过程，在最后遇到问题的时候定位问题较快，按照错误指令的数据通路逐个排查，很快就能发现有问题的代码。

在实现的过程中，同时也遇到了很多问题，一是vivado中内存ip核心的时钟使能问题，

由于该内存不是实时变化的，往往延后一个周期才能拿到正确的数据，导致一开始的指令如果需要跳转，则会发生错误，如下图：



感谢同学的细心指导，只需要在读取内存的同时，不采取CPU的全局时钟，而是通过一层模块的封装，自己实现一个内部的时钟，在每次读取地址发生变化的时候，将封装模块的时钟强制发生两个周期的变换，就可以解决问题。

```
1  `timescale 1ns / 1ps
2  module rom_top(
3      input Clk,//系统时钟
4      input Rst,//高电平复位
5      input [31:0] PC,//rom地址
6      output [31:0] data//输出数据
7  );
8  wire [7:0]addr;
9  reg clk;
10 assign addr[7:0] = PC[9:2];
11 initial begin
12     clk=0;
13     #1;
14     clk=1;
15     #1
16     clk=0;
17     #1;
18     clk=1;
19 end
20 always @(addr)begin
21     clk=0;
22     #1;
23     clk=1;
24     #1;
25     clk=0;
26     #1;
27     clk=1;
28 end
29 blk_mem_gen_0 rom (
30                     .clka(clk),    // input wire clka
31                     .ena(1'b1),      // input wire ena
32                     .addra(addr),  // input wire [7 :
33                     .douta(data)  // output wire [15
34                 );
35
36 endmodule
```

由于代码量较大，在实现过程总难免发生实现bug，通过在vivado的仿真测试中设置断点，添加关键信号的波形展示，逐步修正了实现过程中的许多bug，同时使用git开发记录，使得实现的过程每一步的bug修复都有迹可循。

在代码改崩溃了之后，直接git reset HASH --hard恢复以前的现场，避免了很多修改的后顾之忧。